28th International Conference on Principles and Practice of Constraint Programming

CP 2022, July 31–August 8, 2022, Haifa, Israel

Edited by

Christine Solnon

LIPICS

Editors

Christine Solnon
INSA Lyon, CITI, Inria Chroma, France
christine.solnon@insa-lyon.fr

ACM Classification 2012
Computing methodologies → Artificial intelligence

ISBN 978-3-95977-240-2

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/978-3-95977-240-2.

Publication date
July, 2022

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.2022.0

ISBN 978-3-95977-240-2 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

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 Talk

Regular Papers

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This volume contains the proceedings of the *28th International Conference on Principles and Practice of Constraint Programming* (CP 2022), which was held in Haifa, Israel, August 1-5, 2022. Detailed information about the conference are available at `https://cp2022.a4cp.org`. CP 2022 was part of the *Federated Logic Conference* (FLoC) which is held every four years and brings together several leading international conferences related to logic for computer science. FLoC 2022 included 12 conferences, and CP was colocated with the *25th International Conference on Theory and Applications of Satisfiability Testing* (SAT 2022) and the *38th International Conference on Logic Programming* (ICLP 2022), among other conferences.

Held annually, CP is the premier international conference on constraint programming. As is customary for CP, papers could be submitted to multiple tracks. A first technical track was concerned with all aspects of computing with constraints, including theory, algorithms, environments, languages, models, and systems. A second track, chaired by Helmut Simonis (University College Cork), was dedicated to applications of CP. The last three tracks were dedicated to interdisciplinary research at the intersection between constraint programming and other directly related fields: a *Machine Learning* track, chaired by Andrea Lodi (Cornell Tech), an *Operations Research* track, chaired by Sophie Demassey (Mines ParisTech), and a *Trustworthy Decision Making* track, co-chaired by Nadjib Lazaar (LIRMM), Pierre Marquis (Université d'Artois), and Barry O'Sullivan (University College Cork).

78 papers have been submitted to these tracks, and 40 of them have been accepted. Each paper has been reviewed by at least three members of the program committee. We considered a double blind reviewing process, meaning that authors and reviewers were anonymous to each other. Authors had the opportunity to answer reviewers and clarify possible misunderstandings through a rebuttal phase. For each paper, a senior program committee member was in charge of conducting a discussion with reviewers to find a consensus, and of writing a meta-review that summarised pros and cons. Finally, virtual meetings were organised between meta-reviewers, track chairs, and the program chair to agree on final decisions.

Four papers that had an average score greater than or equal to 2 (possible scores ranged from 3, corresponding to a strong accept, to -3, corresponding to a strong reject) were nominated by at least one program committee member for receiving a best paper award:

- *Selecting SAT Encodings for Pseudo-Boolean and Linear Integer Constraints*, from Felix Ulrich-Oltean, Peter Nightingale and James Alfred Walker;
- *A Constraint Programming Approach to Ship Refit Project Scheduling*, from Raphaël Boudreault, Vanessa Simard, Daniel Lafond and Claude-Guy Quimper;
- *Exploiting Functional Constraints in Generating Dominance Breaking Nogoods for Constraint Optimization*, from Jimmy H. M. Lee and Allen Z. Zhong;
- *Peel-and-Bound: Generating Stronger Relaxed Bounds with Multivalued Decision Diagrams*, from Isaac Rudich, Quentin Cappart and Louis-Martin Rousseau.

The best two of them have been selected by a vote of senior PC members: the best paper prize was awarded to Isaac Rudich, Quentin Cappart and Louis-Martin Rousseau, and the best student paper prize was awarded to Jimmy H. M. Lee and Allen Z. Zhong.

We had the great honour and pleasure to have an invited talk given by Donald E. Knuth (Stanford University), whose next fascicle of *The Art of Computer Programming* is intended to be a solid introduction to techniques for solving Constraint Satisfaction Problems. An abstract of this talk is included in these proceedings. There was also two plenary FLoC invited speakers: Catuscia Palamidessi (INRIA Saclay, France), and Orna Kupferman (Hebrew University of Jerusalem, Israel).

Besides the paper tracks and invited talks, CP also had many other events, handled by special chairs: Ciaran McCreesh (University of Glasgow) organised the three workshops on the first day of the conference; Clément Carbonnel (LIRMM) selected tutorials for the main conference; Hélène Verhaeghe (Polytechnique Montréal) organised the Doctoral Program; Andrea Rendl (Satalia) organised a special event on diversity, equity, and inclusion; Eugene Freuder organised a CP App competition; Jason Nguyen, Peter J. Stuckey and Guido Tack (Monash University) organised the MiniZinc challenge; and Gilles Audemard, Christophe Lecoutre, and Emmanuel Lonca (Université d'Artois) organised the XCSP3 competition (both MiniZinc and XCSP3 competitions were part of the FLoC Olympic Games).

Many people have contributed to make this conference such a success, and I am grateful to all of them. First of all, I wish to thank all authors for their submission of high-quality scientific work, thus providing the material from which the conference is made. I am also very grateful to all chairs, who managed dedicated tracks and special events, to the senior Program Committee members who conducted numerous discussions with reviewers to reach consensual decisions, among other things, and to the Program Committee members who wrote 243 high-quality reviews and participated to numerous discussions. I would also like to thank the *Association for Constraint Programming* (ACP) for its trust and its very helpful organisation support, with a more specific thank to the ACP president, J. Christopher Beck (University of Toronto), and the ACP conference coordinator, Emmanuel Hebrard (LAAS-CNRS).

Finally, the conference would not have been possible without the great job done by all the people involved in the local organisation chaired by Alexandra Silva (Cornell University). I heavily relied on Roie Zivan (Ben Gurion University) and Ferdinando Fioretto (Syracuse University) for making the many necessary arrangements related to the CP 2022 program and to speedily announce program updates on the conference website.

May 2022, Lyon, France Christine Solnon

Senior Program Committee

André Ciré	University of Toronto
David Cohen	Oxford University
Sophie Demassey	CMA research lab
Ferdinando Fioretto	Syracuse University
Pierre Flener	Uppsala University
Maria Garcia de la Banda	Monash University
Ian Gent	University of St. Andrews
Emmanuel Hebrard	LAAS-CNRS
Zeynep Kiziltan	University of Bologna
Christophe Lecoutre	Université d'Artois
Andrea Lodi	Cornell Tech
Ines Lynce	University of Lisboa
Barry O'Sullivan	University College Cork
Pierre Schaus	UCLouvain
Helmut Simonis	Insight Centre for Data Analytics
Charlotte Truchet	University de Nantes
Willem Van Hoeve	Carnegie Mellon University
Roland Yap	National University of Singapore

Program Committee

Gilles Audemard	CRIL
Sebastien Bardin	CEA LIST
Chris Beck	University of Toronto
Nicolas Beldiceanu	IMT Atlantique
Jeremias Berg	University of Helsinki
David Bergman	University of Connecticut
Armin Biere	Albert-Ludwigs-University
Clément Carbonnel	CNRS
Mats Carlsson	RISE Research Institutes of Sweden
Martin Cooper	IRIT - Universite Paul Sabatier
Thi-Bich-Hanh Dao	University of Orleans
Simon de Givry	INRA - MIAT
Guillaume Derval	Liège Université
Agostino Dovier	Univ. di UDINE
Catherine Dubois	ENSIIE-Samovar
Jean-Guillaume Fages	COSLING S.A.S
Maria Andreina Francisco Rodriguez	Uppsala University
Xavier Gillard	Université Catholique de Louvain
Arnaud Gotlieb	Simula Research Laboratory
Tias Guns	Vrije Universiteit Brussel
John Hooker	Carnegie Mellon University
Marie-José Huguet	LAAS-CNRS Université de Toulouse
Alexey Ignatiev	Monash University
Mikolas Janota	Czech Technical University in Prague
Christopher Jefferson	University of St. Andrews
Serdar Kadioglu	Brown University
Roger Kameugne	University of Maroua
George Katsirelos	MIA Paris INRAE AgroParisTech
Lars Kotthoff	University of Wyoming
T. K. Satish Kumar	University of Southern California
Jean Marie Lagniez	CRIL
Nadjib Lazaar	UM2-LIRMM
Chu-Min Li	Université de Picardie Jules Verne
Arnaud Malapert	Université Côte d'Azur CNRS I3S
Felip Manyà	IIIA-CSIC
Pierre Marquis	CRIL U. Artois & CNRS
Ciaran McCreesh	University of Glasgow
Laurent Michel	University of Connecticut
Ian Miguel	University of St Andrews
Michela Milano	DISI Universita' di Bologna
Samba Ndojh Ndiaye	Liris
Peter Nightingale	University of York
Justin Pearson	Uppsala University
Laurent Perron	Google France
Gilles Pesant	Polytechnique Montréal

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Andreas Podelski	University of Freiburg
Enrico Pontelli	New Mexico State University
Cédric Pralet	ONERA Toulouse
Charles Prud'Homme	IMT Atlantique LS2N
Claude-Guy Quimper	Laval University
Louis-Martin Rousseau	Polytechnique Montréal
Michel Rueher	Université Côte d'Azur CNRS I3S
Domenico Salvagnin	University of Padova
Alexander Schiendorfer	Technische Hochschule Ingolstadt
Francesca Rossi	IBM Research
Thomas Schiex	INRAE
Andreas Schutt	CSIRO and The University of Melbourne
Mohamed Siala	INSA Toulouse & LAAS-CNRS
Laurent Simon	Labri Bordeaux Institute of Technology
Gilles Simonin	Institut Mines Telecom - Atlantique
Helge Spieker	Simula Research Laboratory
Kostas Stergiou	University of Western Macedonia
Peter J. Stuckey	Monash University
Guido Tack	Monash University
Cyril Terrioux	LIS - UMR CNRS 7020 - Aix-Marseille Université
Gilles Trombettoni	LIRMM University of Montpellier
Peter van Beek	University of Waterloo
Hélène Verhaeghe	Polytechnique Montréal
Petr Vilím	IBM Czech
Philippe Vismara	SupAgro - MISTEA / LIRMM
Neng-Fa Zhou	CUNY Brooklyn College and Graduate Center

List of Authors

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Augustin Delecluse (19)
TRAIL, ICTEAM, UCLouvain,
Louvain-la-Neuve, Belgium

Rémi Douence (6)
IMT Atlantique, LS2N, Inria, (Gallinette),
Nantes, France

Jan Dreier (20)
Algorithms and Complexity Group,
TU Wien, Austria

Alexander Ek (21)
Dept. of Data Science & AI, Monash University,
Melbourne, Australia;
CSIRO Data61, Melbourne, Australia

Guillaume Escamocher (16)
Insight Centre for Data Analytics, School of
Computer Science and Information Technology,
University College Cork, Ireland

Joan Espasa (18, 22)
School of Computer Science,
University of St Andrews, UK

Hélène Fargier (23)
IRIT, Université de Toulouse, CNRS,
Toulouse INP, UT3, France

Benjamin Fuller (17)
University of Connecticut, Storrs, CT, USA

Mohamed Gaha (34)
IREQ, Varennes, Canada

Junsong Gao (39)
College of Computer Science, Chongqing
University, China

Rebecca Gentzel (24)
University of Connecticut, Storrs, CT, USA

Xavier Gillard (14)
UCLouvain, Louvain-la-Neuve, Belgium

Ramiz Gindullin (6)
IMT Atlantique, LS2N (TASC), Nantes, France

Stephan Gocht (25)
Lund University, Sweden;
University of Copenhagen, Denmark

Priyanka Golia (36)
Indian Institute of Technology Kanpur, India;
National University of Singapore, Singapore

Tias Guns (8, 29)
Declarative Languages and Artificial Intelligence,
KU Leuven, Belgium

Djamal Habet (12)
Aix-Marseille Univ, Université de Toulon, CNRS,
LIS, France

Emmanuel Hebrard (9)
LAAS CNRS, Toulouse, France

Marijn J. H. Heule (26)
Carnegie Mellon University,
Pittsburgh, PA, USA

Amel Hidouri (27)
CRIL – CNRS UMR 8188,
University of Artois, France;
LARODEC, University of Tunis, Tunisia

Ruth Hoffmann (28)
School of Computer Science,
University of St Andrews, UK

Said Jabbour (27)
CRIL – CNRS UMR 8188,
University of Artois, France

Anthony Karahalios (26)
Carnegie Mellon University,
Pittsburgh, PA, USA

George Katsirelos (7, 37)
Université Fédérale de Toulouse, ANITI, INRAE,
MIA Paris, AgroParisTech, 75231 Paris, France

Donald E. Knuth (1)
Stanford University, CA, USA

Samuel Kolb (8, 29)
Declarative Languages and Artificial Intelligence,
KU Leuven, Belgium

Akshat Kumar (5)
Singapore Management University, Singapore

Mohit Kumar (8, 29)
Declarative Languages and Artificial Intelligence,
KU Leuven, Belgium

T. K. Satish Kumar (5)
University of Southern California,
Los Angeles, CA, USA

Daphné Lafleur (30)
Polytechnique Montréal, Canada;
Quebec Artificial Intelligence Institute (Mila),
Canada

Daniel Lafond (10)
Thales Digital Solutions, Québec, Canada

Jimmy H. M. Lee (31)
Department of Computer Science and
Engineering, The Chinese University of Hong
Kong, Shatin, China

Arnaud Lequen (13)
IRIT, University of Toulouse, France

Hongbo Li (32)
School of Information Science and Technology,
Northeast Normal University, Changchun, China

Zhanshan Li (32)
College of Computer Science and Technology,
Jilin University, Changchun, China

Xiangshuang Liu (39)
College of Computer Science,
Chongqing University, China

Jheisson López (33)
University College Cork, School of Computer
Science, Ireland;
SFI Centre for Research Training in Artificial
Intelligence, Cork, Ireland

Frédéric Maris (13)
IRIT, University of Toulouse, France

Ciaran McCreesh (25)
University of Glasgow, UK

Kuldeep S. Meel (36)
National University of Singapore, Singapore

Sebastian Meiswinkel (41)
MCP Algorithm Factory, MCP GmbH,
Wien, Austria

Jérôme Mengin (23)
IRIT, Université de Toulouse, CNRS,
Toulouse INP, UT3, France

Laurent Michel (17, 24)
University of Connecticut, Storrs, CT, USA

Ian Miguel (18, 22)
School of Computer Science,
University of St Andrews, UK

Pierre Montalbano (7)
Université Fédérale de Toulouse, ANITI,
INRAE, UR 875, 31326 Toulouse, France

Nysret Musliu (41)
Christian Doppler Laboratory for Artificial
Intelligence and Optimization for Planning and
Scheduling, DBAI, TU Wien, Austria

Miguel A. Nacenta (28)
Department of Computer Science,
University of Victoria, Canada

Franklin Nguewouo (34)
Hydro-Québec, Canada

Peter Nightingale (18, 38)
Department of Computer Science,
University of York, UK

Jakob Nordström (25)
University of Copenhagen, Denmark;
Lund University, Sweden

Sebastian Ordyniak (20)
Algorithms and Complexity Group,
University of Leeds, UK

Gilles Pesant (30)
Polytechnique Montréal, Canada

Louis Popovic (34)
Computer Engineering and Software Engineering
Department, Polytechnique Montréal, Canada

Siddharth Prasad (3)
Computer Science Department, Carnegie Mellon
University, Pittsburgh, PA, USA

Matthieu Py (12)
Aix-Marseille Univ, Université de Toulon, CNRS,
LIS, France

Luis Quesada (16)
Insight Centre for Data Analytics, School of
Computer Science and Information Technology,
University College Cork, Ireland

Claude-Guy Quimper (6, 10, 15)
Université Laval, Québec, Canada

Badran Raddaoui (27)
SAMOVAR, Télécom SudParis,
Institut Polytechnique de Paris, France

Louis-Martin Rousseau (35)
Mathematics and Industrial Engineering
Department, Polytechnique Montréal, Canada

Isaac Rudich (35)
Mathematics and Industrial Engineering
Department, Polytechnique Montréal, Canada

Tuomas Sandholm (3)
Computer Science Department, Carnegie Mellon
University, Pittsburgh, PA, USA;
Optimized Markets, Inc., Pittsburgh, PA, USA;
Strategic Machine, Inc., Pittsburgh, PA, USA;
Strategy Robot, Inc., Pittsburgh, PA, USA

Pierre Schaus (14, 19)
UCLouvain, Louvain-la-Neuve, Belgium

Nicolas Schmidt (23)
IRIT, Université de Toulouse, CNRS,
Toulouse INP, UT3, France

Andreas Schutt (21)
CSIRO Data61, Melbourne, Australia

Vanessa Simard (10)
NQB.ai, Québec, Canada

Arambam James Singh (5)
National University of Singapore, Singapore

Arunesh Sinha (5)
Singapore Management University, Singapore

Mate Soos (36)
National University of Singapore, Singapore

Peter J. Stuckey (21)
Dept. of Data Science & AI, Monash University,
Melbourne, Australia

Yan (Lindsay) Sun (17)
University of Rhode Island, Kingston, RI, USA

Stefan Szeider (20)
Algorithms and Complexity Group,
TU Wien, Austria

Guido Tack (21)
Dept. of Data Science & AI, Monash University,
Melbourne, Australia

Fulya Trösser (37)
Université Fédérale de Toulouse, ANITI,
INRAE, UR 875, 31326 Toulouse, France

Felix Ulrich-Oltean (38)
Department of Computer Science,
University of York, UK

Pascal Van Hentenryck (19)
Georgia Institute of Technology,
Atlanta, GA, USA

Willem-Jan van Hoeve (24, 26)
Carnegie Mellon University,
Pittsburgh, PA, USA

Mateu Villaret (22)
Department of Computer Science, Applied
Mathematics and Statistics, University of
Girona, Spain

Ellen Vitercik (3)
Department of Electrical Engineering and
Computer Sciences, University of California
Berkeley, CA, USA

James Alfred Walker (38)
Department of Computer Science,
University of York, UK

Daniel Walkiewicz (41)
MCP Algorithm Factory, MCP GmbH,
Wien, Austria

Jie Wang (39)
College of Computer Science,
Chongqing University, China

Ruiwei Wang (40)
School of Computing, National University of
Singapore, Singapore

Felix Winter (41)
Christian Doppler Laboratory for Artificial
Intelligence and Optimization for Planning and
Scheduling, DBAI, TU Wien, Austria

Yaling Wu (32)
School of Information Science and Technology,
Northeast Normal University, Changchun, China

Roland H. C. Yap (40)
School of Computing, National University of
Singapore, Singapore

Minghao Yin (32)
School of Information Science and Technology,
Northeast Normal University, Changchun, China

Allen Z. Zhong (31)
Department of Computer Science and
Engineering, The Chinese University of Hong
Kong, Shatin, China

Xu Zhu (28)
School of Computer Science,
University of St Andrews, UK

All Questions Answered

Donald E. Knuth 🏠
Stanford University, CA, USA

Abstract

During the past two years, the speaker has been drafting Section 7.2.2.3 of *The Art of Computer Programming*, which is intended to be a solid introduction to techniques for solving Constraint Satisfaction Problems. The CP 2022 conference is an excellent opportunity for him to get feedback from the leading experts on the subject, and so he was delighted to learn that the organizers were also interested in hearing a few words from him.

Rather than giving a canned lecture, he much prefers to let the audience choose the topics, and for all questions to be kept a secret from him until the lecture is actually in progress. (He believes that people often learn more from answers that are spontaneously fumbled than from responses that are carefully preplanned.)

Questions related to constraints will naturally be quite welcome, but questions on any subject whatsoever will not be ducked! He'll try to answer them all as best he can, without spending a great deal of time on any one topic, unless there is special interest to go into more depth.

Meanwhile he hopes to have drafted some notes for circulation before the conference begins, in case some attendees might wish to focus some of their questions on expository material related to his forthcoming book, either during this session or informally afterwards.

Warning: His least favorite questions have the form "What is your favorite X?" If you want to ask such questions, please try to do it cleverly so that he doesn't have to choose between different things that he loves in different ways.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases The Art of Computer Programming

Digital Object Identifier 10.4230/LIPIcs.CP.2022.1

Category Invited Talk

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 1; pp. 1:1–1:1
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Fixed-Template Promise Model Checking Problems

Kristina Asimi
Department of Algebra, Faculty of Mathematics and Physics,
Charles University, Prague, Czechia

Libor Barto
Department of Algebra, Faculty of Mathematics and Physics,
Charles University, Prague, Czechia

Silvia Butti
Department of Information and Communication Technologies,
Universitat Pompeu Fabra, Barcelona, Spain

Abstract

The fixed-template constraint satisfaction problem (CSP) can be seen as the problem of deciding whether a given primitive positive first-order sentence is true in a fixed structure (also called model). We study a class of problems that generalizes the CSP simultaneously in two directions: we fix a set $\mathcal{L}$ of quantifiers and Boolean connectives, and we specify two versions of each constraint, one strong and one weak. Given a sentence which only uses symbols from $\mathcal{L}$, the task is to distinguish whether the sentence is true in the strong sense, or it is false even in the weak sense.

We classify the computational complexity of these problems for the existential positive equality-free fragment of first-order logic, i.e., $\mathcal{L} = \{\exists, \wedge, \vee\}$, and we prove some upper and lower bounds for the positive equality-free fragment, $\mathcal{L} = \{\exists, \forall, \wedge, \vee\}$. The partial results are sufficient, e.g., for all extensions of the latter fragment.

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

Keywords and phrases Model Checking Problem, First-Order Logic, Promise Constraint Satisfaction Problem, Multi-Homomorphism

Digital Object Identifier 10.4230/LIPIcs.CP.2022.2

Funding Kristina Asimi and Libor Barto have received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (Grant Agreement No. 771005, CoCoSym). Silvia Butti was supported by a MICCIN grant PID2019-109137GB-C22 and by a fellowship from "la Caixa" Foundation (ID 100010434). The fellowship code is LCF/BQ/DI18/11660056. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 713673.

1 Introduction

The fixed-template finite-domain constraint satisfaction problem (CSP) is a framework for expressing many computational problems such as various versions of logical satisfiability, graph coloring, and systems of equations, see the survey [3]. A convenient formalization, that we adopt in this paper, is as follows: a *template* is a relational structure $\mathbb{A}$, and the *CSP over* $\mathbb{A}$ is the problem of deciding whether a given $\{\exists, \wedge\}$-sentence is true in $\mathbb{A}$. Here, an $\{\exists, \wedge\}$*-sentence* is a sentence of first-order logic that uses only the relation symbols of $\mathbb{A}$, the logical connective $\wedge$, and the quantifier $\exists$. To see that this formalization indeed expresses *constraint* satisfaction problems, consider, e.g., the sentence $\exists x \exists y \exists z \; R(x, y) \wedge S(y, z)$: this sentence is true in a structure $\mathbb{A}$ if the variables x, y, z can be evaluated so that both atomic formulas (constraints) are satisfied in $\mathbb{A}$.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 2; pp. 2:1–2:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Motivated by recent developments in the area, we study an extension of this framework in two simultaneous directions. One direction, discussed in Subsection 1.1, is to enable other choices of permitted quantifiers and connectives. Another direction, discussed in Subsection 1.2, is to consider two versions of each relation, strong and weak (a so-called promise problem). Our contributions are then described in Subsection 1.3.

1.1 Model checking problem parametrized by the model

The model checking problem [13] takes as input a structure $\mathbb{A}$ (often called a model) and a sentence ϕ in a specified logic and asks whether $\mathbb{A} \models \phi$, i.e., whether $\mathbb{A}$ satisfies ϕ. We study the situation where $\mathbb{A}$ is a fixed finite relational structure, so the input is simply ϕ, and the logic is a fragment of the first-order logic obtained by restricting the allowed quantifiers to a subset $\mathcal{L}$ of $\{\exists, \forall, \wedge, \vee, =, \neq, \neg\}$. Thus, for each $\mathbb{A}$ and each of the 2^7 choices for $\mathcal{L}$, we obtain a computational problem, which we call the *$\mathcal{L}$-Model Checking Problem over* $\mathbb{A}$ and denote $\mathcal{L}$-MC($\mathbb{A}$).

The computational complexity classification of $\{\exists, \wedge\}$-MC($\mathbb{A}$), i.e., CSP over $\mathbb{A}$, has been a very active research program in the last 20 years, which culminated in the celebrated dichotomy theorem obtained independently in [6] and [18]: each CSP over $\mathbb{A}$ is in P (solvable in polynomial time) or is NP-complete. For the case $\mathcal{L} = \{\exists, \forall, \wedge\}$, $\mathcal{L}$-MC($\mathbb{A}$) is the so called *quantified CSP*, another well-studied class of problems, see the survey [16]. It was widely believed that this class exhibits a P/NP-complete/PSPACE-complete trichotomy [8]. A recent breakthrough [19] shows that at least three more complexity classes appear within quantified CSPs, and ongoing work suggests that even 6 is not the final number. In any case, the full complexity classification of $\{\exists, \forall, \wedge\}$-MC($\mathbb{A}$) is a challenging open problem.

The remaining $2^7 - 2$ choices for $\mathcal{L}$ do not need to be considered separately. For instance, $\{\exists, \wedge, =\}$-MC($\mathbb{A}$) is no harder than $\{\exists, \wedge\}$-MC($\mathbb{A}$) because equalities can be propagated out in this case, and $\{\forall, \vee\}$-MC($\mathbb{A}$) is dual to $\{\exists, \wedge\}$-MC($\mathbb{A}$) so we get a P/coNP-complete dichotomy for free, etc. Moreover, some choices of $\mathcal{L}$, such as $\mathcal{L} = \{\exists, \vee\}$, lead to very simple problems. It turns out [14] (see Subsection 3.3) that, in addition to $\mathcal{L} = \{\exists, \wedge\}$ and $\mathcal{L} = \{\exists, \forall, \wedge\}$, only two more fragments need to be considered in order to fully understand the complexity of $\mathcal{L}$-MC($\mathbb{A}$), namely $\mathcal{L} = \{\exists, \wedge, \vee\}$ and $\mathcal{L} = \{\exists, \forall, \wedge, \vee\}$.

The former fragment was addressed in [14]: except for a simple case solvable in polynomial time (in fact, L, the logarithmic space), all the remaining problems are NP-complete. The latter fragment turned out to be more challenging but, after a series of partial results [14, 11, 17] (see also [15, 7]), the full complexity classification was given in [12, 13]: each problem in this class is in P (even L), or is NP-complete, coNP-complete, or PSPACE-complete. These results are summarized in Table 1.

Table 1 Known complexity results for $\mathcal{L}$-MC($\mathbb{A}$).

$\mathcal{L}$-MC($\mathbb{A}$)	Complexity
$\{\exists, \wedge\}$-MC($\mathbb{A}$) (CSP)	dichotomy: P or NP-complete
$\{\exists, \forall, \wedge\}$-MC($\mathbb{A}$) (QCSP)	≥ 6 classes
$\{\exists, \wedge, \vee\}$-MC($\mathbb{A}$)	dichotomy: L or NP-complete
$\{\forall, \exists, \wedge, \vee\}$-MC($\mathbb{A}$)	tetrachotomy: L, NP-complete, coNP-complete, PSPACE-complete

1.2 Promise model checking problem

The Promise CSP is a recently introduced extension of the CSP framework motivated by open problems in (in)approximability of satisfiability and coloring problems [1, 5, 2]. The template consists of two structures $\mathbb{A}$ and $\mathbb{B}$ of the same signature, where $\mathbb{A}$ specifies a strong form of each relation and $\mathbb{B}$ its weak form. The Promise CSP over $(\mathbb{A}, \mathbb{B})$ is then the problem of distinguishing $\{\exists, \wedge\}$-sentences that are true in $\mathbb{A}$ from those that are not true in $\mathbb{B}$.

For example, by choosing an appropriate template, we obtain the problem of distinguishing k-colorable graphs from those that are not even l-colorable (where $k \leq l$ are fixed), a problem whose complexity is notoriously open.

The generalization of Promise CSP over $(\mathbb{A}, \mathbb{B})$ to an arbitrary choice $\mathcal{L} \subseteq \{\exists, \forall, \wedge, \vee, =, \neq, \neg\}$ is referred to as the *$\mathcal{L}$-Promise Model Checking Problem over* $(\mathbb{A}, \mathbb{B})$ and is denoted $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$. Similarly as in the special case $\mathbb{A} = \mathbb{B}$, which is exactly $\mathcal{L}$-MC$(\mathbb{A})$, it is sufficient to consider only four fragments. A full complexity classification for $\{\exists, \wedge\}$-PMC (i.e., Promise CSP) is much desired but widely open, and $\{\exists, \forall, \wedge\}$-PMC is likely even harder. This work concentrates on the remaining two classes of problems, $\{\exists, \wedge, \vee\}$-PMC and $\{\exists, \forall, \wedge, \vee\}$-PMC.

Our motivation was that these cases might be substantially simpler, as indicated by the non-promise special case, and at the same time, the investigation could uncover interesting intermediate problems towards the grand endeavor of understanding the sources of tractability and hardness in computation. We believe that our findings confirm this hope.

▶ **Example 1.** Consider structures $\mathbb{A}$ and $\mathbb{B}$ with a single relation symbol $=$ interpreted as the equality on a three-element domain in $\mathbb{A}$ and as the equality on a two-element domain in $\mathbb{B}$. For $\mathcal{L} = \{\exists, \forall, \wedge, \vee\}$, both $\mathcal{L}$-MC$(\mathbb{A})$ and $\mathcal{L}$-MC$(\mathbb{B})$ are PSPACE-complete problems, see [14].

It is not hard to see that every $\mathcal{L}$-sentence true in $\mathbb{A}$ is also true in $\mathbb{B}$. In this sense, the relation in $\mathbb{A}$ is stronger than the relation in $\mathbb{B}$. On the other hand, there are $\mathcal{L}$-sentences true in $\mathbb{B}$ that are not true in $\mathbb{A}$, e.g., $\phi = \forall x \exists y \forall z \ (z = x) \vee (z = y)$. Therefore, $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$ could potentially be easier than the above non-promise problems – instances such as ϕ need not be considered (there is no requirement on the algorithm for such inputs). Nevertheless, the problem remains PSPACE-complete, as shown in Proposition 18.

1.3 Contributions

Theorem 7 and Theorem 13 provide basics for an algebraic approach to $\{\exists, \wedge, \vee\}$-PMC and $\{\exists, \forall, \wedge, \vee\}$-PMC by characterizing definability in terms of compatible functions: multi-homomorphisms for the $\{\exists, \wedge, \vee\}$ fragment and surjective multi-homomorphisms (*smuhoms*) for $\{\exists, \forall, \wedge, \vee\}$. The proofs can be obtained as relatively straightforward generalizations of the proofs for MC in [13]; however, we believe that our approach is somewhat more transparent. In particular, it allows us to easily characterize meaningful templates for these problems (Propositions 6 and 12).

For $\{\exists, \wedge, \vee\}$-PMC, we obtain an L/NP-complete dichotomy in Theorem 9. It turns out that, apart from some simple cases, the problem is NP-complete. Interestingly, there is a "single reason" for hardness: the NP-hardness of coloring a rainbow colorable hypergraph from [9].

For $\{\exists, \forall, \wedge, \vee\}$-PMC, our complexity results are only partial, leaving two gaps for further investigation. The results are sufficient for full complexity classification of $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$ in the case that $\mathcal{L} = \{\exists, \forall, \wedge, \vee\}$ and one of the structures $\mathbb{A}$, $\mathbb{B}$ has a two-element domain, and also in the case that $\mathcal{L} \supsetneq \{\exists, \forall, \wedge, \vee\}$. We also give some examples where our efforts have failed so far. One such example is a particularly interesting $\{\exists, \forall, \wedge, \vee\}$-PMC over 3-element domains: given a $\{\exists, \forall, \wedge, \vee\}$-sentence ϕ whose atomic formulas are all of the form $R^i(x)$, $i \in \{1, 2, 3\}$, distinguish between the case where ϕ is true when $R^i(x)$ is interpreted as "$x = i$", and the case where ϕ is false when $R^i(x)$ is interpreted as "$x \neq i$".

Our complexity results are summarized in Table 2, the conditions for $\mathcal{L} = \{\exists, \forall, \wedge, \vee\}$ are stated in terms of special surjective multi-homomorphisms of the template, introduced in Subsection 5.3.

Table 2 Complexity results for $\mathcal{L}$-PMC($\mathbb{A}, \mathbb{B}$).

$\mathcal{L}$-PMC($\mathbb{A}, \mathbb{B}$)	**Condition**	**Complexity**
$\{\exists, \forall, \wedge\}$-PMC($\mathbb{A}, \mathbb{B}$)		L/NP-complete
$\{\exists, \forall, \wedge, \vee\}$-PMC($\mathbb{A}, \mathbb{B}$)	AE-smuhom, or A-smuhom and E-smuhom and $\mathbb{A}, \mathbb{B}$ digraphs	L
	A-smuhom and E-smuhom	NP $\cap$ coNP
	A-smuhom, no E-smuhom	NP-complete
	E-smuhom, no A-smuhom	coNP-complete
	no A-smuhom and no E-smuhom	NP-hard and coNP-hard
$\{\exists, \forall, \wedge, \vee, =\}$-PMC($\mathbb{A}, \mathbb{B}$), $\{\exists, \forall, \wedge, \vee, \neq\}$-PMC($\mathbb{A}, \mathbb{B}$), $\{\exists, \forall, \wedge, \vee, \neg\}$-PMC($\mathbb{A}, \mathbb{B}$)		L/PSPACE-complete

2 Preliminaries

Structures. We use a standard model-theoretic terminology, but restrict the generality of some concepts for the purposes of this paper. A *relation* of arity $n \geq 1$ on a set A is a set of n-tuples of elements of A, i.e., a subset of A^n. The *complement* of a relation S is denoted $\overline{S} := A^n \setminus S$. The equality relation on A is denoted $=_A$ and the disequality relation $\neq_A$. Components of a tuple $\mathbf{a}$ are referred to as a_1, a_2, ..., i.e., $\mathbf{a} = (a_1, \ldots, a_n)$.

A *signature* is a nonempty collection of relation symbols each with an associated arity, denoted $\mathrm{ar}(R)$ for a relation symbol R. A *relational structure* (also called a *model*) $\mathbb{A}$ in the signature σ, or simply a *structure*, consists of a finite set A of size at least two, called the *universe* of $\mathbb{A}$, and a nonempty proper relation $\emptyset \subsetneq R^{\mathbb{A}} \subsetneq A^{\mathrm{ar}(R)}$ for each symbol R in σ, called the *interpretation* of R in $\mathbb{A}$. Two structures are called *similar* if they are in the same signature. The *complement* of a relational structure $\mathbb{A}$ is obtained by taking complements of all relations in the structure and is denoted $\overline{\mathbb{A}}$. A structure over a signature containing a single binary relation symbol is called a *digraph*.

We emphasize that the universe of a structure is denoted by the same letter as the structure, that the universe of every structure in this paper is assumed to be finite and at least two-element, and that each relation in a structure is assumed to be at least unary, nonempty and proper. These nonstandard requirements are placed for technical convenience and do not significantly decrease the generality of our results.

Given two similar structures $\mathbb{A}$ and $\mathbb{B}$, a function f from A to B is called a *homomorphism* from $\mathbb{A}$ to $\mathbb{B}$ if $f(\mathbf{a}) \in R^{\mathbb{B}}$ for any $\mathbf{a} \in R^{\mathbb{A}}$, where $f(\mathbf{a})$ is computed component-wise. We only work with total functions, that is, $f(a)$ is defined for every $a \in A$.

Multi-homomorphisms. A *multi-valued function* f from A to B is a mapping from A to $\mathcal{P}_{\neq\emptyset}B$, the set of all nonempty subsets of B. It is called *surjective* if for every $b \in B$, there exists $a \in A$ such that $b \in f(a)$. The *inverse* of a surjective multi-valued function f from A to B is the multi-valued function from B to A defined by $f^{-1}(b) = \{a : b \in f(a)\}$. For a tuple $\mathbf{a} \in A^n$ we write $f(\mathbf{a})$ for $f(a_1) \times \cdots \times f(a_n)$. The value $\max\{|f(a)| : a \in A\}$ is referred to as the *multiplicity* of f; in particular, multi-valued functions of multiplicity one are essentially functions. For two multi-valued functions f and f' from A to B, we say that f' is *contained in* f if $f'(a) \subseteq f(a)$ for each $a \in A$.

Given two similar structures $\mathbb{A}$ and $\mathbb{B}$, a multi-valued function f from A to B is called a *multi-homomorphism* [1] from $\mathbb{A}$ to $\mathbb{B}$ if for any R in the signature and any $\mathbf{a} \in R^{\mathbb{A}}$, we have $f(\mathbf{a}) \subseteq R^{\mathbb{B}}$, i.e., $\mathbf{b} \in R^{\mathbb{B}}$ whenever $b_i \in f(a_i)$ for each $i \in [\mathrm{ar}(R)] = \{1, 2, \ldots, \mathrm{ar}(R)\}$. Notice that if f is a multi-homomorphism from $\mathbb{A}$ to $\mathbb{B}$, then so is any multi-valued function contained in f. In particular, if f is a multi-homomorphism from $\mathbb{A}$ to $\mathbb{B}$, then any function $g : A \to B$ with $g(a) \in f(a)$ for each $a \in A$ is a homomorphism from $\mathbb{A}$ to $\mathbb{B}$. The converse does not hold in general, as witnessed by structures $\mathbb{A} = \mathbb{B}$ with a single binary equality relation and any multi-valued function of multiplicity greater than one.

The set of all multi-homomorphisms from $\mathbb{A}$ to $\mathbb{B}$ is denoted by $\mathrm{MuHom}(\mathbb{A}, \mathbb{B})$ and the set of all surjective multi-homomorphisms by $\mathrm{SMuHom}(\mathbb{A}, \mathbb{B})$.

Fragments of first-order logic. Let $\mathcal{L} \subseteq \{\exists, \forall, \wedge, \vee, =, \neq, \neg\}$ and fix some signature. By an *$\mathcal{L}$-sentence* (resp., *$\mathcal{L}$-formula*) we mean a sentence (resp., formula) of first-order logic that only uses variables (denoted x_i, y_i, z_i), relation symbols in the signature, and connectives and quantifiers in $\mathcal{L}$. We refer to this fragment of first-order logic as the *$\mathcal{L}$-logic*.

The *prenex normal form* of an $\mathcal{L}$-formula is an equivalent formula that begins with quantified variables followed by a quantifier-free formula. The prenex normal form can be computed in logarithmic space and it is an $\mathcal{L}$-formula whenever $\mathcal{L}$ does not contain the negation.

For a structure $\mathbb{A}$ in the signature and an $\mathcal{L}$-sentence ϕ, we write $\mathbb{A} \models \phi$ if ϕ is satisfied in $\mathbb{A}$. More generally, given an $\mathcal{L}$-formula ψ, a tuple of distinct variables $(v_1, \ldots, v_n)$ which contains every free variable of ψ and a tuple $(a_1, \ldots, a_n) \in A^n$, we write $\mathbb{A} \models \psi(a_1, \ldots, a_n)$ if ψ is satisfied when $v_1, \ldots, v_n$ are evaluated as $\varepsilon_A(v_1) = a_1, \ldots, \varepsilon_A(v_n) = a_n$, respectively. Notice that variables $v_1, \ldots, v_n$ indeed need to be pairwise distinct, otherwise this notation would not make sense. The tuple $(v_1, \ldots, v_n)$ is often specified by writing $\psi = \psi(v_1, \ldots, v_n)$.

We say that a relation $S \subseteq A^n$ is *$\mathcal{L}$-definable* from $\mathbb{A}$ if there exists an $\mathcal{L}$-formula $\psi(v_1, \ldots, v_n)$ such that, for all $(a_1, \ldots, a_n) \in A^n$, we have $(a_1, \ldots, a_n) \in S$ if and only if $\mathbb{A} \models \psi(a_1, \ldots, a_n)$. In this case, we also say that $\psi(v_1, \ldots, v_n)$ defines S in $\mathbb{A}$.

3 Promise model checking

In this section we define the promise model checking problem restricted to $\mathcal{L} \subseteq \{\exists, \forall, \wedge, \vee, =, \neq, \neg\}$. We start by briefly discussing the non-promise setting.

3.1 Model checking problem

Let $\mathcal{L} \subseteq \{\exists, \forall, \wedge, \vee, =, \neq, \neg\}$ and $\mathbb{A}$ be a structure in a signature σ. Recall that the *$\mathcal{L}$-Model Checking Problem over* $\mathbb{A}$, denoted $\mathcal{L}$-MC($\mathbb{A}$), is the problem of deciding whether a given $\mathcal{L}$-sentence ϕ (in the same signature as $\mathbb{A}$) is true in $\mathbb{A}$.

A simple but important observation sometimes allows us to compare the complexity of the $\mathcal{L}$-MC problems over two templates $\mathbb{A}$ and $\mathbb{C}$ with the same universe $A = C$ but possibly different signatures: If every relation in $\mathbb{C}$ is $\mathcal{L}$-definable from $\mathbb{A}$, then $\mathcal{L}$-MC($\mathbb{C}$) can be reduced in polynomial-time (even logarithmic space) to $\mathcal{L}$-MC($\mathbb{A}$). Indeed, the reduction amounts to replacing atomic formulas of the form $R(\mathbf{v})$ by their definitions.

[1] We deviate here from the terminology of [12, 11] because it would not work well in the promise setting.

The starting point of the algebraic approach to $\mathcal{L}$-MC is to find a characterization of definability in terms of certain "compatible functions" or "symmetries" (so called polymorphisms for $\mathcal{L} = \{\exists, \wedge, =\}$ [3], surjective polymorphisms for $\mathcal{L} = \{\exists, \forall, \wedge, =\}$ [16], multi-endomorphisms for $\mathcal{L} = \{\exists, \wedge, \vee\}$, surjective multi-endomorphisms for $\mathcal{L} = \{\exists, \forall, \wedge, \vee\}$ [13]; see also [4]). Because such characterizations are central in this paper as well, we now explain the basic idea for a simple case.

For $\mathcal{L} = \{\exists, \wedge, \vee, =\}$, the appropriate type of compatible function is endomorphism: a nonempty relation $S \subseteq A^n$ is $\mathcal{L}$-definable from $\mathbb{A}$ if and only if it is invariant under every endomorphism of $\mathbb{A}$ (i.e., a homomorphism from $\mathbb{A}$ to itself). The forward direction is well-known and easy to verify. For the backward direction, assume $A = [k] := \{1, \ldots, k\}$ and consider the following formula.

$$\phi(x_1, \ldots, x_k) := \bigwedge_{R \in \sigma} \bigwedge_{\mathbf{r} \in R^{\mathbb{A}}} R(x_{r_1}, \ldots, x_{r_{\mathrm{ar}(R)}}) \tag{1}$$

It follows immediately from definitions that, for any structure $\mathbb{E}$ in the signature of $\mathbb{A}$, $\mathbb{E} \models \phi(e_1, \ldots, e_k)$ if and only if the mapping defined by $i \mapsto e_i$ for each $i \in [k]$ is a homomorphism from $\mathbb{A}$ to $\mathbb{E}$. This in particular holds for $\mathbb{E} = \mathbb{A}$. By existential quantification we can then obtain an $\mathcal{L}$-formula defining the closure of any tuple $\mathbf{a} \in A^n$ with distinct entries under endomorphisms of $\mathbb{A}$; e.g., $\psi(x_1, x_3, x_2) := (\exists x_4)(\exists x_5) \ldots (\exists x_k)\phi$ defines the closure of $(1, 3, 2)$ under endomorphisms. Using $=$ we can also define closures of the remaining tuples with repeated entries. Finally, S is the union of closures of its members (since it is closed under endomorphisms of $\mathbb{A}$), so S can be defined by a disjunction of formulas that we have already found (after appropriately renaming variables).

Notice that this construction would not work without the equality in $\mathcal{L}$ because of tuples with repeated entries. This is the reason why we need to work with multi-valued functions for the equality-free logics that we deal with in this paper.

3.2 Promise model checking problem

Let $\mathcal{L} \subseteq \{\exists, \forall, \wedge, \vee, =, \neq, \neg\}$. The $\mathcal{L}$-Promise Model Checking Problem over a pair of similar structures $(\mathbb{A}, \mathbb{B})$ is the problem of distinguishing $\mathcal{L}$-sentences ϕ that are true in $\mathbb{A}$ from those that are not true in $\mathbb{B}$. This problem makes sense only if every $\mathcal{L}$-sentence that is true in $\mathbb{A}$ is also true in $\mathbb{B}$; we call such pairs $\mathcal{L}$-PMC templates.

► **Definition 2.** *A pair of similar structures* $(\mathbb{A}, \mathbb{B})$ *is called an* $\mathcal{L}$-PMC template *if* $\mathbb{A} \models \phi$ *implies* $\mathbb{B} \models \phi$ *for every* $\mathcal{L}$*-sentence* ϕ *in the signature of* $\mathbb{A}$ *and* $\mathbb{B}$*.*

Given an $\mathcal{L}$-PMC *template* $(\mathbb{A}, \mathbb{B})$*, the* $\mathcal{L}$-Promise Model Checking Problem over $(\mathbb{A}, \mathbb{B})$*, denoted* $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$*, is the following problem.*

Input: an $\mathcal{L}$*-sentence* ϕ *in the signature of* $\mathbb{A}$ *and* $\mathbb{B}$*;*

Output: `Yes` *if* $\mathbb{A} \models \phi$*;* `No` *if* $\mathbb{B} \not\models \phi$*.*

The definition of a template guarantees that the sets of `Yes`-instances and `No`-instances are disjoint. However, their union need not be the whole set of $\mathcal{L}$-sentences; an algorithm for $\mathcal{L}$-PMC is only required to produce correct outputs for `Yes`-instances and `No`-instances. Alternatively, we are *promised* that the input sentence is a `Yes`-instance or a `No`-instance. The complexity-theoretic notions (such as membership in NP, NP-completeness, reductions) can be adjusted naturally for the promise setting. We write $\mathcal{L}$-PMC$(\mathbb{C}, \mathbb{D}) \leq \mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$ if the former problem can be reduced to the latter problem by a logarithmic space reduction, that is, a logarithmic space transformation that maps each `Yes`-instance ϕ of $\mathcal{L}$-PMC$(\mathbb{C}, \mathbb{D})$ to a `Yes`-instance ψ of $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$ (equivalently, $\mathbb{C} \models \phi$ must imply $\mathbb{A} \models \psi$) and `No`-instances to `No`-instances (equivalently, $\mathbb{B} \models \psi$ must imply $\mathbb{D} \models \phi$).

An appropriate adjustment of definability for the promise setting is as follows. Note that we do not allow the negation in $\mathcal{L}$, otherwise the concept would need to be defined differently because of the inclusions in the definition.

▶ **Definition 3.** *Assume $\neg \notin \mathcal{L}$ and let $(\mathbb{A}, \mathbb{B})$ be a pair of similar structures. We say that a pair of relations (S, T), where $S \subseteq A^n$ and $T \subseteq B^n$, is* promise-$\mathcal{L}$-definable *(or* p-$\mathcal{L}$-definable*) from $(\mathbb{A}, \mathbb{B})$ if there exist relations S' and T' and an $\mathcal{L}$-formula $\psi(v_1, \ldots, v_n)$ such that $S \subseteq S'$, $T' \subseteq T$, $\psi(v_1, \ldots, v_n)$ defines S' in $\mathbb{A}$, and $\psi(v_1, \ldots, v_n)$ defines T' in $\mathbb{B}$.*

*We say that an $\mathcal{L}$-*PMC *template $(\mathbb{C}, \mathbb{D})$ is p-$\mathcal{L}$-definable from $(\mathbb{A}, \mathbb{B})$ (the signatures can differ) if $(Q^{\mathbb{C}}, Q^{\mathbb{D}})$ is p-$\mathcal{L}$-definable from $(\mathbb{A}, \mathbb{B})$ for each relation symbol Q in the signature of $\mathbb{C}$ and $\mathbb{D}$.*

▶ **Theorem 4.** *Assume $\neg \notin \mathcal{L}$. If $(\mathbb{A}, \mathbb{B})$ and $(\mathbb{C}, \mathbb{D})$ are $\mathcal{L}$-*PMC *templates such that $(\mathbb{C}, \mathbb{D})$ is p-$\mathcal{L}$-definable from $(\mathbb{A}, \mathbb{B})$, then $\mathcal{L}$-*PMC$(\mathbb{C}, \mathbb{D}) \leq \mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$.

Proof. The reduction is to replace each atomic $Q(\mathbf{v})$ by the corresponding formula ψ from Definition 3. For correctness of this reduction, observe that an $\mathcal{L}$-sentence which is true in a structure $\mathbb{E}$ remains true when we add tuples to the relations of $\mathbb{E}$ (since $\mathcal{L}$ does not contain $\neg$). ◀

3.3 Interesting fragments

We now explain why only four fragments of first-order logic need to be considered in order to fully understand the problems $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$. Observe first that if $\mathcal{L}$ does not contain any connective $(\wedge, \vee)$, or $\mathcal{L}$ does not contain any quantifier $(\exists, \forall)$, or $\mathcal{L} \subseteq \{\exists, \vee\}$, then each $\mathcal{L}$-PMC is in L, the logarithmic space. (In some of these cases we do not even have any valid inputs in our definition of structures.)

Secondly, notice that $(\mathcal{L} \cup \{=\})$-PMC$(\mathbb{A}, \mathbb{B})$ is essentially the same as $\mathcal{L}$-PMC$(\mathbb{A}', \mathbb{B}')$, where $\mathbb{A}'$ and $\mathbb{B}'$ are obtained from the original structures by adding a fresh binary symbol Q to the signature and setting $Q^{\mathbb{A}'}$ to $=_A$ and $Q^{\mathbb{B}'}$ to $=_B$. The disequality is dealt with analogously, thus we can and shall restrict to fragments with $\mathcal{L} \subseteq \{\exists, \forall, \wedge, \vee, \neg\}$.

Next, we deal with the negation. If $\neg$ is in $\mathcal{L}$, and $\mathcal{L}$ contains a quantifier and a connective, then it is enough to consider the case $\mathcal{L} = \{\exists, \forall, \wedge, \vee, \neg\}$ since the remaining quantifier and connective can be expressed using negation. Moreover, the complements of relations can also be expressed, so we may assume that each template $(\mathbb{A}, \mathbb{B})$ is *closed under complementation*, meaning that for every symbol R in the signature, we have a symbol $\overline{R}$ interpreted as $\overline{R}^{\mathbb{A}} = \overline{R^{\mathbb{A}}}$, $\overline{R}^{\mathbb{B}} = \overline{R^{\mathbb{B}}}$. But then $\neg$ is no longer necessary since we can propagate the negations inwards in an input sentence. We are down to $\mathcal{L} \subseteq \{\exists, \forall, \wedge, \vee\}$.

Finally, note that $\mathbb{E} \models \neg\phi$, where ϕ is an $\mathcal{L}$-sentence, is equivalent to $\overline{\mathbb{E}} \models \phi'$ where ϕ' is an $\mathcal{L}'$-sentence and $\mathcal{L}'$ is obtained from $\mathcal{L}$ by swapping $\forall \leftrightarrow \exists$ and $\vee \leftrightarrow \wedge$ (ϕ' can be, again, computed from $\neg\phi$ by inward propagation). It follows that $\phi \mapsto \phi'$ transforms every `Yes`-instance (resp., `No`-instance) of $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$ to a `No`-instance (resp., `Yes`-instance) of $\mathcal{L}'$-PMC$(\overline{\mathbb{B}}, \overline{\mathbb{A}})$, and a similar "dual" reduction works in the opposite direction. Therefore, the latter PMC has the "dual" complexity to the former PMC, e.g., if the former is NP-complete, then the latter is coNP-complete; and if the former is PSPACE-complete, then the latter is PSPACE-complete as well. We will refer to this reasoning as the *duality argument.*

Eliminating one of the logic fragments from each of the "dual" pairs, we are left with only four fragments: $\mathcal{L} = \{\exists, \wedge\}$ (whose $\mathcal{L}$-PMC is Promise CSP), $\mathcal{L} = \{\exists, \forall, \wedge\}$ (Promise Quantified CSP), $\mathcal{L} = \{\exists, \wedge, \vee\}$, and $\mathcal{L} = \{\exists, \forall, \wedge, \vee\}$. We investigate the last two separately in the next two sections.

4 Existential positive fragment

This section concerns the existential positive equality-free logic, that is, the $\mathcal{L}$-logic with $\mathcal{L} = \{\exists, \wedge, \vee\}$. We fix this $\mathcal{L}$ for the entire section.

4.1 Characterization of templates and p-$\mathcal{L}$-definability

We start by characterizing $\mathcal{L}$-PMC templates. One direction of the characterization follows from the discussion below (1), the other one from the following observation.

▶ **Lemma 5.** *Let f be a multi-homomorphism from $\mathbb{A}$ to $\mathbb{B}$, let $\phi(x_1, \dots, x_n)$ be a quantifier-free $\mathcal{L}$-formula in the same signature, and let $\mathbf{a} \in A^n$, $\mathbf{b} \in B^n$. If $\mathbb{A} \models \phi(\mathbf{a})$ and $\mathbf{b} \in f(\mathbf{a})$, then $\mathbb{B} \models \phi(\mathbf{b})$.*

Proof. The claim holds for atomic formulas by definition of multi-homomorphisms. The proof is then finished by induction on the complexity of ϕ; both $\vee$ and $\wedge$ are dealt with in a straightforward way. ◀

▶ **Proposition 6.** *A pair $(\mathbb{A}, \mathbb{B})$ of similar structures is an $\mathcal{L}$-PMC template if and only if there exists a homomorphism from $\mathbb{A}$ to $\mathbb{B}$.*

Proof. Suppose that there exists a homomorphism from $\mathbb{A}$ to $\mathbb{B}$ and $\mathbb{A} \models \phi$, where $\phi = \exists x_1 \exists x_2 \dots \exists x_n \phi'(x_1, \dots, x_n)$ is in prenex normal form. Then we have $\mathbb{A} \models \phi'(\mathbf{a})$ for some $\mathbf{a} \in A^n$, therefore $\mathbb{B} \models \phi'(f(\mathbf{a}))$ by Lemma 5, and it follows that $\mathbb{B} \models \phi$.

For the forward implication, observe that the sentence obtained from the formula (1) by existentially quantifying all the variables is true in $\mathbb{A}$ (as there exists a homomorphism from $\mathbb{A}$ to $\mathbb{A}$ – the identity), so it must be true in $\mathbb{B}$, giving us a homomorphism from $\mathbb{A}$ to $\mathbb{B}$. ◀

Note that this characterization would remain the same if we add $=$ to $\mathcal{L}$ (and/or remove $\vee$). For the following characterization of promise definability, the absence of the equality relation does make a difference, which is why we need to use multi-homomorphisms instead of homomorphisms.

▶ **Theorem 7.** *Let $(\mathbb{A}, \mathbb{B})$ and $(\mathbb{C}, \mathbb{D})$ be $\mathcal{L}$-PMC templates such that $A = C$ and $B = D$. Then $(\mathbb{C}, \mathbb{D})$ is p-$\mathcal{L}$-definable from $(\mathbb{A}, \mathbb{B})$ if and only if $\mathrm{MuHom}(\mathbb{A}, \mathbb{B}) \subseteq \mathrm{MuHom}(\mathbb{C}, \mathbb{D})$. Moreover, in such a case, $\mathcal{L}$-PMC$(\mathbb{C}, \mathbb{D}) \leq \mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$.*

Proof. It is enough to verify the equivalence, since then the second claim follows from Theorem 4. To prove the forward implication, assume that $(\mathbb{C}, \mathbb{D})$ is p-$\mathcal{L}$-definable from $(\mathbb{A}, \mathbb{B})$, let $f \in \mathrm{MuHom}(\mathbb{A}, \mathbb{B})$, and let Q be a symbol in the signature of $\mathbb{C}$ and $\mathbb{D}$. To show that $f(\mathbf{a}) \subseteq Q^{\mathbb{D}}$ for any $\mathbf{a} \in Q^{\mathbb{C}}$ we apply Lemma 5 as follows. We have $\mathbb{A} \models \psi(\mathbf{a})$, where $\psi(\mathbf{x}) = \exists y_1 \exists y_2 \dots \exists y_m \psi'(\mathbf{x}, \mathbf{y})$ is a formula from Definition 3, turned into prenex normal form. Then $\mathbb{A} \models \psi'(\mathbf{a}, \mathbf{a}')$ for some $\mathbf{a}' \in A^m$, thus $\mathbb{B} \models \psi'(\mathbf{b}, \mathbf{b}')$ for any $\mathbf{b} \in f(\mathbf{a})$ and $\mathbf{b}' \in f(\mathbf{a}')$ by Lemma 5. Therefore, $\mathbb{B} \models \psi(\mathbf{b})$ and, finally, $\mathbf{b} \in Q^{\mathbb{D}}$, as required.

For the backward implication, assume that $\mathrm{MuHom}(\mathbb{A}, \mathbb{B}) \subseteq \mathrm{MuHom}(\mathbb{C}, \mathbb{D})$, denote σ the signature of $\mathbb{A}$ and $\mathbb{B}$, and consider an n-ary relational symbol Q in the signature of $\mathbb{C}$ and $\mathbb{D}$. To prove the claim, we need to find a formula $\psi(x_1, \dots, x_n)$ that defines, in $\mathbb{A}$, a relation containing $Q^{\mathbb{C}}$ and, in $\mathbb{B}$, a relation contained in $Q^{\mathbb{D}}$.

For simplicity, assume $A = [k]$ and consider the formula

$$\phi(x_{1,1}, \dots, x_{1,n}, x_{2,1}, \dots, x_{2,n}, \dots, x_{k,n}) := \bigwedge_{R \in \sigma} \bigwedge_{\mathbf{r} \in R^{\mathbb{A}}} \bigwedge_{\mathbf{j} \in [n]^{\mathrm{ar}(R)}} R(x_{r_1, j_1}, \dots, x_{r_{\mathrm{ar}(R)}, j_{\mathrm{ar}(R)}}) \quad (2)$$

It follows immediately from definitions that, for any structure $\mathbb{E}$ in the signature σ, we have $\mathbb{E} \models \phi(e_{1,1}, \ldots, e_{k,n})$ if and only if the mapping $i \mapsto \{e_{i,1}, \ldots, e_{i,n}\}$, $1 \leq i \leq k$ is a multi-homomorphism from $\mathbb{A}$ to $\mathbb{E}$. Therefore, for any $\mathbf{a} \in A^n$, the formula $\tau_{\mathbf{a}}(x_1, \ldots, x_n)$, obtained from ϕ by renaming $x_{a_i,i}$ to x_i and existentially quantifying the remaining variables, defines in $\mathbb{E}$ the union of $f(\mathbf{a})$ over $f \in \mathrm{MuHom}(\mathbb{A}, \mathbb{E})$ of multiplicity at most n. This relation is clearly equal to the union of $f(\mathbf{a})$ over all $f \in \mathrm{MuHom}(\mathbb{A}, \mathbb{E})$. The sought after formula ψ is then the disjunction of $\tau_{\mathbf{a}}$ over all $\mathbf{a} \in Q^{\mathbb{C}}$: it defines in $\mathbb{A}$ a relation containing $Q^{\mathbb{C}}$ (because of the identity "multi"-homomorphism $\mathbb{A} \to \mathbb{A}$) and, in $\mathbb{B}$, a relation contained in $Q^{\mathbb{D}}$ (because every multi-homomorphism from $\mathbb{A}$ to $\mathbb{B}$ is a multi-homomorphism from $\mathbb{C}$ to $\mathbb{D}$, whence $f(\mathbf{a}) \subseteq Q^{\mathbb{D}}$ for any $\mathbf{a} \in Q^{\mathbb{C}}$ and any $f \in \mathrm{MuHom}(\mathbb{A}, \mathbb{B})$). ◀

4.2 Complexity classification

Since $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$ reduces to $\mathcal{L}$-MC$(\mathbb{A})$ (or $\mathcal{L}$-MC$(\mathbb{B})$) by the trivial reduction which does not change the input, and the latter problem is clearly in NP, then the former problem is in NP as well. Theorem 9 shows that $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$ is NP-hard in all the "nontrivial" cases, as in the non-promise setting. However, our proof of hardness requires (in addition to Theorem 7) a much more involved hardness result than in the non-promise case: NP-hardness of c-coloring rainbow k-colorable $2k$-uniform hypergraphs from [9] (here $c, k \geq 2$).

To state the result in our formalism, we introduce the n-ary "rainbow coloring" and "not-all-equal" relations on a set D as follows.

$$\mathrm{Rb}_D^n = \{\mathbf{d} \in D^n : \{d_1, d_2, \ldots, d_n\} = D\}, \quad \mathrm{NAE}_D^n = \{\mathbf{d} \in D^n : \neg(d_1 = d_2 = \cdots = d_n)\}$$

In the statement of Theorem 8 and further, we use $(A; S_1, \ldots, S_k)$ to denote a structure with universe A and relations $S_1, \ldots, S_k$.

▶ **Theorem 8** (Corollary 1.2 in [9]). *For any A and B of size at least 2, the problem $\{\exists, \wedge\}$-PMC$((A; \mathrm{Rb}_A^{2|A|}), (B; \mathrm{NAE}_B^{2|A|}))$ is NP-complete.*

Given this hardness result, the complexity classification is a simple consequence of Theorem 7.

▶ **Theorem 9** ($\mathcal{L} = \{\exists, \wedge, \vee\}$). *Let $(\mathbb{A}, \mathbb{B})$ be an $\mathcal{L}$-PMC template. If there is a constant homomorphism from $\mathbb{A}$ to $\mathbb{B}$, then $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$ is in L (in fact, decidable in constant time), otherwise $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$ is NP-complete.*

Proof. If there exists a constant homomorphism $f : \mathbb{A} \to \mathbb{B}$, say with image $\{b\}$, then all the relations $R^{\mathbb{B}}$ in $\mathbb{B}$ contain the constant tuple $(b, b, \ldots, b)$. It follows that every input sentence is satisfied in $\mathbb{B}$ by evaluating the existentially quantified variables to b; therefore, `Yes` is always a correct output.

If there is no constant homomorphism $\mathbb{A} \to \mathbb{B}$, we observe that no multi-homomorphism from $\mathbb{A}$ to $\mathbb{B}$ contains a constant homomorphism (as the set of multi-homomorphisms of a PMC template is closed under containment). It follows that the image of any "rainbow" tuple of A under any multi-homomorphism from $\mathbb{A}$ to $\mathbb{B}$ does not contain any constant tuple, and so any multi-homomorphism from $\mathbb{A}$ to $\mathbb{B}$ is a multi-homomorphism from $(A; \mathrm{Rb}_A^{2|A|})$ to $(B; \mathrm{NAE}_B^{2|A|})$. The reduction from Theorem 7 and the hardness from Theorem 8 conclude the proof. ◀

5 Positive fragment

We now turn our attention to the more complex case – the positive equality-free logic, that is, the $\mathcal{L}$-logic with $\mathcal{L} = \{\exists, \forall, \wedge, \vee\}$. We again fix this $\mathcal{L}$ for the entire section.

5.1 Witnesses for quantified formulas

It will be convenient to work with $\mathcal{L}$-formulas of the special form

$$\phi(x_1,\dots,x_n) = \forall y_1 \exists z_1 \forall y_2 \exists z_2 \dots \forall y_m \exists z_m \ \phi'(\mathbf{x},\mathbf{y},\mathbf{z}), \tag{3}$$

where ϕ' is quantifier-free. Note that each formula is equivalent to a formula in this form (by transforming to prenex normal form and adding dummy quantification as needed) and the conversion can be done in logarithmic space.

Observe that for a structure $\mathbb{A}$ and a tuple $\mathbf{a} \in A^n$, we have $\mathbb{A} \vDash \phi(\mathbf{a})$ if and only if there exist functions $\alpha_1 : A \to A$, $\alpha_2 : A^2 \to A$, …, $\alpha_m : A^m \to A$ which give us evaluations of the existentially quantified variables given the value of the previous universally quantified variables, i.e., these functions satisfy $\mathbb{A} \vDash \phi'(\mathbf{a},\mathbf{c},\alpha_1(c_1),\alpha_2(c_1,c_2),\dots,\alpha_m(c_1,\dots,c_m))$ for every $\mathbf{c} \in A^m$. We call such functions *witnesses* for $\mathbb{A} \vDash \phi(\mathbf{a})$.

We state a simple consequence of this viewpoint, a version of Lemma 5.

▶ **Lemma 10.** *Let f be a surjective multi-homomorphism from $\mathbb{A}$ to $\mathbb{B}$, let $\phi(x_1,\dots,x_n)$ be an $\mathcal{L}$-formula in the same signature as $\mathbb{A}$ and $\mathbb{B}$, and let $\mathbf{a} \in A^n$, $\mathbf{b} \in B^n$. If $\mathbb{A} \vDash \phi(\mathbf{a})$ and $\mathbf{b} \in f(\mathbf{a})$, then $\mathbb{B} \vDash \phi(\mathbf{b})$.*

In particular, if there exists a surjective multi-homomorphism from $\mathbb{A}$ to $\mathbb{B}$, and ϕ is an $\mathcal{L}$-sentence such that $\mathbb{A} \vDash \phi$, then $\mathbb{B} \vDash \phi$.

Proof. The claim holds for quantifier-free $\mathcal{L}$-formulas by Lemma 5.

Next, we assume that ϕ is of the form (3) and select witnesses α_1, …, α_m for $\mathbb{A} \vDash \phi(\mathbf{a})$. Let $g : B \to A$ be any function such that $b \in f(g(b))$ for every $b \in B$, which exists as f is surjective. We claim that any functions β_1, …, β_m such that $\beta_i(b_1,\dots,b_i) \in f(\alpha_i(g(b_1),\dots,g(b_i)))$ for every $i \in [m]$, are witnesses for $\mathbb{B} \vDash \phi(\mathbf{b})$. Indeed, for all $\mathbf{d} \in B^m$, we have $\mathbb{A} \vDash \phi'(\mathbf{a}, g(\mathbf{d}), \alpha_1(g(d_1)),\dots,\alpha_m(g(d_1),\dots,g(d_m)))$, and also $\mathbf{b} \in f(\mathbf{a})$, $\mathbf{d} \in f(g(\mathbf{d}))$, and $\beta_i(d_1,\dots,d_i) \in f(\alpha_i(g(d_1),\dots,g(d_i)))$ (by the assumption, choice of g, and choice of β_i, respectively); therefore, $\mathbb{B} \vDash \phi'(\mathbf{b},\mathbf{d},\beta_1(d_1),\dots,\beta_m(d_1,\dots,d_m))$ by the first paragraph. ◀

5.2 Characterization of templates and p-$\mathcal{L}$-definability

Unlike in the existential case, both characterizations require surjective and multi-valued functions. The core of these characterizations is an adjustment of (2) for surjective homomorphisms.

▶ **Lemma 11.** *Let $\mathbb{A}$ be a structure with $A = [k]$ and m, n be arbitrary positive integers. Then there exists a formula $\phi(x_{1,1},\dots,x_{1,n},x_{2,1},\dots,\dots,x_{k,n})$ such that, for any structure $\mathbb{E}$ similar to $\mathbb{A}$ with $|E| \leq m$, we have $\mathbb{E} \vDash \phi(e_{1,1},\dots,e_{k,n})$ if and only if the mapping $i \mapsto \{e_{i,1},\dots,e_{i,n}\}$, $i \in [k]$ is contained in a surjective multi-homomorphism from $\mathbb{A}$ to $\mathbb{E}$.*

Proof. For every function h from $[m]$ to $[k]$ we take a formula $\phi_h(x_{1,1},\dots,x_{k,n},z_1,\dots,z_m)$ such that, for any structure $\mathbb{E}$ in the signature of $\mathbb{A}$, we have $\mathbb{E} \vDash \phi_h(e_{1,1},\dots,e_{k,n},e'_1,\dots,e'_m)$ if and only if the mapping $i \mapsto \{e_{i,1},\dots,e_{i,n}\} \cup \bigcup_{h(l)=i} e'_l$, $1 \leq i \leq k$, is a multi-homomorphism from $\mathbb{A}$ to $\mathbb{E}$. Such a formula can be obtained by directly translating the definition of a multi-homomorphism into the language of logic, similarly to (2).

We claim that the formula ϕ obtained by taking the disjunction of ϕ_h over all $h : [m] \to [k]$ and universally quantifying the variables z_1, …, z_m satisfies the requirement of the lemma, provided $|E| \leq m$. Indeed, on the one hand, if $\mathbb{E} \vDash \phi(e_{1,1},\dots,e_{k,n})$, then for every evaluation of the z variables, some ϕ_h must be satisfied. We choose any evaluation that covers the whole set E (which is possible since $|E| \leq m$) and the satisfied disjunct ϕ_h then gives us the required surjective multi-homomorphism from $\mathbb{A}$ to $\mathbb{E}$ (by the choice of ϕ_h). On the other hand, if $i \mapsto \{e_{i,1}, \dots, e_{i,n}\}$ is contained in a surjective multi-homomorphism f, then for any evaluation $\varepsilon_E(z_1)$, …, $\varepsilon_E(z_m)$ of the universally quantified variables, a disjunct ϕ_h is satisfied whenever $\varepsilon_E(z_l) \in f(h(l))$ for every $l \in [m]$. Such an h exists since f is surjective. ◀

▶ **Proposition 12.** *A pair $(\mathbb{A}, \mathbb{B})$ of similar structures is an $\mathcal{L}$-PMC template if and only if there exists a surjective multi-homomorphism from $\mathbb{A}$ to $\mathbb{B}$.*

Proof. For the forward implication, consider the sentence obtained by existentially quantifying all the variables in the formula ϕ provided by Lemma 11 (with $m \geq |A|, |B|$). This sentence is true in $\mathbb{A}$ (as there exists a surjective multi-homomorphism from $\mathbb{A}$ to $\mathbb{A}$ – the identity), so it must be true in $\mathbb{B}$, giving us a surjective multi-homomorphism from $\mathbb{A}$ to $\mathbb{B}$. The backward implication follows from Lemma 10. ◀

An example which shows that one cannot replace in Proposition 12 "surjective multi-homomorphism" by "(multi-)homomorphism" is the input formula $\varphi = \forall x \exists y R(x, y)$ ("there are no sinks") for a template where $\mathbb{A}$ is a digraph with no sinks and $\mathbb{B}$ is, say, $\mathbb{A}$ plus an isolated vertex.

The following characterization of promise definability is also a straightforward consequence of Lemmata 10 and 11.

▶ **Theorem 13.** *Let $(\mathbb{A}, \mathbb{B})$ and $(\mathbb{C}, \mathbb{D})$ be $\mathcal{L}$-PMC templates such that $A = C$ and $B = D$. Then $(\mathbb{C}, \mathbb{D})$ is p-$\mathcal{L}$-definable from $(\mathbb{A}, \mathbb{B})$ if and only if $\mathrm{SMuHom}(\mathbb{A}, \mathbb{B}) \subseteq \mathrm{SMuHom}(\mathbb{C}, \mathbb{D})$. Moreover, in such a case, $\mathcal{L}\text{-PMC}(\mathbb{C}, \mathbb{D}) \leq \mathcal{L}\text{-PMC}(\mathbb{A}, \mathbb{B})$.*

Proof. The theorem is proved in the same way as Theorem 7; using Lemma 10 instead of Lemma 5 for the forward implication, and the formula provided by Lemma 11 instead of (2) for the backward implication. ◀

5.3 Membership

Clearly, every $\mathcal{L}$-MC, as well as $\mathcal{L}$-PMC, is in PSPACE. We now give a generalization of the remaining membership results from [12] using an appropriate generalization of "A-shops" and "E-shops" from that paper. We say that a surjective multi-homomorphism f from $\mathbb{A}$ to $\mathbb{B}$ is an A-smuhom if there exists $a^* \in A$ such that $f(a^*) = B$. We also say that $(\mathbb{A}, \mathbb{B})$ admits an A-smuhom in such a case. We call f an E-smuhom if $f^{-1}(b^*) = A$ for some $b^* \in B$. Finally, we call f an AE-smuhom if it is simultaneously an A-smuhom and an E-smuhom.

An additional simple reduction will be useful in the proof of the membership result (Theorem 15) and later as well. We say that an $\mathcal{L}$-PMC template $(\mathbb{C}, \mathbb{D})$ is a *relaxation* of an $\mathcal{L}$-PMC template $(\mathbb{A}, \mathbb{B})$ if $(\mathbb{C}, \mathbb{A})$ and $(\mathbb{B}, \mathbb{D})$ are $\mathcal{L}$-PMC templates. Recall that, by Proposition 12, the property is equivalent to the existence of surjective multi-homomorphisms from $\mathbb{C}$ to $\mathbb{A}$ and from $\mathbb{B}$ to $\mathbb{D}$.

▶ **Proposition 14.** *Let $(\mathbb{A}, \mathbb{B})$ and $(\mathbb{C}, \mathbb{D})$ be $\mathcal{L}$-PMC templates. If $(\mathbb{C}, \mathbb{D})$ is a relaxation of $(\mathbb{A}, \mathbb{B})$, then $\mathcal{L}\text{-PMC}(\mathbb{C}, \mathbb{D}) \leq \mathcal{L}\text{-PMC}(\mathbb{A}, \mathbb{B})$.*

Proof. The trivial reduction, which does not change the input, works. Indeed, `Yes`-instances of $\mathcal{L}\text{-PMC}(\mathbb{C}, \mathbb{D})$ are `Yes`-instances of $\mathcal{L}\text{-PMC}(\mathbb{A}, \mathbb{B})$ since $(\mathbb{C}, \mathbb{A})$ is an $\mathcal{L}$-PMC template, and `No`-instances of $\mathcal{L}\text{-PMC}(\mathbb{C}, \mathbb{D})$ are `No`-instances of $\mathcal{L}\text{-PMC}(\mathbb{A}, \mathbb{B})$ since $(\mathbb{B}, \mathbb{D})$ is an $\mathcal{L}$-PMC template. ◀

▶ **Theorem 15.** *Let* $(\mathbb{A},\mathbb{B})$ *be an* $\mathcal{L}$-PMC *template. Then the following holds.*

1. *If* $(\mathbb{A},\mathbb{B})$ *admits an* A-smuhom*, then* $\mathcal{L}$-PMC$(\mathbb{A},\mathbb{B})$ *is in* NP*.*
2. *If* $(\mathbb{A},\mathbb{B})$ *admits an* E-smuhom*, then* $\mathcal{L}$-PMC$(\mathbb{A},\mathbb{B})$ *is in* coNP*.*
3. *If* $(\mathbb{A},\mathbb{B})$ *admits an* AE-smuhom*, then* $\mathcal{L}$-PMC$(\mathbb{A},\mathbb{B})$ *is in* L*.*

Proof. For the first item, let f be an A-smuhom from $\mathbb{A}$ to $\mathbb{B}$ with $f(a^*) = B$, and consider an input ϕ in the special form (3), i.e., $\phi = \forall y_1 \exists z_1 \forall y_2 \exists z_2 \dots \forall y_m \exists z_m \ \phi'(\mathbf{y},\mathbf{z})$, where ϕ' is quantifier-free. We answer `Yes` if there exists $\mathbf{a} \in A^m$ such that $\mathbb{A} \models \phi'(a^*, a^*, \dots, a^*, \mathbf{a})$; this can be clearly decided in NP. It is clear that the answer is `Yes` whenever ϕ is a `Yes`-instance of $\mathcal{L}$-PMC$(\mathbb{A},\mathbb{B})$. On the other hand, if $\mathbb{A} \models \phi'(a^*, \dots, a^*, \mathbf{a})$, then any functions $\beta_1 : B \to B$, $\dots$, $\beta_m : B^m \to B$ such that $\beta_i(b_1, \dots, b_i) \in f(a_i)$ (for all $i \in [m]$ and $b_1, \dots, b_m \in B$) provide witnesses for $\mathbb{B} \models \phi$ by Lemma 5. Therefore, if ϕ is a `No`-instance, then the answer is `No`, as needed.

The second item follows by the duality argument.

In the case $\mathbb{A} = \mathbb{B}$, the third item can be proved in an analogous way (by eliminating both quantifiers instead of just one), see Corollary 9 in [12]. For the general case, we will construct $\mathbb{C}$ such that there is an AE-smuhom from $\mathbb{C}$ to $\mathbb{C}$ and there are surjective multi-homomorphisms from $\mathbb{A}$ to $\mathbb{C}$ and from $\mathbb{C}$ to $\mathbb{B}$. Then $(\mathbb{A},\mathbb{B})$ will be a relaxation of $(\mathbb{C},\mathbb{C})$ by Proposition 12, and then membership of $\mathcal{L}$-PMC$(\mathbb{A},\mathbb{B})$ in L will follow from Proposition 14 and the mentioned Corollary 9 in [12]. Let f be an AE-smuhom from $\mathbb{A}$ to $\mathbb{B}$ with $f(a^*) = B$ and $f^{-1}(b^*) = A$, and define a surjective multi-valued function f' from A to B by $f'(a^*) = B$ and $f'(a) = \{b^*\}$ if $a \neq a^*$. Note that f' is contained in f, so f' is a surjective multi-homomorphism from $\mathbb{A}$ to $\mathbb{B}$. We define $\mathbb{C}$ as the "image" of $\mathbb{A}$ under f', that is, $C = B$ and $R^{\mathbb{C}} = \cup_{\mathbf{a} \in R^{\mathbb{A}}} f'(\mathbf{a})$ for each relation symbol R. Clearly, f' is a surjective multi-homomorphism from $\mathbb{A}$ to $\mathbb{C}$ and the identity is a surjective homomorphism from $\mathbb{C}$ to $\mathbb{B}$. It remains to find an AE-smuhom from $\mathbb{C}$ to $\mathbb{C}$. We claim that g defined by $g(b^*) = \{b^*\}$ and $g(c) = C$ for $c \neq b^*$ is such an AE-smuhom. Indeed, if $\mathbf{c} \in R^{\mathbb{C}}$, then $\mathbf{c} \in f'(\mathbf{a})$ for some $\mathbf{a} \in R^{\mathbb{A}}$. By the definition of f', we necessarily have $a_i = a^*$ whenever $c_i \neq b^*$; therefore, $f'(\mathbf{a}) \supseteq g(\mathbf{c})$. But $f'(\mathbf{a}) \subseteq R^{\mathbb{C}}$ as $f' \in \mathrm{SMuHom}(\mathbb{A},\mathbb{C})$, and we are done. ◀

These membership results together with the (more involved) hardness results were sufficient for the tetrachotomy in [12]. One problem with generalizing this tetrachotomy is that, unlike in the non-promise setting, an $\mathcal{L}$-PMC template can admit an A-smuhom and an E-smuhom, but no AE-smuhom. However, such a situation cannot happen for digraphs.

▶ **Proposition 16.** *Let* $(\mathbb{A},\mathbb{B})$ *be an* $\mathcal{L}$-PMC *template such that* $\mathbb{A}$ *and* $\mathbb{B}$ *are digraphs. If* $(\mathbb{A},\mathbb{B})$ *admits an* A-smuhom *and an* E-smuhom*, then it admits an* AE-smuhom*.*

Proof. See Appendix A. ◀

5.4 Hardness

As a consequence of Theorems 8 and 13, we obtain the following hardness result.

▶ **Theorem 17.** *Let* $(\mathbb{A},\mathbb{B})$ *be an* $\mathcal{L}$-PMC *template.*

1. *If there is no* E-smuhom *from* $\mathbb{A}$ *to* $\mathbb{B}$*, then* $\mathcal{L}$-PMC$(\mathbb{A},\mathbb{B})$ *is* NP*-hard.*
2. *If there is no* A-smuhom *from* $\mathbb{A}$ *to* $\mathbb{B}$*, then* $\mathcal{L}$-PMC$(\mathbb{A},\mathbb{B})$ *is* coNP*-hard.*

Proof. If there exists no E-smuhom from $\mathbb{A}$ to $\mathbb{B}$, then $\mathrm{SMuHom}(\mathbb{A},\mathbb{B})$ is contained in $\mathrm{SMuHom}((A; \mathrm{Rb}_A^{2|A|}), (B; \mathrm{NAE}_B^{2|A|}))$. Theorem 8 and Theorem 13 then imply the first item. The second item follows by the duality argument. ◀

In the non-promise setting, the absence of A-smuhoms and E-smuhoms is sufficient for PSPACE-hardness [12, 13]. This most involved part of the tetrachotomy result seems much more challenging in the promise setting and we do not have strong reasons to believe that templates without A-smuhoms and E-smuhoms are necessarily PSPACE-hard. Nevertheless, we are able to prove some additional hardness results which will cover all the extensions of $\mathcal{L}$.

▶ **Proposition 18.** *$\mathcal{L}$-PMC$((A; =_A), (B; =_B))$ is PSPACE-hard for any A, B such that $|A| \geq |B| \geq 2$.*

Note here that surjective multi-homomorphisms from $(A; =_A)$ to $(B; =_B)$ are exactly the surjective multi-valued functions from A to B of multiplicity one. In particular, if $|A| < |B|$, then $((A; =_A), (B; =_B))$ is not an $\mathcal{L}$-PMC template.

Proof. We start by noticing that the template $((A; =_A), ([2]; =_{[2]}))$ is a relaxation of $(\mathbb{A}, \mathbb{B}) := ((A; =_A), (B; =_B))$. So by Proposition 14, it is enough to prove the claim in the case $B = [2]$. For simplicity, we assume that $A = [k]$ $(k \geq 2)$. We prove the PSPACE-hardness by a reduction from $\mathcal{L}$-MC$(\mathbb{B})$, a PSPACE-hard problem by, e.g., [14]. Consider an input ϕ to $\mathcal{L}$-MC$(\mathbb{B})$ in the special form (3), i.e., $\phi = \forall y_1 \exists z_1 \forall y_2 \exists z_2 \dots \forall y_m \exists z_m \ \phi'(\mathbf{y}, \mathbf{z})$, where ϕ' is quantifier-free. We need to find a log-space computable formula ψ such that $\mathbb{B} \vDash \phi$ implies $\mathbb{A} \vDash \psi$ (so that `Yes`-instances of MC$(\mathbb{B})$ are transformed to `Yes`-instances of $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$) and $\mathbb{B} \vDash \psi$ implies $\mathbb{B} \vDash \phi$ (so that `No`-instances are transformed to `No`-instances).

The rough idea to construct ψ is to reinterpret the values in $A = [k]$ as values in $B = [2]$ via a mapping $A \to B$. We set

$$\psi = \forall x_1 \forall x_2 \ \exists x_3 \exists x_4 \dots \exists x_k \ \ (x_1 = x_2) \ \vee \bigwedge_{f : A \to B} \rho_f, \quad \text{where} \tag{4}$$

$$\rho_f = (\forall y_1' \exists z_1 \dots \forall y_m' \exists z_m) \ \ (\exists y_1 \dots \exists y_m) \left(\bigwedge_{i=1}^{m} \sigma[f, y_i', y_i] \right) \wedge \phi'(\mathbf{y}, \mathbf{z}) \tag{5}$$

$$\sigma[f, y_i', y_i] = \bigvee_{a \in A} \left((y_i' = x_a) \wedge (y_i = x_{f(a)}) \right) \tag{6}$$

Observe first that ψ can be constructed from ϕ in logarithmic space.

Next, we verify that $\mathbb{B} \vDash \psi$ implies $\mathbb{B} \vDash \phi$. So, we suppose $\mathbb{B} \vDash \psi$ and aim to find witnesses $\beta_1, \dots, \beta_m$ for $\mathbb{B} \vDash \phi$; to this end, let $\mathbf{c}$ be some tuple in B^m that corresponds to evaluations of universally quantified variables in ϕ. We evaluate the variables x_1 and x_2 in ψ as $\varepsilon_B(x_1) = 1$ and $\varepsilon_B(x_2) = 2$, and pick an evaluation $\varepsilon_B(x_3), \dots, \varepsilon_B(x_k)$ making ψ true in $\mathbb{B}$. Set $f(a) = \varepsilon_B(x_a)$, $a \in A$. The first disjunct of (4) is not satisfied, so ρ_f is satisfied with this choice of ε_B. When it is the turn to evaluate y_i', we set $\varepsilon_B(y_i') = c_i$ and define $\beta_i(c_1, \dots, c_i) = \varepsilon_B(z_i)$, where $\varepsilon_B(z_i)$ is a satisfactory evaluation of z_i. Inspecting the definition (6), we see that $y_1, \dots, y_m$ are necessarily evaluated as $\varepsilon_B(y_1) = c_1, \dots, \varepsilon_B(y_m) = c_m$: indeed, if a disjunct $(y_i' = x_a) \wedge (y_i = x_{f(a)})$ is satisfied, then $c_i = \varepsilon_B(y_i') = \varepsilon_B(x_a)$ and $\varepsilon_B(y_i) = \varepsilon_B(x_{f(a)}) = \varepsilon_B(x_{\varepsilon_B(x_a)}) = \varepsilon_B(x_a)$; in particular, $\varepsilon_B(y_i) = c_i$. Therefore, the conjunct $\phi'(\mathbf{y}, \mathbf{z})$ in (5) ensures $\mathbb{B} \vDash \phi'(\mathbf{c}, \beta_1(c_1), \dots, \beta_m(c_1, \dots, c_m))$. As $\mathbf{c}$ was chosen arbitrarily, we get that $\beta_1, \dots, \beta_m$ are witnesses for $\mathbb{B} \vDash \phi$, as required.

We now suppose that $\beta_1, \dots, \beta_m$ are witnesses for $\mathbb{B} \vDash \phi$, and aim to show that $\mathbb{A} \vDash \psi$. Because of the first disjunct of (4), it is enough to consider only evaluations of x_1 and x_2 with $\varepsilon_A(x_1) \neq \varepsilon_A(x_2)$. Since any bijection, regarded as a surjective multi-homomorphism from $\mathbb{A}$ to $\mathbb{A}$ of multiplicity one, preserves $\mathcal{L}$-formulas (in the sense of Lemma 10), then we can as well assume that $\varepsilon_A(x_1) = 1$ and $\varepsilon_A(x_2) = 2$. We evaluate the remaining x variables as $\varepsilon_A(x_a) = a$, $a = 3, 4, \dots, k$. We take a function $f : A \to B$ and argue that ρ_f is satisfied in $\mathbb{A}$. Given a selection of $\varepsilon_A(y'_i)$, we evaluate z_i as $\varepsilon_A(z_i) = \beta_i(f(\varepsilon_A(y'_1)), \dots, f(\varepsilon_A(y'_i)))$, and we define the evaluation of the remaining variables by $\varepsilon_A(y_i) = f(\varepsilon_A(y'_i))$. With these choices, each $\sigma[f, y'_i, y_i]$ is satisfied because of the disjunct $a = \varepsilon_A(y'_i)$ in (6). The second conjunct in (5), $\phi'(\mathbf{y}, \mathbf{z})$, is also satisfied: we know $\mathbb{B} \vDash \phi'(\mathbf{c}, \beta_1(c_1), \dots, \beta_m(c_1, \dots, c_m))$ in particular for $c_1 = f(\varepsilon_A(y'_1))$, $\dots$, $c_m = f(\varepsilon_A(y'_m))$ and, with this $\mathbf{c}$, it is apparent from the choice of evaluations that $\mathbb{B} \vDash \phi'(\mathbf{c}, \beta_1(c_1), \dots, \beta_m(c_1, \dots, c_m))$ is equivalent to $\mathbb{A} \vDash \phi'(\varepsilon_A(y_1), \dots, \varepsilon_A(y_m), \varepsilon_A(z_1), \dots, \varepsilon_A(z_m))$. The proof of $\mathbb{A} \vDash \psi$ is concluded. ◀

It follows that $\{\exists, \forall, \wedge, \vee, =\}$-PMC over any template is PSPACE-hard and so is, by the duality argument, $\{\exists, \forall, \wedge, \vee, \neq\}$-PMC. The next proposition implies PSPACE-hardness for $\{\exists, \forall, \wedge, \vee, \neg\}$-PMC.

▶ **Proposition 19.** *Let* $(\mathbb{A}, \mathbb{B})$ *be an* $\mathcal{L}$-PMC *template which is closed under complementation. Then* $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$ *is* PSPACE*-hard.*

Proof. Suppose that $(\mathbb{A}, \mathbb{B})$ is closed under complementation. We define an equivalence relation $\sim_A$ on A by considering two elements equivalent if they play the same role in every relation of $\mathbb{A}$. Formally, $a \sim a'$ if for every symbol R from the signature, every coordinate $i \in [\mathrm{ar}(R)]$, and every $\mathbf{c}, \mathbf{c}' \in A^{\mathrm{ar}(R)}$, if $c_i = a$, $c'_i = a'$, $c_j = c'_j$ for all $j \in [\mathrm{ar}(R)] \setminus \{i\}$, and $\mathbf{c} \in R^{\mathbb{A}}$, then $\mathbf{c}' \in R^{\mathbb{A}}$. We define an equivalence relation $\sim_B$ on B analogously. Notice that $\sim_A$ (resp., $\sim_B$) is indeed an equivalence relation; let m and n denote the number of equivalence classes of $\sim_A$ and $\sim_B$, respectively.

Observe that $m, n \geq 2$. Indeed, otherwise any nonempty relation in the corresponding template contains all the tuples, and we do not allow such structures in this paper.

Let $\mathbb{C} = (A; \sim_A)$ and $\mathbb{D} = (B; \sim_B)$. We claim that every surjective multi-homomorphism f from $\mathbb{A}$ to $\mathbb{B}$ preserves $\sim$, i.e., is a surjective multi-homomorphism from $\mathbb{C}$ to $\mathbb{D}$. Consider $a, a' \in A$, and $b, b' \in B$ such that $a \sim_A a'$, $b \in f(a)$, and $b' \in f(a')$. In order to prove $b \sim_B b'$, take arbitrary R, i, $\mathbf{d}$, $\mathbf{d}'$ such that $d_i = b$, $d'_i = b'$, $d_j = d'_j$ for all $j \neq i$, and $\mathbf{d} \in R^{\mathbb{B}}$. Let $\mathbf{c}, \mathbf{c}' \in A^{\mathrm{ar}(R)}$ be tuples such that $c_i = a$, $c'_i = a'$, and $c_j = c'_j \in f^{-1}(d_j)$ for all $j \neq i$ (which exist as f is surjective). If $\mathbf{c} \notin R^{\mathbb{A}}$, then $\mathbf{c} \in \overline{R}^{\mathbb{A}}$ and, consequently, $\mathbf{d} \in f(\mathbf{c}) \subseteq \overline{R}^{\mathbb{B}}$ (as f is a surjective multi-homomorphism from $\mathbb{A}$ to $\mathbb{B}$), a contradiction with $\mathbf{d} \in R^{\mathbb{B}}$. Therefore, $\mathbf{c} \in R^{\mathbb{A}}$ and also $\mathbf{c}' \in R^{\mathbb{A}}$ as $a \sim_A a'$. Now $\mathbf{d}' \in f(\mathbf{c}') \subseteq R^{\mathbb{B}}$, and $b \sim_B b'$ follows.

By Theorem 13, $\mathcal{L}$-PMC$(\mathbb{C}, \mathbb{D}) \leq \mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$. Since there exists a surjective multi-valued function from A to B that preserves $\sim$ (namely, any $f \in \mathrm{SMuHom}(\mathbb{A}, \mathbb{B})$), we also know that $m \geq n$. The template $(\mathbb{E}, \mathbb{F}) := (([m]; =_{[m]}), ([n]; =_{[n]}))$ is a relaxation of $(\mathbb{C}, \mathbb{D})$, because there exists a surjective multi-homomorphism from $\mathbb{E}$ to $\mathbb{C}$ (a multi-valued function that maps i to the i-th equivalence class of $\sim_A$ under an arbitrary linear ordering of classes) and a surjective multi-homomorphism from $\mathbb{D}$ to $\mathbb{F}$ (a "multi"-valued function that maps every element in the i-th equivalence class of $\sim_B$ to $\{i\}$). By Proposition 14, $\mathcal{L}$-PMC$(\mathbb{E}, \mathbb{F}) \leq \mathcal{L}$-PMC$(\mathbb{C}, \mathbb{D})$; therefore, $\mathcal{L}$-PMC$(\mathbb{E}, \mathbb{F}) \leq \mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$. The former $\mathcal{L}$-PMC is PSPACE-hard by Proposition 18, so $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$ is PSPACE-hard, too. ◀

5.5 Summary and examples

The claims stated in Table 2 are now immediate consequences of the obtained results. Note that the claims remain true without the imposed restrictions on structures (i.e., we can allow singleton universes, nullary relations, etc.); the only nontrivial ingredient is the L-membership of the Boolean Sentence Value Problem [10].

We observe that the results imply a complete complexity classification in the case that one of the two template structures is *Boolean*, i.e., has a two-element universe.

▶ **Corollary 20** ($\mathcal{L} = \{\exists, \forall, \wedge, \vee\}$)**.** *Let* $(\mathbb{A}, \mathbb{B})$ *be an* $\mathcal{L}$-PMC *template.*

1. *If* $\mathbb{B}$ *is Boolean, then* $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$ *is in* L*, or is* NP*-complete, or* PSPACE*-complete.*
2. *If* $\mathbb{A}$ *is Boolean, then* $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$ *is in* L*, or is* coNP*-complete, or* PSPACE*-complete.*
3. *If* $\mathbb{A}$ *and* $\mathbb{B}$ *are Boolean, then* $\mathcal{L}$-PMC$(\mathbb{A}, \mathbb{B})$ *is in* L*, or is* PSPACE*-complete.*

Proof. If $\mathbb{B}$ is Boolean, then every E-smuhom (from $\mathbb{A}$ to $\mathbb{B}$) is an AE-smuhom. Moreover, if there is no A-smuhom, then every surjective multi-homomorphism is of multiplicity one, so it is also a multi-homomorphism from $(A; =_A)$ to $(B; =_B)$. The first item now follows from Proposition 18 and Theorem 13. The other items are easy as well. ◀

There are two wide gaps left for further investigation. First, it is unclear what the complexity is for the $\mathcal{L}$-PMC over templates that admit both an A-smuhom and an E-smuhom, but no AE-smuhom. While there is no such a digraph template, there are examples with one ternary or two binary relations, e.g., the following. We use ij as a shortcut for the pair (i, j).

$$\mathbb{A} = ([3];\ \{(1,2,3)\}), \quad \mathbb{B} = ([3];\ \{1,2,3\} \times \{2\} \times \{3\}\ \cup\ \{1,2\} \times \{2\} \times \{2,3\})$$
$$\mathbb{A} = ([3];\ \{12\},\ \{13\}), \quad \mathbb{B} = ([3];\ \{12,22,32\},\ \{12,13,22,23,33\})$$

The second gap is between simultaneous NP- and coNP-hardness, and PSPACE-hardness, when the template admits neither an A-smuhom nor an E-smuhom. Examples with unknown complexity include the following.

$$\mathbb{A} = ([3];\ \{(1,2,3)\}), \quad \mathbb{B} = ([3];\ \{2,3\} \times \{1,3\} \times \{1,2\})$$
$$\mathbb{A} = ([3];\ \{(1,2,3)\}), \quad \mathbb{B} = ([3]; \{1,2\} \times \{1,2\} \times \{3\}\ \cup\ \{1,3\} \times \{2\} \times \{2\})$$
$$\mathbb{A} = ([4];\ \{12,34\}), \quad \mathbb{B} = ([4];\ \{12,13,14,23,24,34,32\})$$

In an ongoing work, we have developed some more general PSPACE-hardness criteria, but the examples above remain elusive. The following equivalent *unary* version of the first example is an especially interesting template, whose $\mathcal{L}$-PMC is the problem described in the introduction.

$$\mathbb{A} = ([3];\ \{1\},\{2\},\{3\}), \quad \mathbb{B} = ([3];\ \{2,3\},\{1,3\},\{1,2\})$$

6 Conclusion

We gave a full complexity classification of $\{\exists, \wedge, \vee\}$-PMC, initiated an algebraic approach to $\{\exists, \forall, \wedge, \vee\}$-PMC, and applied it to provide several complexity results about this class of problems.

An interesting concrete problem, whose complexity is currently open, is the $\{\exists, \forall, \wedge, \vee\}$-PMC over the unary template above. As for the theory-building, the next natural step is to capture more complex reductions by means of surjective multi-homomorphisms; namely, the analogue of pp-constructions, which proved to be so useful in the theory of (Promise) CSPs [3, 2]. It may be also helpful to characterize and study the sets of surjective multi-homomorphisms in the spirit of [15, 7].

References

1 Per Austrin, Venkatesan Guruswami, and Johan Håstad. $(2+\epsilon)$-Sat is NP-hard. *SIAM J. Comput.*, 46(5):1554–1573, 2017. doi:10.1137/15M1006507.

2 Libor Barto, Jakub Bulín, Andrei A. Krokhin, and Jakub Opršal. Algebraic approach to promise constraint satisfaction. *J. ACM*, 68(4):28:1–28:66, 2021. doi:10.1145/3457606.

3 Libor Barto, Andrei Krokhin, and Ross Willard. Polymorphisms, and How to Use Them. In Andrei Krokhin and Stanislav Živný, editors, *The Constraint Satisfaction Problem: Complexity and Approximability*, volume 7 of *Dagstuhl Follow-Ups*, pages 1–44. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017. doi:10.4230/DFU.Vol7.15301.1.

4 Ferdinand Börner. Basics of Galois Connections. In Nadia Creignou, Phokion G. Kolaitis, and Heribert Vollmer, editors, *Complexity of Constraints - An Overview of Current Research Themes [Result of a Dagstuhl Seminar]*, volume 5250 of *Lecture Notes in Computer Science*, pages 38–67. Springer, 2008. doi:10.1007/978-3-540-92800-3_3.

5 Joshua Brakensiek and Venkatesan Guruswami. Promise Constraint Satisfaction: Structure Theory and a Symmetric Boolean Dichotomy. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'18, pages 1782–1801, Philadelphia, PA, USA, 2018. Society for Industrial and Applied Mathematics. doi:10.1137/1.9781611975031.117.

6 A. A. Bulatov. A dichotomy theorem for nonuniform CSPs. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 319–330, October 2017. doi:10.1109/FOCS.2017.37.

7 Catarina Carvalho and Barnaby Martin. The lattice and semigroup structure of multipermutations. *International Journal of Algebra and Computation*, 0(0):1–25, 2021. doi:10.1142/S0218196722500096.

8 Hubie Chen. Meditations on quantified constraint satisfaction. In Robert L. Constable and Alexandra Silva, editors, *Logic and Program Semantics - Essays Dedicated to Dexter Kozen on the Occasion of His 60th Birthday*, volume 7230 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2012. doi:10.1007/978-3-642-29485-3_4.

9 Venkatesan Guruswami and Euiwoong Lee. Strong inapproximability results on balanced rainbow-colorable hypergraphs. *Comb.*, 38(3):547–599, 2018. doi:10.1007/s00493-016-3383-0.

10 Nancy Lynch. Log space recognition and translation of parenthesis languages. *J. ACM*, 24(4):583–590, October 1977. doi:10.1145/322033.322037.

11 Florent Madelaine and Barnaby Martin. The complexity of positive first-order logic without equality. *ACM Trans. Comput. Logic*, 13(1), January 2012. doi:10.1145/2071368.2071373.

12 Florent R. Madelaine and Barnaby Martin. A tetrachotomy for positive first-order logic without equality. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*, pages 311–320. IEEE Computer Society, 2011. doi:10.1109/LICS.2011.27.

13 Florent R. Madelaine and Barnaby Martin. On the complexity of the model checking problem. *SIAM J. Comput.*, 47(3):769–797, 2018. doi:10.1137/140965715.

14 Barnaby Martin. First-order model checking problems parameterized by the model. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, volume 5028 of *Lecture Notes in Computer Science*, pages 417–427. Springer, 2008. doi:10.1007/978-3-540-69407-6_45.

15 Barnaby Martin. The lattice structure of sets of surjective hyper-operations. 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 368–382. Springer, 2010. doi:10.1007/978-3-642-15396-9_31.

16 Barnaby Martin. Quantified Constraints in Twenty Seventeen. In Andrei Krokhin and Stanislav Živný, editors, *The Constraint Satisfaction Problem: Complexity and Approximability*, volume 7 of *Dagstuhl Follow-Ups*, pages 327–346. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017. doi:10.4230/DFU.Vol7.15301.327.

17 Barnaby Martin and Jos Martin. The complexity of positive first-order logic without equality II: the four-element case. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, volume 6247 of *Lecture Notes in Computer Science*, pages 426–438. Springer, 2010. doi:10.1007/978-3-642-15205-4_33.

18 Dmitriy Zhuk. A proof of the CSP dichotomy conjecture. *J. ACM*, 67(5):30:1–30:78, August 2020. doi:10.1145/3402029.

19 Dmitriy Zhuk and Barnaby Martin. QCSP monsters and the demise of the chen conjecture. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 91–104. ACM, 2020. doi:10.1145/3357713.3384232.

A Proof of Proposition 16

Proof. Denote by R the unique binary symbol in the signature. Let f be an $\mathbb{A}$-smuhom from $\mathbb{A}$ to $\mathbb{B}$ with $f(a^*) = B$ and let g be an $\mathbb{E}$-smuhom from $\mathbb{A}$ to $\mathbb{B}$ with $g^{-1}(b^*) = A$.

If a^* is isolated in $\mathbb{A}$ (i.e., $(a, a^*), (a^*, a) \notin R^{\mathbb{A}}$ for every $a \in A$), then we define a surjective multi-valued function h by $h(a^*) = B$ and $h(a) = \{b^*\}$ for every $a \neq a^*$. It is a multi-homomorphism from $\mathbb{A}$ to $\mathbb{B}$ since for any $(a, a') \in R^{\mathbb{A}}$, we have $h(a, a') = \{(b^*, b^*)\}$, which is contained in $R^{\mathbb{B}}$ because $R^{\mathbb{A}}$ is nonempty, so $g(R^{\mathbb{A}}) \ni (b^*, b^*)$.

Suppose next that there is an edge $(a_1, a^*) \in R^{\mathbb{A}}$ but a^* has no outgoing edges in $\mathbb{A}$. Let b_1 be an arbitrary element from $f(a_1)$ and define h by $h(a^*) = B$ and $h(a) = \{b_1\}$ for every $a \neq a^*$. To verify that $h \in \mathrm{SMuHom}(\mathbb{A}, \mathbb{B})$, consider an edge $(a, a') \in R^{\mathbb{A}}$. As a^* has no outgoing edges in $\mathbb{A}$, we get $a \neq a^*$, so $h(a) = \{b_1\}$. Now $h(a, a') \subseteq \{b_1\} \times B$, which is contained in $R^{\mathbb{B}}$ because $R^{\mathbb{B}} \supseteq f(a_1, a^*) \supseteq \{b_1\} \times B$.

If a^* has an outgoing edge $(a^*, a_1) \in R^{\mathbb{A}}$ but no incoming edges, we proceed similarly, defining $h(a^*) = B$ and $h(a) = \{b_1\}$ for all $a \neq a^*$, where b_1 is an arbitrary element from $f(a_1)$.

Finally, suppose that $(a_1, a^*) \in R^{\mathbb{A}}$ and $(a^*, a_2) \in R^{\mathbb{A}}$ for some $a_1, a_2 \in A$. If there is an element $a_3 \in A$ with no outgoing (resp., incoming) edges, define h by $h(a_3) = B$ and $h(a) = \{b'\}$ for all $a \neq a_3$, where b' is an arbitrary element from $f(a_1)$ (resp., $f(a_2)$). If there is no such element a_3, then we define $h(a^*) = B$ and $h(a) = \{b^*\}$ for all $a \neq a^*$. Since g is surjective, and every $a \in A$ has both an incoming and an outgoing edge, then $(b, b^*) \in R^{\mathbb{B}}$ and $(b^*, b) \in R^{\mathbb{B}}$ for all $b \in B$, therefore, $h \in \mathrm{SMuHom}(\mathbb{A}, \mathbb{B})$.

The proof of Proposition 16 is concluded. ◀

Improved Sample Complexity Bounds for Branch-And-Cut

Maria-Florina Balcan
Computer Science and Machine Learning Departments,
Carnegie Mellon University, Pittsburgh, PA, USA

Siddharth Prasad
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Tuomas Sandholm
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
Optimized Markets, Inc., Pittsburgh, PA, USA
Strategic Machine, Inc., Pittsburgh, PA, USA
Strategy Robot, Inc., Pittsburgh, PA, USA

Ellen Vitercik
Department of Electrical Engineering and Computer Sciences,
University of California Berkeley, CA, USA

Abstract

The branch-and-cut algorithm for integer programming has a wide variety of tunable parameters that have a huge impact on its performance, but which are challenging to tune by hand. An increasingly popular approach is to use machine learning to configure these parameters based on a training set of integer programs from the application domain. We bound how large the training set should be to ensure that for any configuration, its average performance over the training set is close to its expected future performance. Our guarantees apply to parameters that control the most important aspects of branch-and-cut: node selection, branching constraint selection, and cut selection, and are sharper and more general than those from prior research.

2012 ACM Subject Classification Theory of computation → Integer programming; Theory of computation → Sample complexity and generalization bounds

Keywords and phrases Automated algorithm configuration, integer programming, machine learning theory, tree search, branch-and-bound, branch-and-cut, cutting planes, sample complexity, generalization guarantees, data-driven algorithm design

Digital Object Identifier 10.4230/LIPIcs.CP.2022.3

Funding This material is based on work supported by the National Science Foundation under grants CCF-1733556, CCF-1910321, IIS-1901403, and SES-1919453, the ARO under award W911NF2010081, the Defense Advanced Research Projects Agency under cooperative agreement HR00112020003, a Simons Investigator Award, an AWS Machine Learning Research Award, an Amazon Research Award, a Bloomberg Research Grant, and a Microsoft Research Faculty Fellowship.

1 Introduction

Branch-and-cut (B&C) is a powerful algorithmic paradigm that is the backbone of all modern integer programming (IP) solvers. The main components of B&C can be tuned and tweaked in a myriad of ways. The fastest commercial IP solvers like CPLEX and Gurobi employ an array of heuristics to make decisions at every stage of B&C to reduce the solving time as much as possible, and give the user freedom to tune the multitude of parameters influencing the search through the space of feasible solutions. However, tuning the parameters that control B&C in a principled way is an inexact science with little to no formal mathematical guidelines. A rapidly growing line of work studies machine-learning approaches to speeding

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 3; pp. 3:1–3:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

up the various aspects of B&C – in particular investigating whether high-performing B&C parameter configurations can be learned from a *training set* of typical IPs from the particular application at hand [2, 22, 37, 42, 25, 32, 27, 43, 29]. Complementing the substantial number of experimental approaches using machine learning for B&C, a recent generalization theory has developed in parallel that aims to provide a rigorous theoretical foundation for how well any B&C configuration learned from training IP data will perform on new unseen IPs [7, 9]. In particular, this line of theoretical research provides *sample complexity guarantees* that bound how large the training set should be to ensure that *no matter how the parameters are configured* (i.e., using any approach from prior research), the average performance of branch-and-cut over the training set is close to its expected future performance. Sample complexity bounds are important because with too small a training set, learning is impossible: a configuration may have strong average performance over the training set but terrible expected performance on future IPs. If the training set is too small, then no matter how the parameters are tuned, the resulting configuration will not have reliably better performance than any default configuration. State-of-the-art parameter tuning methods have historically come without any provable guarantees, and our results fill in that gap for a wide array of tunable B&C parameters. In this paper, we expand and improve upon the existing theory to develop a wider and sharper handle on the learnability of the key components of B&C.

1.1 Summary of main contributions

Our main contribution is a formalization of a general model of tree search, presented in Section 2.1, that allows us to improve and generalize prior results on the sample complexity of tuning B&C. In this model, the algorithm repeatedly chooses a leaf node of the search tree, performs a series of actions (for example, a cutting plane to apply and a constraint to branch on), and adds children to that leaf in the search tree. The algorithm will also fathom nodes when applicable. The node and action selection are governed by *scoring rules*, which assign a real-valued score to each node and possible action. For example, a node-selection scoring rule might equal the objective value of the node's LP relaxation. We focus on general tree search with *path-wise* scoring rules. At a high level, a score of a node or action is path-wise if its value only depends on information contained along the path between the root and that node, as is often the case in B&C. Many commonly used scoring rules are path-wise including the efficacy [4], objective parallelism [1], directed cutoff distance [15], and integral support [41] scoring rules, all used for cut selection by the leading open-souce solver SCIP [15]; the best-bound scoring rule for node selection; and the linear, product, and most-fractional scoring rules for variable selection using strong branching [1]. In Section 4, we show how this general model of tree search captures a wide array of B&C components, including node selection, general branching constraint selection, and cutting plane selection, simultaneously. We also provide experimental evidence that, in the case of cutting plane selection, the data-dependent tuning suggested by our model can lead to dramatic reductions in the number of nodes expanded by B&C.

In Section 3, we prove our main structural result: for any IP, the tree search parameter space can be partitioned into a finite number of regions such that in any one region, the resulting search tree is fixed. This is in spite of the fact that the B&C search tree can be an extremely unstable function of its parameters, with minuscule changes leading to exponentially better or worse performance [7, 9]. By analyzing the complexity of this partition, we prove our sample complexity bound. In particular, we relate the complexity of the partition to the *pseudo-dimension* of the set of functions that measure the performance of B&C as a function of the input IP. Pseudo-dimension (defined in Section 3) is a combinatorial notion from machine learning theory that measures the *intrinsic complexity* of a set of functions. At a high level, it measures how well a set of functions are able match complex patterns. Classic results from learning theory then allow us to translate our pseudo-dimension bound into a sample complexity guarantee [3], capturing the intuition that the more complex patterns one can fit (i.e., the larger the pseudo-dimension is), the more samples needed to generalize. The sample complexity bound grows linearly with the pseudo-dimension, so ideally, the pseudo-dimension will be polynomial in the size of the problem.

We show that the pseudo-dimension is only polynomial in the depth of the tree (which is, for example, at most the number of variables in the case of binary integer programming). By contrast, we might naïvely expect the pseudo-dimension to grow linearly with the number of arithmetic operations required to compute the B&C tree (as in Theorem 8.4 by Anthony and Bartlett [3]), which is exponential in the depth of the tree. In fact, our bound is exponentially smaller than the pseudo-dimension bound of prior research by Balcan et al. [9], which grows linearly with the total number of nodes in the tree. Their results apply to any type of scoring rule, path-wise or otherwise. By taking advantage of the path-wise structure, we are able to reason inductively over the depth of the tree, leading to our exponentially improved bound. Our results recover those of Balcan et al. [7], who only studied path-wise scoring rules for single-variable selection for branching. In contrast, we are able to handle many more of the critical components of tree search: node selection, general branching constraint selection, and cutting plane selection.

1.2 Additional related research

A growing body of research has studied how machine learning can be used to speed up the time it takes to solve integer programs, primarily from an empirical perspective, whereas we study this problem from a theoretical perspective. This line of research has included general parameter tuning procedures [25, 27, 24, 37], which are not restricted to any one aspect of B&C. Researchers have also honed in on specific aspects of tree search and worked towards improving those using machine learning. These include variable selection [29, 2, 13, 7, 16, 19], general branching constraint selection [44], cut selection [37, 39, 23, 9], node selection [36, 21], and heuristic scheduling [30, 10]. Machine learning approaches to large neighborhood search have also been used to speed up solver runtimes [38].

This paper contributes to a line of research that provides sample complexity guarantees for algorithm configuration, often by using structure exhibited by the algorithm's performance as a function of its parameters [20, 8, 7, 6, 9, 5]. This line of research has studied algorithms for clustering [8], computational biology [6], and integer programming [7, 9], among other computational problems. The main contribution of this paper is to provide a sharp yet general analysis of the performance of tree search as a function of its parameters.

A related line of research provides algorithm configuration procedures with provable guarantees that are agnostic to the specific algorithm that is being configured [31, 40] and are particularly well-suited for algorithms with a finite number of possible configurations (though they can be applied to algorithms with infinite parameter spaces by randomly sampling a finite set of configurations).

2 Main tree search model

In this section we present our general tree search model and situate it within the framework of sample complexity. Balcan et al. [9] studied the sample complexity of a much more general formulation of a tunable search algorithm without any inherent tree structure. Our formulation explicitly builds a tree.

Algorithm 1 Tree search.

```
Input: Root node $Q$, depth limit $\Delta$
Initialize $\mathcal{T} = Q$.
while $\mathcal{T}$ contains an unfathomed leaf do
    Select a leaf $Q$ of $\mathcal{T}$ that maximizes $\texttt{nscore}(\mathcal{T}, Q)$.
    if $\texttt{depth}(Q) = \Delta$ or $\texttt{fathom}(\mathcal{T}, Q, \texttt{None})$ then
        Fathom $Q$.
    else
        Select an action $A \in \texttt{actions}(\mathcal{T}, Q)$ that maximizes $\texttt{ascore}(\mathcal{T}, Q, A)$.
        if $\texttt{fathom}(\mathcal{T}, Q, A)$ then
            Fathom $Q$.
        else if $\texttt{children}(\mathcal{T}, Q, A) = \emptyset$ then
            Fathom $Q$.
        else
            Add all nodes in $\texttt{children}(\mathcal{T}, Q, A)$ to $\mathcal{T}$ as children of $Q$.
```

2.1 General model of tree search

Tree search starts with a root node. In each round of tree search, a leaf node Q is selected. At this node, one of three things may occur: (1) Q is fathomed, meaning it is never visited again, (2) some action is taken at Q, and then it is fathomed, or (3) some action is taken at Q, and then some number of children nodes of Q are added to the tree. (For example, an action might represent a decision about which variable to branch on.) This process repeats until the tree has no unfathomed leaves. More formally, there are functions `actions`, `children`, and `fathom` prescribing how the search proceeds. Given a partial tree $\mathcal{T}$ and a leaf Q of $\mathcal{T}$, $\texttt{actions}(\mathcal{T}, Q)$ outputs a set of actions available at Q. Given a partial tree $\mathcal{T}$, a leaf Q of $\mathcal{T}$, and an action $A \in \texttt{actions}(\mathcal{T}, Q)$, $\texttt{fathom}(\mathcal{T}, Q, A) \in \{\texttt{true}, \texttt{false}\}$ is a Boolean function used to determine when to fathom a leaf Q of $\mathcal{T}$ given that action $A \in \texttt{actions}(\mathcal{T}, Q) \cup \{\texttt{None}\}$ was taken at Q, and $\texttt{children}(\mathcal{T}, Q, A)$ outputs a (potentially empty) list of nodes representing the children of Q to be added to the search tree given that action A was taken at Q. Finally, $\texttt{nscore}(\mathcal{T}, Q)$ is a node-selection score that outputs a real-valued score for each leaf of $\mathcal{T}$, and $\texttt{ascore}(\mathcal{T}, Q, A)$ is an action-selection score that outputs a real-valued score for each action $A \in \texttt{actions}(\mathcal{T}, Q)$. These scores are heuristics that are meant to indicate the quality of exploring a node or performing an action.

Many aspects of B&C are governed by scoring rules [1]. For example, commonly used scoring rules for cutting plance selection include *efficacy* [4], which is the perpendicular distance from the current LP solution to the cutting plane; *parallelism* [1], which measures the angle between the objective and the normal vector to the cutting plane; and *directed cutoff* [15], which is the distance from the current LP solution to the cutting plane along the direction of the line segment connecting the LP solution to the current best incumbent integer solution For node selection, under the commonly used best-first node selection policy, $\texttt{nscore}(\mathcal{T}, Q)$ equals the objective value of the LP relaxation of the IP represented by the node Q. Finally, for variable selection, popular scoring rules include a maximum change in LP objective value after branching on the variable (where the maximum is taken over the two resulting children), the minimum change in the LP objective value, linear combinations of these two values, and the product of these two values [1]. Algorithm 1 is a formal description of tree search using these functions.

The key condition that enables us to derive stronger sample complexity bounds compared to prior research is the notion of a *path-wise* function, which was also used in prior research but only in the context of variable selection [7].

▶ **Definition 1** (Path-wise functions). *A function f on tree-leaf pairs is path-wise if for all $\mathcal{T}$ and $Q \in \mathcal{T}$, $f(\mathcal{T}, Q) = f(\mathcal{T}_Q, Q)$, where $\mathcal{T}_Q$ is the path from the root of $\mathcal{T}$ to Q. A function g on tree-leaf-action triples is path-wise if for all A, $f_A(\mathcal{T}, Q) := g(\mathcal{T}, Q, A)$ is path-wise.*

We assume that `actions`, `ascore`, `nscore` and `children` are path-wise, though `fathom` is not necessarily path-wise.

Many commonly-used scoring rules are path-wise. For example, scoring rules are often functions of the LP relaxation of the IP represented by a given node, and these scoring rules are path-wise. Specific examples include the efficacy, objective parallelism, directed cutoff distance, and integral support scoring rules used for cut selection; the best-bound scoring rule for node selection; and the linear, product, and most-fractional scoring rules for variable selection using strong branching. A point of clarification: the pathwise assumption is with respect to the numerical scores assigned to actions/nodes. The actual act of, for example, node selection, can depend on the entire tree. For example, consider the best-bound node selection rule in branch-and-cut, which chooses the node with the best LP estimate. Here, the scoring rule, which is the LP objective value itself, is pathwise, but ultimately the node that is selected depends on the LP bounds at every unexplored node of the tree. This is fine for our analysis. Similarly, the decision to fathom a node based on LP bounds is a decision that depends on the entire tree built so far, which is also captured by our analysis.

No one scoring rule is optimal across all application domains, and prior research on variable selection has shown that it can be advantageous to adapt the scoring rule to the application domain at hand [7]. To this end, Algorithm 1 can be tuned by two parameters $\mu \in [0, 1]$ and $\lambda \in [0, 1]$ that control action selection and node selection, respectively. Given two fixed path-wise action-selection scores $\texttt{ascore}_1$ and $\texttt{ascore}_2$, we define a new score by

$$\texttt{ascore}_\mu(\mathcal{T}, Q) = \mu \cdot \texttt{ascore}_1(\mathcal{T}, Q) + (1 - \mu) \cdot \texttt{ascore}_2(\mathcal{T}, Q).$$

Similarly, given two path-wise node-selection scores $\texttt{nscore}_1$ and $\texttt{nscore}_2$, we define

$$\texttt{nscore}_\lambda(\mathcal{T}, Q, A) = \lambda \cdot \texttt{nscore}_1(\mathcal{T}, Q, A) + (1 - \lambda) \cdot \texttt{nscore}_2(\mathcal{T}, Q, A).$$

Then, if $\texttt{nscore}_\lambda$ and $\texttt{ascore}_\mu$ are used as the scores in Algorithm 1, we can view the behavior of tree search as a function of μ and λ. The choice to use a convex combination of scores is not new: prior research has shown that this idea can lead to dramatic improvements in the case of single-variable branching [7]. Furthermore, the leading open source solver SCIP uses a hard-coded weighted sum of scoring rules to select cutting planes. More broadly, interpolating between two scores is a commonly-studied modeling choice in other machine learning topics such as clustering [8].

Finally, we assume there exists $b, k \in \mathbb{N}$ such that $|\texttt{actions}(\mathcal{T}, Q)| \le b$ for any $Q \in \mathcal{T}$, and $|\texttt{children}(\mathcal{T}, Q, A)| \le k$ for all Q, A.

2.2 Problem formulation

We now define the notion of a *sample complexity bound* more formally. Let $\mathcal{Q}$ denote the domain of possible input root nodes Q to Algorithm 1 (for example, the set of all IPs with n variables and m constraints). As is common in prior research on algorithm configuration [22, 37, 42, 25, 32, 27, 43], we assume there is some unknown distribution $\mathcal{D}$

over $\mathcal{Q}$. In the IP setting, $\mathcal{D}$ could represent, for example, typical scheduling IP instances solved by an airline company. The *sample complexity* of a class of real valued functions $\mathcal{F} = \{f : \mathcal{Q} \to \mathbb{R}\}$ is the minimum number of independent samples required from $\mathcal{D}$ so that with high probability over the samples, the empirical value of f on the samples is a good approximation of the expected value of f over $\mathcal{D}$, uniformly over all $f \in \mathcal{F}$. Formally, given an error parameter ε and confidence parameter δ, the sample complexity $N_{\mathcal{F}}(\varepsilon, \delta)$ is the minimum $N_0 \in \mathbb{N}$ such that for any $N \geq N_0$,

$$\Pr_{Q_1,\ldots,Q_N \sim \mathcal{D}} \left(\sup_{f \in \mathcal{F}} \left| \frac{1}{N} \sum_{i=1}^{N} f(Q_i) - \mathop{\mathbb{E}}_{Q \sim \mathcal{D}}[f(Q)] \right| \leq \varepsilon \right) \geq 1 - \delta$$

for all distributions $\mathcal{D}$ supported on $\mathcal{Q}$. Equivalently, our results bound the error $\varepsilon_{\mathcal{F}}(N, \delta)$ between the empirical value of any $f \in \mathcal{F}$ and its true expected value in terms of the number of training samples N and the confidence parameter δ. $N_{\mathcal{F}}(\varepsilon, \delta)$ is the number of samples required to achieve a prescribed error bound ε, while $\varepsilon_{\mathcal{F}}(N, \delta)$ provides an error bound for any number N of samples at hand. We provide bounds on $N_{\mathcal{F}}(\varepsilon, \delta)$ and $\varepsilon_{\mathcal{F}}(N, \delta)$ in terms of a common learning-theoretic measure of intrinsic complexity of $\mathcal{F}$ called *pseudo-dimension*, which is detailed in Section 3.

In the context of Algorithm 1, we study families of *tree-constant* cost functions. A cost function $\texttt{cost} : \mathcal{Q} \to \mathbb{R}$ is tree constant if $\texttt{cost}(Q)$ only depends on the tree built by Algorithm 1 on input Q (an example is tree size). Let $\texttt{cost}_{\mu,\lambda}(Q)$ denote this cost when Algorithm 1 is run using the scores $\texttt{ascore}_\mu = \mu \cdot \texttt{ascore}_1 + (1 - \mu) \cdot \texttt{ascore}_2$ and $\texttt{nscore}_\lambda = \lambda \cdot \texttt{nscore}_1 + (1 - \lambda) \cdot \texttt{nscore}_2$. We study the sample complexity of $\mathcal{F} = \{\texttt{cost}_{\mu,\lambda} : \mu, \lambda \in [0, 1]\}$. We emphasize that we primarily interpret tree-constant functions as proxies for run-time. In the context of integer programming, tree size is one such measure. A strength of these guarantees is that they apply no matter how the parameters are tuned: optimally or suboptimally, manually or automatically. For *any* configuration, these guarantees bound the difference between average performance over the training set and expected future performance on unseen IPs.

3 Generalization guarantees for tree search

In order to derive our sample complexity guarantees, we first prove a key structural property: the behavior of Algorithm 1 is piecewise constant as a function of the node-selection score parameter λ and the action-selection score parameter μ. We give a high-level outline of our approach. We first assume that the conditional checks $\texttt{fathom}(\mathcal{T}, Q, \cdot) = \texttt{true}$ (lines 4 and 8) are suppressed. Let $\mathcal{A}'$ denote Algorithm 1 without these checks (so $\mathcal{A}'$ fathoms a node if and only if the depth limit is reached or if the node has no children). The behavior of $\mathcal{A}'$ as a function of μ and λ can be shown to be piecewise constant using the same argument as in Claim 3.4 of Balcan et al. [7]. Given this, our first main technical contribution (Lemma 2) is a generalization of Claim 3.5 of Balcan et al. [7] that relates the behavior of $\mathcal{A}'$ to Algorithm 1. The argument in Balcan et al. [7] is specific to branching, but we are able to prove our result in a much more general setting. Our second main technical contribution (Lemma 4) is to establish piecewise structure when the node-selection score is controlled by $\lambda \in [0, 1]$. The main reason for this auxiliary step of analyzing $\mathcal{A}'$ is due to the fact that $\texttt{fathom}$ is *not* necessarily a path-wise function, and can depend on the state of the entire tree.

▶ **Lemma 2.** *Fix $\mu \in [0, 1]$. Let $\mathcal{T}$ and $\mathcal{T}'$ be the trees built by Algorithm 1 and $\mathcal{A}'$, respectively, using the action-selection score $\mu \cdot \texttt{ascore}_1 + (1 - \mu) \cdot \texttt{ascore}_2$. Let Q be any node in $\mathcal{T}$, and let $\mathcal{T}_Q$ be the path from the root of $\mathcal{T}$ to Q. Then, $\mathcal{T}_Q$ is a rooted subtree of $\mathcal{T}'$, no matter what node selection policy is used.*

Proof. Let t denote the length of the path $\mathcal{T}_Q$. Let $\mathcal{T}_Q$ be comprised of the sequence of nodes $(Q_1, \dots, Q_t)$ such that Q_1 is the root of $\mathcal{T}$, $Q_t = Q$, and for each τ, $Q_{\tau+1} \in \texttt{children}(\mathcal{T}_{Q_\tau}, Q_\tau, A_\tau)$ where $A_\tau \in \texttt{actions}(\mathcal{T}_{Q_\tau}, Q_\tau)$ is the action selected by Algorithm 1 at node Q_τ. We show that $(Q_1, \dots, Q_t)$ is a rooted path in $\mathcal{T}'$ as well.

Suppose for the sake of contradiction that this is not the case. Let $\tau \in \{2, \dots, t\}$ be the minimal index such that $(Q_1, \dots, Q_{\tau-1})$ is a rooted path in $\mathcal{T}'$, but there is no edge in $\mathcal{T}'$ from $Q_{\tau-1}$ to node Q_τ. There are two possible cases:

Case 1. $Q_{\tau-1}$ was fathomed by $\mathcal{A}'$. This case is trivially not possible since whenever $\mathcal{A}'$ fathoms a node, so does Algorithm 1 (recall $\mathcal{A}'$ was defined by suppressing fathoming conditions of Algorithm 1).

Case 2. $Q_\tau \notin \texttt{children}(\mathcal{T}', Q_{\tau-1}, A'_{\tau-1})$ where $A'_{\tau-1}$ is the action taken by $\mathcal{A}'$ at node $Q_{\tau-1}$. In this case, if $\texttt{children}(\mathcal{T}', Q_{\tau-1}, A'_{\tau-1}) = \emptyset$, then $Q_{\tau-1}$ would be fathomed by $\mathcal{A}'$, which cannot happen by the first case. Otherwise, if $\texttt{children}(\mathcal{T}', Q_{\tau-1}, A'_{\tau-1}) \neq \emptyset$, we show that we arrive at a contradiction due to the fact that the scoring rules, action-set functions, and children functions are all path-wise. Let $A'_{\tau-1}$ denote the action taken by $\mathcal{A}'$ at $Q_{\tau-1}$, and let $A_{\tau-1}$ denote the action taken by Algorithm 1 at $Q_{\tau-1}$. Since $\texttt{actions}$ is path-wise,

$$\texttt{actions}(\mathcal{T}, Q_{\tau-1}) = \texttt{actions}(\mathcal{T}_{Q_{\tau-1}}, Q_{\tau-1}) = \texttt{actions}(\mathcal{T}', Q_{\tau-1}).$$

Since $\texttt{ascore}_1$ and $\texttt{ascore}_2$ are path-wise, we have

$$\begin{aligned}\mu \cdot \texttt{ascore}_1(\mathcal{T}, &Q_{\tau-1}, A) + (1-\mu) \cdot \texttt{ascore}_2(\mathcal{T}, Q_{\tau-1}, A)\\ &= \mu \cdot \texttt{ascore}_1(\mathcal{T}_{Q_{\tau-1}}, Q_{\tau-1}, A) + (1-\mu) \cdot \texttt{ascore}_2(\mathcal{T}_{Q_{\tau-1}}, Q_{\tau-1}, A)\\ &= \mu \cdot \texttt{ascore}_1(\mathcal{T}', Q_{\tau-1}, A) + (1-\mu) \cdot \texttt{ascore}_2(\mathcal{T}', Q_{\tau-1}, A).\end{aligned}$$

for all actions $A \in \texttt{actions}(\mathcal{T}_{Q_{\tau-1}}, Q_{\tau-1})$. Therefore Algorithm 1 and $\mathcal{A}'$ choose the same action at node Q_{t-1}, that is, $A_{\tau-1} = A'_{\tau-1}$. Finally, since $\texttt{children}$ is path-wise, we have

$$\texttt{children}(\mathcal{T}, Q_{\tau-1}, A_{\tau-1}) = \texttt{children}(\mathcal{T}_{Q_{\tau-1}}, Q_{\tau-1}, A_{\tau-1}) = \texttt{children}(\mathcal{T}', Q_{\tau-1}, A_{\tau-1}).$$

Since $Q_\tau \in \texttt{children}(\mathcal{T}, Q_{\tau-1}, A_{\tau-1})$, this is a contradiction, which completes the proof. ◀

We use the following generalization of Claim 3.4 of Balcan et al. [7] that shows the behavior of $\mathcal{A}'$ is piecewise constant. While their argument only applies to single-variable branching, our key insight is that the same reasoning can be readily adapted to handle any actions (including general branching constraints and cutting planes). The structure of our proof (which we defer to the appendix) is identical, but is modified to work in our more general setting. This style of analysis is similar in spirit to [34].

▶ **Lemma 3.** *Let $\texttt{ascore}_1$ and $\texttt{ascore}_2$ be two path-wise action-selection scores. Fix the input root node Q. There are $T \leq k^{\Delta(\Delta-1)/2} b^\Delta$ subintervals $I_1, \dots, I_T$ partitioning $[0,1]$ where for any subinterval I_t, the action-selection score $\mu \cdot \texttt{ascore}_1 + (1-\mu) \cdot \texttt{ascore}_2$ results in the same tree built by $\mathcal{A}'$ for all $\mu \in I_t$, no matter what node selection policy is used.*

We now prove our main structural result for Algorithm 1.

▶ **Lemma 4.** *Let $\texttt{ascore}_1$ and $\texttt{ascore}_2$ be path-wise action-selection scores and let $\texttt{nscore}_1$ and $\texttt{nscore}_2$ be path-wise node-selection scores. Fix the input root node Q. There are $T \leq k^{\Delta(9+\Delta)} b^\Delta$ rectangles partitioning $[0,1]^2$ such that for any rectangle R_t, the node-selection score $\lambda \cdot \texttt{nscore}_1 + (1-\lambda) \cdot \texttt{nscore}_2$ and the action-selection score $\mu \cdot \texttt{ascore}_1 + (1-\mu) \cdot \texttt{ascore}_2$ result in the same tree built by Algorithm 1 for all $(\mu, \lambda) \in R_t$.*

Proof. By Lemma 3, there is a partition of $[0,1]$ into subintervals $I_1 \cup \cdots \cup I_T$ such that for all μ within a given subinterval, the tree built by $\mathcal{A}'$ is invariant (independent of the node-selection score). Fix a subinterval I_t of this partition. Let $\mathcal{T}$ denote the tree built by Algorithm 1. For each node $Q \in \mathcal{T}$, let $\mathcal{T}_Q$ denote the path from the root to Q in $\mathcal{T}$. Since $\texttt{nscore}_1$ is path-wise, for any tree $\mathcal{T}'$ containing $\mathcal{T}_Q$ as a rooted path, $\texttt{nscore}_1(\mathcal{T}', Q) = \texttt{nscore}_1(\mathcal{T}_Q, Q)$. The same holds for $\texttt{nscore}_2$. For every pair of nodes $Q_1, Q_2 \in \mathcal{T}$, let $\lambda(Q_1, Q_2) \in [0,1]$ denote the unique solution to

$$\begin{aligned}\lambda \cdot \texttt{nscore}_1(\mathcal{T}_{Q_1}, Q_1) + (1-\lambda) \cdot \texttt{nscore}_2(\mathcal{T}_{Q_1}, Q_1)\\ = \lambda \cdot \texttt{nscore}_1(\mathcal{T}_{Q_2}, Q_2) + (1-\lambda) \cdot \texttt{nscore}_2(\mathcal{T}_{Q_2}, Q_2),\end{aligned}$$

if it exists (if there are either (1) no solutions or (2) infinitely many solutions, set $\lambda(Q_1, Q_2) = 0$). The thresholds $\lambda(Q_1, Q_2)$ for every pair of nodes $Q_1, Q_2 \in \mathcal{T}$ partition $[0,1]$ into subintervals such that for all λ within a given subinterval, the total order over the nodes of $\mathcal{T}$ induced by $\texttt{nscore}_\lambda$ is invariant. In particular, this means that the node selected by each iteration of Algorithm 1 is invariant. Let $J_1 \cup \cdots \cup J_S$ denote these subintervals induced by the thresholds over all subinterval $I_t \in \{I_1, \ldots, I_T\}$ established in Lemma 3.

We now show that this implies that the tree built by Algorithm 1 is invariant over all (μ, λ) within a given rectangle $I_t \times J_s$. Fix some rectangle $I_t \times J_s$. We proceed by induction on the iterations (of the while loop) of Algorithm 1. For the base case (iteration 0, before entering the while loop), the tree consists of only the root, so the hypothesis trivially holds. Now, suppose the statement holds up until the jth iteration, for some j. We analyze each line of Algorithm 1 to show that the behavior of the $j+1$st iteration is independent of $(\mu, \lambda) \in I_t \times J_s$. First, since J_s determines the node selected at each iteration (as argued above), the node selected on the $j+1$st iteration (line 3) is fixed, independent of $(\mu, \lambda) \in I_t \times J_s$. Denote this node by Q. Thus, whether $\texttt{depth}(Q) = \Delta$ is independent of $(\mu, \lambda) \in I_t \times J_s$, and similarly whether $\texttt{fathom}(\mathcal{T}, Q, \texttt{None}) = \texttt{true}$ is independent of $(\mu, \lambda) \in I_t \times J_s$ (line 4). This implies that whether or not Q is fathomed at this stage is independent of $(\mu, \lambda) \in I_t \times J_s$. If Q was fathomed, we are done. Otherwise, we argue that the action selected at line 7 is invariant over $(\mu, \lambda) \in I_t \times J_s$. By Lemma 3, $\mathcal{A}'$ builds the same tree for all $\mu \in I_t$. Let $\mathcal{T}_Q$ denote the path from the root to Q in this tree. By Lemma 2, $\mathcal{T}_Q$ is the path from the root to Q in the tree built by Algorithm 1 as well. The action selected at Q by $\mathcal{A}'$ is invariant over $\mu \in I_t$ (by Lemma 3). Therefore, since $\texttt{actions}$, $\texttt{ascore}_1$, and $\texttt{ascore}_2$ are path-wise, the action A selected by Algorithm 1 at Q is invariant over $\mu \in I_t$. Finally, $\texttt{fathom}(\mathcal{T}, Q, A)$ and $\texttt{children}(\mathcal{T}, Q, A)$ are completely determined, so the execution of the remaining conditional statement (line 8 to line 13) is invariant over $(\mu, \lambda) \in I_t \times J_s$. Thus, the entire iteration of Algorithm 1 is invariant over $(\mu, \lambda) \in I_t \times J_s$, which completes the induction.

Finally, we count the total number of rectangles in our partition of $[0,1]^2$. For each interval I_t in the partition established in Lemma 3, we obtained a partition of $I_t \times [0,1]$ into rectangles induced by at most $\binom{|\mathcal{T}|}{2}$ thresholds, which consists of at most at most

$$1 + \binom{(k^{\Delta+1}-1)/(k-1)}{2} \le 1 + \left(\frac{k^{\Delta+1}-1}{k-1}\right)^2 \le k^{5\Delta}$$

subintervals. Accounting for every interval $I_t \in \{I_1, \ldots, I_T\}$ in the partition from Lemma 3, we get a total of $Tk^{5\Delta} \le k^{\Delta(9+\Delta)/2} b^{\Delta}$ rectangles, as desired. ◀

We now derive generalization guarantees for the collection $\mathcal{F} = \{\texttt{cost}_{\mu,\lambda} : (\mu, \lambda) \in [0,1]^2\}$ where $\texttt{cost}$ is any tree-constant function, such as tree size. We do this by bounding the *pseudo-dimension* of $\mathcal{F}$, which is a combinatorial measure of intrinsic complexity of a class of real valued functions. The pseudo-dimension of $\mathcal{F}$, denoted by $\mathrm{Pdim}(\mathcal{F})$, is the largest positive integer N such that there exist N nodes $Q_1, \ldots, Q_N \in \mathcal{Q}$ and N thresholds $r_1, \ldots, r_N \in \mathbb{R}$ such that $|\{(\mathrm{sign}(f(Q_1) - r_1), \ldots, \mathrm{sign}(f(Q_N) - r_N)) : f \in \mathcal{F}\}| = 2^N$. A well-known result in learning theory [3] states that if functions in $\mathcal{F}$ have bounded range $[-H, H]$, then

$$N_{\mathcal{F}}(\varepsilon, \delta) = O\left(\frac{H^2}{\varepsilon^2}\left(\mathrm{Pdim}(\mathcal{F}) + \ln(1/\delta)\right)\right) \text{ and } \varepsilon_{\mathcal{F}}(N, \delta) = O\left(H\sqrt{\frac{\mathrm{Pdim}(\mathcal{F}) + \ln(1/\delta)}{N}}\right).$$

When each function in $\mathcal{F}$ maps to $\{0, 1\}$, the pseudo-dimension is more commonly referred to as the *VC dimension*.

Bounding the pseudo-dimension is a simple instantiation of the general framework provided by Balcan et al. [6] with the piecewise structure established in Lemma 4. Balcan et al.'s [6] main result gives pseudo-dimension bounds for families of piecewise structured functions in terms of the VC dimension of the class of 0/1 classifiers defining the boundaries of the functions, the number of classifiers defining the boundaries, and the pseudo-dimension of the family of functions when restricted to each piece. (Strictly, this result is in terms of the dual classes of the boundary and piece functions. However, since the dual class of all linear separators is the set of all linear separators, we omit this detail for simplicity.)

▶ **Theorem 5.** *Let $\mathtt{cost}(Q)$ be any tree-constant cost function, and let $\mathtt{cost}_{\mu,\lambda}(Q)$ be the cost of the tree built by Algorithm 1 on input root node Q using action-selection score parameterized by μ and node-selection score parameterized by λ. Then,* $\mathrm{Pdim}(\{\mathtt{cost}_{\mu,\lambda}\}) = O(\Delta^2 \log k + \Delta \log b)$.

Proof. By Lemma 4, there are at most $T = k^{\Delta(9+\Delta)} b^{\Delta}$ rectangles partitioning $[0, 1]^2$ such that for a fixed input node Q, $\mathtt{cost}_{\mu,\lambda}(Q)$ is constant over each rectangle as a function of μ, λ. These T rectangles can be defined by T thresholds on $[0, 1]$ corresponding to μ and T thresholds on $[0, 1]$ corresponding to λ. Thus, the T rectangles can be identified by $T^2 = k^{2\Delta(9+\Delta)} b^{2\Delta}$ linear separators in $\mathbb{R}^2$. The VC dimension of linear separators in $\mathbb{R}^2$ is $O(1)$. The pseudo-dimension of the set of constant functions is also $O(1)$. Plugging these quantities into the main theorem of Balcan et al. [6] yields the theorem statement. ◀

3.1 Multiple actions

Theorem 5 can be easily generalized to the case where there are multiple actions of different types taken at each node of Algorithm 1. Specifically, there are now d path-wise action-set functions $\mathtt{actions}_1, \ldots, \mathtt{actions}_d$, and at line 7 of Algorithm 1 we take one action of each type, that is, we select action $A_1 \in \mathtt{actions}_1(\mathcal{T}, Q)$, $A_2 \in \mathtt{actions}_2(\mathcal{T}, Q)$, and so on. The functions `fathom` and `children` then depend on all d actions taken at node Q. We assume that there are two scoring rules $\mathtt{ascore}_1^i$ and $\mathtt{ascore}_2^i$ for each action type $i = 1, \ldots, d$. Algorithm 1 can then be parameterized by $(\boldsymbol{\mu}, \lambda)$, where $\boldsymbol{\mu} \in \mathbb{R}^d$ is a vector of parameters controlling each action, so the ith action is selected to maximize $\mu_i \cdot \mathtt{ascore}_1^i + (1 - \mu_i) \cdot \mathtt{ascore}_2^i$. Then, as long as $d = O(1)$, we get the same pseudo-dimension bound. We assume b is a uniform upper bound on the size of $\mathtt{actions}_i$ for any i. The proof is nearly identical, and we defer it to the appendix (which also contains more details on the multiple-action setup).

▶ **Theorem 6.** *Let $\mathtt{cost}(Q)$ be any tree-constant cost function, and let $\mathtt{cost}_{\boldsymbol{\mu},\lambda}(Q)$ be the cost of the tree built by Algorithm 1 on input root node Q using action-selection scores parameterized by $\boldsymbol{\mu} \in \mathbb{R}^d$, where $d = O(1)$, and node-selection score parameterized by λ. Then,* $\mathrm{Pdim}(\{\mathtt{cost}_{\boldsymbol{\mu},\lambda}\}) = O(\Delta^2 \log k + \Delta \log b)$.

4 Branch-and-cut for integer programming

We now instantiate our main results with the three main components of the B&C algorithm: branching, cutting planes, and node selection, used to solve IPs $\max\{\boldsymbol{c}^T\boldsymbol{x} : A\boldsymbol{x} \le \boldsymbol{b}, \boldsymbol{x} \ge 0, \boldsymbol{x} \in \mathbb{Z}^n\}$ where $\boldsymbol{c} \in \mathbb{R}^n$, $A \in \mathbb{Z}^{m\times n}$, $\boldsymbol{b} \in \mathbb{Z}^m$. The function $\texttt{fathom}(\mathcal{T}, Q, A)$ outputs $\texttt{true}$ if after having taken action A the LP relaxation at Q is integral, infeasible, or worse than the best integral solution found so far in $\mathcal{T}$. The function $\texttt{children}(\mathcal{T}, Q, A)$ outputs the two subproblems generated by the branching procedure on the IP at Q after having taken action A. For simplicity we refer only to IPs, but everything in our discussion applies to mixed IPs as well. In our model of tree search, node selection is controlled by λ. Cutting planes and branching are types of actions and controlled by μ.

4.1 Branching

In this section, we provide guarantees for branching. Throughout this section we assume $\Delta = O(n)$, as is the case with single-variable branching.

4.1.1 Multivariable branching constraints

It is well known that allowing for more general generation of branching constraints can result in smaller B&C trees. Gilpin and Sandholm [17] studied multivariable branches of the form $\sum_{i\in S} \boldsymbol{x}[i] \le \left\lfloor \sum_{i\in S} \boldsymbol{x}^*_{\mathsf{LP}}[i] \right\rfloor$, $\sum_{i\in S} \boldsymbol{x}[i] \ge \left\lceil \sum_{i\in S} \boldsymbol{x}^*_{\mathsf{LP}}[i] \right\rceil$ where S is a subset of the integer variables such that $\sum_{i\in S} \boldsymbol{x}^*_{\mathsf{LP}}[i] \notin \mathbb{Z}$. Here, $\texttt{actions}(\mathcal{T}, Q) = 2^{[n]}$, so, $\mathrm{Pdim}(\{\texttt{cost}_{\mu,\lambda}\}) = O(n^2)$. So our sample complexity bound for multivariable branching constraints is, surprisingly, only a constant factor worse than the bound for single-variable branching constraints.

We give a simple example where B&C using only single variable branches builds a tree of exponential size, while a single branch on the entire set of variables at the root yields two infeasible subproblems (and a B&C tree of size 3).

▶ **Theorem 7.** *For any n, there is an IP with two constraints and n variables such that with only single variable branches, B&C builds a tree of size $2^{(n-1)/2}$, while with a suitable multivariable branch, B&C builds a tree of size three.*

Proof. Let n be an odd positive integer. Consider the infeasible IP $\max\{\sum_{i=1}^n x[i] : 2\sum_{i=1}^n x[i] = n, \boldsymbol{x} \in \{0,1\}^n\}$. Jeroslow [26] proved that with only single-variable branches, B&C builds a tree with $2^{(n-1)/2}$ nodes to determine infeasibility. However, with a suitable multivariable branch, B&C will build a tree of constant size. The optimal solution to the LP relaxation of the IP is attained when all variables are set to $1/2$. A multivariable branch on all n variables produces the two subproblems with constraints $\sum_{i=1}^n x[i] \le \lfloor n/2 \rfloor$ and $\sum_{i=1}^n x[i] \ge \lceil n/2 \rceil$, respectively. Since n is odd, $\lfloor n/2 \rfloor < n/2$ and $\lceil n/2 \rceil > n/2$, so the LP relaxations of both subproblems are infeasible. Thus, B&C builds a tree with three nodes. ◀

Yang et al. [45] provide more examples of situations where multivariable branching yields dramatic improvements in tree size over single variable branching. They also perform a computational evaluation of a few different strategies for generating multivariable branching constraints. Yang et al. [44] explore gradient-boosting for learning to mimic strong branching for multiple variables.

4.1.2 Branching on general disjunctions

Branching constraints can be even more general than multivariable branches. Given any integer vector $\boldsymbol{\pi} \in \mathbb{Z}^n$ and any integer $\pi_0 \in \mathbb{Z}$ (jointly referred to as a *disjunction*), the constraints $\boldsymbol{\pi}^T\boldsymbol{x} \le \pi_0$ or $\boldsymbol{\pi}^T\boldsymbol{x} \ge \pi_0 + 1$ represent a valid partition of the feasible region into subproblems. Owen and Mehrotra [35] ran the first experiments demonstrating that branching on general disjunctions can lead to significantly smaller tree sizes. Subsequent works have posed different heuristics to select disjunctions to branch on [14, 33].

In practice it is known that additional IP constraints should not have coefficients that are too large. If C is a bound on the magnitude of the coefficient of any disjunction, then $\texttt{actions}(\mathcal{T}, Q) = \{-C, \ldots, C\}^{n+1}$, so $\text{Pdim}(\{\texttt{cost}_{\mu,\lambda}\}) = O(n^2 \log C)$. Karamanov and Cornuéjols [28] conduct a computational evaluation of disjunctions derived from Gomory mixed-integer cuts. In this setting, $\texttt{actions}(\mathcal{T}, Q)$ is the set of m or fewer disjunctions corresponding to the m or fewer Gomory mixed-integer cuts derived from the simplex tableau from solving the LP relaxation of Q. In this case, $\text{Pdim}(\{\texttt{cost}_{\mu,\lambda}\}) = O(n^2 + n \log m)$.

4.2 Cutting planes

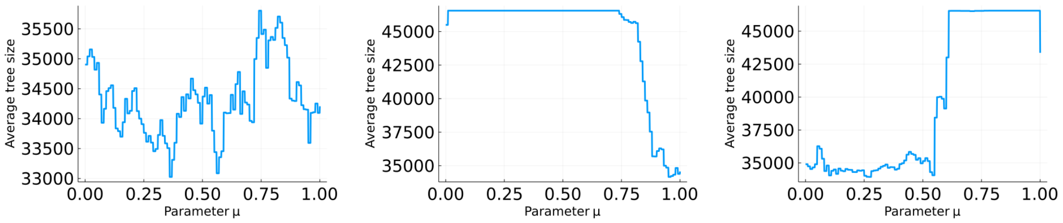
The action set can also correspond to cutting planes used to refine the feasible region of the IP at any stage of B&C. Here, $\texttt{actions}(\mathcal{T}, Q)$ is any set of cutting planes derived solely using the path from the root to the IP at Q. Examples include the set of Chvátal-Gomory (CG) derived from the simplex tableau [18], and various combinatorial families of cutting planes such as clique cuts, odd-hole cuts, and cover cuts. The set $\texttt{actions}(\mathcal{T}, Q)$ can also consist of sequences of cutting planes, representing adding several cutting planes to the IP in waves. For example, the set of all sequences of w CG cuts generated from the simplex tableau for an IP with m constraints has size at most m^w (regardless of whether the LP is resolved after each cut). The number of such cutting planes provided by the LP tableau at any node in the tree is at most $O(m + nw)$ (the original IP has m constraints, and after at most n branches there are an additional n branching constraints and at most nw cutting planes), which means that $|\texttt{actions}(\mathcal{T}, Q)| \leq O(m + nw)^w$. Thus, $\text{Pdim}(\{\texttt{cost}_{\mu,\lambda}\}) = O(n^2 + nw \log(m + nw))$.

We can also handle arbitrary CG cuts (not just ones from the LP tableau). Balcan et al. [9] proved that given an IP with feasible region $\{\boldsymbol{x} \in \mathbb{Z}^n : A\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \geq 0\}$, even though there are infinitely many CG cut parameters, there are effectively only $O(w2^w \|A\|_{1,1} + 2^w \|\boldsymbol{b}\|_1 + nw)^{1+mw}$ distinct sequences of cutting planes that w CG cut parameters can produce. At any node in the B&C tree, the number of constraints is at most $O(m + nw)$. So, on the domain of IPs with $\|A\|_{1,1} \leq \alpha$ and $\|\boldsymbol{b}\|_1 \leq \beta$, $|\texttt{actions}(\mathcal{T}, Q)| \leq O(w2^w\alpha + 2^w\beta + nw)^{1+w\cdot O(m+nw)}$. Thus, $\text{Pdim}(\{\texttt{cost}_{\mu,\lambda}\}) = O(n^2 w^3 m \log(\alpha + \beta + n))$.

4.2.1 Experiments on cover cuts for the multiple knapsack problem

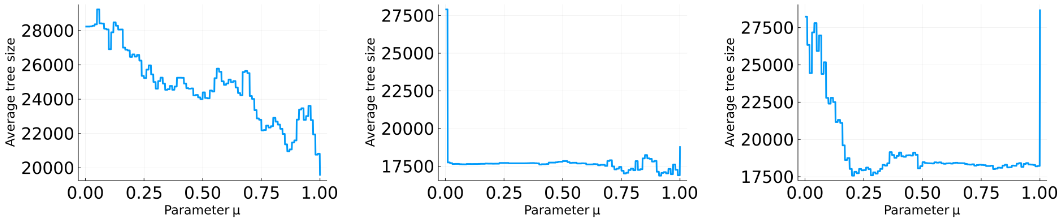
In this section, we demonstrate via experiments that tuning a convex combination of scoring rules to select cuts can lead to dramatically smaller branch-and-cut trees when done in a data-dependent manner. We study the classical NP-hard *multiple knapsack problem*: given a set N of items where each item $i \in N$ has a value $p_i \geq 0$ and a weight $w_i \geq 0$, and a set K of knapsacks where each knapsack $k \in K$ has a capacity $W_k \geq 0$, the goal is to find a feasible packing of the items into the knapsacks of maximum value. We assume, without loss of generality, that the items are labeled in descending order of weight, that is, $w_1 \geq w_2 \geq \cdots \geq w_{|N|}$. This problem can be formulated as the following binary IP:

$$\begin{array}{lll} \text{maximize} & \sum_{i\in N}\sum_{k\in K} p_i x_{k,i} & \\ \text{subject to} & \sum_{i\in N} w_i x_{k,i} \leq W_k & \forall\, k \in K \\ & \sum_{k\in K} x_{k,i} \leq 1 & \forall\, i \in N \\ & x_{k,i} \in \{0,1\} & \forall\, i \in N, k \in K \end{array}$$



(a) $\mu \cdot \texttt{E} + (1-\mu) \cdot \texttt{P}$. **(b)** $\mu \cdot \texttt{E} + (1-\mu) \cdot \texttt{D}$. **(c)** $\mu \cdot \texttt{D} + (1-\mu) \cdot \texttt{P}$.

Figure 1 Chvátal distribution with 35 items and 2 knapsacks.



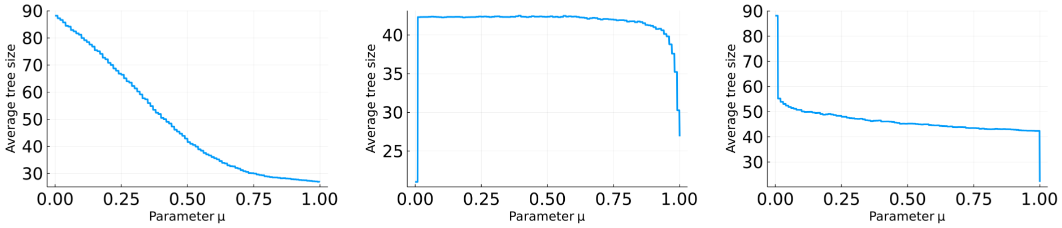
(a) $\mu \cdot \texttt{E} + (1-\mu) \cdot \texttt{P}$. **(b)** $\mu \cdot \texttt{E} + (1-\mu) \cdot \texttt{D}$. **(c)** $\mu \cdot \texttt{D} + (1-\mu) \cdot \texttt{P}$.

Figure 2 Chvátal distribution with 35 items and 3 knapsacks.

A subset $C \subseteq N$ of items is called a *cover* for knapsack $k \in K$ if $\sum_{i \in C} w_i > W_k$. If C is a cover, no feasible solution can have $x_{k,i} = 1$ for all $i \in C$, so $\sum_{i \in C} x_{k,i} \leq |C| - 1$ is a valid constraint – called a *cover cut*. When C is minimal (that is, $C \setminus \{i\}$ is not a cover for every $i \in C$), such cover cuts help tighten the knapsack IP by cutting off fractional LP solutions. We generate (a subset of all) cover cuts for each knapsack k as follows: for each $i \in N$, let $j > i$ be minimal such that $C = \{i, i+1, \ldots, j\}$ is a cover for k (if such a j exists). Since $w_i \geq w_j$ for $j > i$, C is a minimal cover, and moreover the *extended cover cut* $\sum_{i=1}^{j} x_i \leq |C| - 1$ is valid and dominates the minimal cover cut $\sum_{i \in C} x_i \leq |C| - 1$. Extended cover cuts generated from minimal covers are known to be facet defining for the integer hull under certain natural conditions [12].

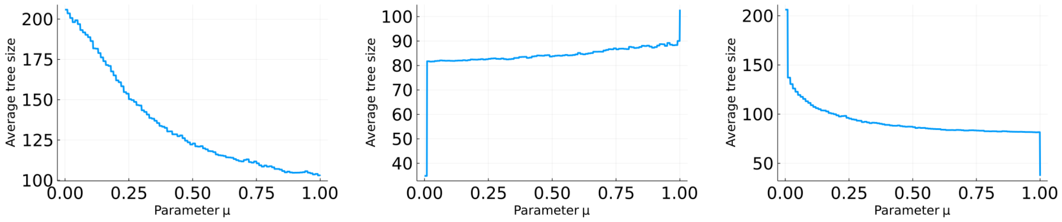
We investigate the relationship between three scoring rules for cutting planes. The first is *efficacy* (E), which is the perpendicular distance from the current LP solution to the cutting plane. The second is *parallelism* (P), which measures the angle between the objective and the normal vector to the cutting plane. The third is *directed cutoff* (D), which is the distance from the current LP solution to the cutting plane along the direction of the line segment connecting the LP solution to the current best incumbent integer solution. More details, including explicit formulas, can be found in [9] and references therein.

We consider two specific instances of the multiple knapsack problem, which are loosely based on a class of knapsack problems introduced by Chvátal that are difficult to solve with vanilla branch-and-bound [11, 45]. In the first, $p_i = w_i$ for all $i \in N$, and $W_k = \lfloor (\sum_{i \in N} w_i)/2|K| \rfloor + (k-1)$ for each $k = 1, \ldots, |K|$. In the second, $p_i = w_{|N|-i+1}$, so the most valuable item is the lightest and the least valuable item is the heaviest, and W_k is defined as in the first type. We call the first class of problems *Chvátal instances* and the second class *reverse Chvátal instances*. For a given N, K, we generate (reverse) Chvátal instances by drawing each weight independently as $w_i = \lfloor z_i \rfloor$, where $z_i \sim \mathcal{N}(50, 2)$, and sorting the items by weight in descending order.



(a) $\mu \cdot \mathtt{E} + (1-\mu) \cdot \mathtt{P}$. **(b)** $\mu \cdot \mathtt{E} + (1-\mu) \cdot \mathtt{D}$. **(c)** $\mu \cdot \mathtt{D} + (1-\mu) \cdot \mathtt{P}$.

Figure 3 Reverse Chvátal distribution with 100 items and 10 knapsacks.



(a) $\mu \cdot \mathtt{E} + (1-\mu) \cdot \mathtt{P}$. **(b)** $\mu \cdot \mathtt{E} + (1-\mu) \cdot \mathtt{D}$. **(c)** $\mu \cdot \mathtt{D} + (1-\mu) \cdot \mathtt{P}$.

Figure 4 Reverse Chvátal distribution with 100 items and 15 knapsacks.

In our experiments, we add (whenever possible) two extended cover cuts obtained in the aforementioned manner at every node of the B&C tree. The two cuts chosen are the two with the highest score $\mu \cdot \mathtt{ascore}_1 + (1-\mu) \cdot \mathtt{ascore}_2$ among all extended cover cuts that are violated by the current LP optimum, where $\mathtt{ascore}_1, \mathtt{ascore}_2 \in \{\mathtt{E}, \mathtt{D}, \mathtt{P}\}$. Figures 1-4 display the average tree size over 1000 samples for different Chvátal and reverse Chvátal distributions as a function of μ, where the domain $[0,1]$ of μ is discretized in increments of 0.01. We ran our experiments using the Python API of CPLEX 12.10 with default cut generation turned off. All other aspects of B&C (e.g. variable and node selection) are controlled by the default settings of CPLEX. The key takeaway of our plots is that tuning a convex combination of scoring rules can lead to significant savings in B&C tree size, and that this tuning must be done with the IP distribution in mind. No single parameter produces small trees for all the distributions we considered, and in fact a μ that minimizes tree size for one distribution can result in the largest trees for another (as in Figures 2b and 4b, for example). Furthermore, many of the plots display discernible trends (and in some cases are quite smooth), suggesting that the number of samples required to avoid overfitting in practice can be significantly smaller than our theoretical bounds.

4.3 Improved bounds for branch-and-cut

To allow node selection, branching, and cutting-plane selection to be tuned simultaneously, we apply Theorem 6. This allows us to bound the pseudo-dimension of the family of functions $\{\mathtt{cost}_{\mu_1,\mu_2,\lambda}\}$, where μ_1 controls branching, μ_2 controls cutting-plane selection, and λ controls node selection. Let $\mathtt{actions}_1(\mathcal{T}, Q)$ denote the set of branching actions available at Q, and let $\mathtt{actions}_2(\mathcal{T}, Q)$ denote the set of cutting planes available at Q. Let $b_1, b_2 \in \mathbb{N}$ be such that $\mathtt{actions}_1(\mathcal{T}, Q) \leq b_1$ and $\mathtt{actions}_2(\mathcal{T}, Q) \leq b_2$ for all $\mathcal{T}$ and all $Q \in \mathcal{T}$. Fix two branching scores $\mathtt{ascore}^1_1, \mathtt{ascore}^1_2$, fix two cutting-plane selection scores $\mathtt{ascore}^2_1, \mathtt{ascore}^2_2$, and fix two node-selection scores $\mathtt{nscore}_1, \mathtt{nscore}_2$.

▶ **Theorem 8.** *Let* $\texttt{cost}(Q)$ *be any tree-constant cost function, and let* $\texttt{cost}_{\mu_1,\mu_2,\lambda}$ *be the cost of the tree built by B&C using branching score* $\mu_1 \cdot \texttt{ascore}_1^1 + (1-\mu_1) \cdot \texttt{ascore}_2^1$*, cutting-plane selection score* $\mu_2 \cdot \texttt{ascore}_1^2 + (1-\mu_2) \cdot \texttt{ascore}_2^2$*, and node-selection score* $\lambda \cdot \texttt{nscore}_1 + (1-\lambda) \cdot \texttt{nscore}_2$*. Then, with* $\Delta = O(n)$*,* $\text{Pdim}(\{\texttt{cost}_{\mu_1,\mu_2,\lambda}\}) = O(n^2 + n\log(b_1+b_2))$*.*

4.3.1 Comparison to existing bounds

Balcan et al. [9] give a pseudo-dimension bound for tree search with a linear dependence on a cap κ on the number of nodes allowed in any tree. Their pseudo-dimension bound in our setting is $\text{Pdim}(\{\texttt{cost}_{\mu_1,\mu_2,\lambda}\}) = O(\kappa\log\kappa + \kappa\log b_1 + \kappa\log b_2)$. While κ is treated as a constant, it can be a prohibitively large quantity. In fact, without explicitly enforcing a limit on the number of nodes expanded by B&C, Balcan et al. [9] obtain a pseudo-dimension bound of $O(2^n(\log b_1 + \log b_2))$. Balcan et al. [7] use the path-wise property to prove that $\text{Pdim}(\{\texttt{cost}_\mu\}) = O(n^2)$ for single-variable branching, but for the case where branching is the only tunable component of B&C (and node selection is fixed).

5 Conclusions and future research

We presented a general model of tree search and proved sample complexity guarantees for this model that improve and generalize upon the recent sample complexity theory for configuring branch-and-cut. There are many interesting and open directions for future research. One compelling open question is to obtain pseudo-dimension bounds when action sets are infinite. Balcan et al. [9] alluded to this question in the case of cutting planes, and neither the techniques of their work nor the techniques of the present work can handle, for example, important infinite cutting-plane families such as the class of Gomory mixed-integer cuts, or the infinitely many valid disjunctions that could be branched on. Beyond integer programming, our model of tree search could potentially be applied to completely different problem domains that exhibit tree structure. Another direction is to extend our results to convex combinations of $\ell > 2$ scoring rules $\mu_1\texttt{score}_1 + \ldots \mu_\ell\texttt{score}_\ell$, as Balcan et al. [7] do in the special case of single-variable branching. However, their pseudo-dimension bound grows exponentially in the number of variables n in that special case; developing techniques that lead to a polynomial dependence on n remains a challenging open question.

References

1 Tobias Achterberg. *Constraint Integer Programming.* PhD thesis, Technische Universität Berlin, 2007.

2 Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29(1):185–195, 2017.

3 Martin Anthony and Peter Bartlett. *Neural Network Learning: Theoretical Foundations.* Cambridge University Press, 2009.

4 Egon Balas, Sebastián Ceria, and Gérard Cornuéjols. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42(9):1229–1246, 1996.

5 Maria-Florina Balcan. Data-driven algorithm design. In Tim Roughgarden, editor, *Beyond Worst Case Analysis of Algorithms.* Cambridge University Press, 2020.

6 Maria-Florina Balcan, Dan DeBlasio, Travis Dick, Carl Kingsford, Tuomas Sandholm, and Ellen Vitercik. How much data is sufficient to learn high-performing algorithms? Generalization guarantees for data-driven algorithm design. In *Annual Symposium on Theory of Computing (STOC)*, 2021.

7 Maria-Florina Balcan, Travis Dick, Tuomas Sandholm, and Ellen Vitercik. Learning to branch. In *International Conference on Machine Learning (ICML)*, 2018.

8 Maria-Florina Balcan, Vaishnavh Nagarajan, Ellen Vitercik, and Colin White. Learning-theoretic foundations of algorithm configuration for combinatorial partitioning problems. In *Conference on Learning Theory (COLT)*, 2017.

9 Maria-Florina Balcan, Siddharth Prasad, Tuomas Sandholm, and Ellen Vitercik. Sample complexity of tree search configuration: Cutting planes and beyond. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

10 Antonia Chmiela, Elias B Khalil, Ambros Gleixner, Andrea Lodi, and Sebastian Pokutta. Learning to schedule heuristics in branch-and-bound. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

11 Vasek Chvátal. Hard knapsack problems. *Operations Research*, 28(6):1402–1411, 1980.

12 Michele Conforti, Gérard Cornuéjols, Giacomo Zambelli, et al. *Integer programming*, volume 271. Springer, 2014.

13 Giovanni Di Liberto, Serdar Kadioglu, Kevin Leo, and Yuri Malitsky. Dash: Dynamic approach for switching heuristics. *European Journal of Operational Research*, 248(3):943–953, 2016.

14 Matteo Fischetti and Andrea Lodi. Local branching. *Mathematical Programming*, 98:23–47, 2002.

15 Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, Gregor Hendel, Christopher Hojny, Thorsten Koch, Pierre Le Bodic, Stephen J. Maher, Frederic Matter, Matthias Miltenberger, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Christine Tawfik, Stefan Vigerske, Fabian Wegscheider, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 7.0. Technical report, Optimization Online, March 2020. URL: `http://www.optimization-online.org/DB_HTML/2020/03/7705.html`.

16 Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 15554–15566, 2019.

17 Andrew Gilpin and Tuomas Sandholm. Information-theoretic approaches to branching in search. *Discrete Optimization*, 8(2):147–159, 2011. Early version in IJCAI-07.

18 Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1958.

19 Prateek Gupta, Maxime Gasse, Elias B Khalil, M Pawan Kumar, Andrea Lodi, and Yoshua Bengio. Hybrid models for learning to branch. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

20 Rishi Gupta and Tim Roughgarden. A PAC approach to application-specific algorithm selection. *SIAM Journal on Computing*, 46(3):992–1017, 2017.

21 He He, Hal Daume III, and Jason M Eisner. Learning to search in branch and bound algorithms. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2014.

22 Eric Horvitz, Yongshao Ruan, Carla Gomez, Henry Kautz, Bart Selman, and Max Chickering. A Bayesian approach to tackling hard computational problems. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2001.

23 Zeren Huang, Kerong Wang, Furui Liu, Hui-ling Zhen, Weinan Zhang, Mingxuan Yuan, Jianye Hao, Yong Yu, and Jun Wang. Learning to select cuts for efficient mixed-integer programming. *arXiv preprint*, 2021. `arXiv:2105.13645`.

24 Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization (LION)*, pages 507–523, 2011.

25 Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.

26 Robert G Jeroslow. Trivial integer programs unsolvable by branch-and-bound. *Mathematical Programming*, 6(1):105–109, 1974.

27 Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC—instance-specific algorithm configuration. In *European Conference on Artificial Intelligence (ECAI)*, 2010.

28 Miroslav Karamanov and Gérard Cornuéjols. Branching on general disjunctions. *Mathematical Programming*, 128(1-2):403–436, 2011.

29 Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *AAAI Conference on Artificial Intelligence*, 2016.

30 Elias Khalil, Bistra Dilkina, George Nemhauser, Shabbir Ahmed, and Yufen Shao. Learning to run heuristics in tree search. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

31 Robert Kleinberg, Kevin Leyton-Brown, and Brendan Lucier. Efficiency through procrastination: Approximately optimal algorithm configuration with runtime guarantees. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

32 Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4):1–52, 2009.

33 Ashutosh Mahajan and Theodore K Ralphs. Experiments with branching using general disjunctions. In *Operations Research and Cyber-Infrastructure*, pages 101–118. Springer, 2009.

34 Nimrod Megiddo. Combinatorial optimization with rational objective functions. *Mathematics of Operations Research*, pages 414–424, 1979.

35 Jonathan H. Owen and Sanjay Mehrotra. Experimental results on using general disjunctions in branch-and-bound for general-integer linear programs. *Computational Optimization and Applications*, 20(2):159–170, November 2001.

36 Ashish Sabharwal, Horst Samulowitz, and Chandra Reddy. Guiding combinatorial optimization with UCT. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer, 2012.

37 Tuomas Sandholm. Very-large-scale generalized combinatorial multi-attribute auctions: Lessons from conducting $60 billion of sourcing. In Zvika Neeman, Alvin Roth, and Nir Vulkan, editors, *Handbook of Market Design*. Oxford University Press, 2013.

38 Jialin Song, Ravi Lanka, Yisong Yue, and Bistra Dilkina. A general large neighborhood search framework for solving integer programs. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

39 Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement learning for integer programming: Learning to cut. *International Conference on Machine Learning (ICML)*, 2020.

40 Gellért Weisz, András György, and Csaba Szepesvári. LeapsAndBounds: A method for approximately optimal algorithm configuration. In *International Conference on Machine Learning (ICML)*, 2018.

41 Franz Wesselmann and Uwe Suhl. Implementing cutting plane management and selection techniques. Technical report, University of Paderborn, 2012.

42 Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32(1):565–606, 2008.

43 Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In *RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion at the International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.

44 Yu Yang, Natashia Boland, Bistra Dilkina, and Martin Savelsbergh. Learning generalized strong branching for set covering, set packing, and 0-1 knapsack problems. Technical report, Technical Report, 2020., 2020.

45 Yu Yang, Natashia Boland, and Martin Savelsbergh. Multivariable branching: A 0-1 knapsack problem case study. *INFORMS Journal on Computing*, 2021.

A Analysis of $\mathcal{A}'$

Proof of Lemma 3. Let $\mathcal{T}$ denote the tree built by $\mathcal{A}'$. For $i \in [\Delta]$, let $\mathcal{T}[i]$ denote the restriction of $\mathcal{T}$ to nodes of depth at most i. Let $\texttt{ascore}_\mu = \mu \cdot \texttt{ascore}_1 + (1-\mu) \cdot \texttt{ascore}_2$. We prove the lemma by induction on i. In particular, we show that for each $i \in [\Delta]$, there are $k^{i(i-1)/2}b^i$ subintervals partitioning $[0,1]$ such that $\mathcal{T}[i]$ is invariant over all μ within any given subinterval. Since $\mathcal{T}[\Delta] = \mathcal{T}$, this implies the lemma statement. The base case of $i = 1$ is trivial since $\mathcal{T}[1]$ consists of only the root.

Now, suppose the statement holds for some $i \in \{1, \dots, \Delta - 1\}$. That is, there are $T \leq k^{i(i-1)/2}b^i$ disjoint intervals $I_1 \cup \dots \cup I_T = [0,1]$ such that $\mathcal{T}[i]$ is invariant over all μ within any given subinterval (our inductive hypothesis). Fix one of these subintervals I_t. We subdivide I_t into subintervals such that $\mathcal{T}[i+1]$ is invariant within each one of these smaller subintervals. Let Q be any leaf of $\mathcal{T}[i]$, and for $\mu \in I_t$ let $\mathcal{T}_\mu$ denote the state of the tree using $\texttt{ascore}_\mu$ at the point that Q is selected. Since $i < \Delta$, Q is not fathomed at line 4, regardless of μ. Next, since $\texttt{actions}$ is path-wise, the actions available at Q depend only on the path $\mathcal{T}_Q$ from the root of $\mathcal{T}$ to Q, which, by the inductive hypothesis, is invariant over all $\mu \in I_t$. That is, $\texttt{actions}(\mathcal{T}_\mu, Q) = \texttt{actions}(\mathcal{T}_Q, Q)$ for all $\mu \in I_t$. Then, $\texttt{ascore}_\mu$ with parameter μ will select action $A \in \texttt{actions}(\mathcal{T}_Q, Q)$ if and only if

$$\begin{aligned} A &= \operatorname*{argmax}_{A_0 \in \texttt{actions}(\mathcal{T}_Q, Q)} \mu \cdot \texttt{ascore}_1(\mathcal{T}_\mu, Q, A_0) + (1-\mu) \cdot \texttt{ascore}_2(\mathcal{T}_\mu, Q, A_0) \\ &= \operatorname*{argmax}_{A_0 \in \texttt{actions}(\mathcal{T}_Q, Q)} \mu \cdot \texttt{ascore}_1(\mathcal{T}_Q, Q, A_0) + (1-\mu) \cdot \texttt{ascore}_2(\mathcal{T}_Q, Q, A_0), \end{aligned}$$

where the second equality follows from the fact that $\texttt{ascore}_1$ and $\texttt{ascore}_2$ are path-wise. Thus, for a fixed A_0, $\texttt{ascore}_\mu$ is linear in μ, so for each A_0 there is at most one subinterval of $[0,1]$ such that for all μ in that subinterval, A_0 maximizes $\texttt{ascore}_\mu$. Thus, each leaf of $\mathcal{T}[i]$ contributes at most b subintervals such that for μ within a given subinterval, the action selected at each leaf of $\mathcal{T}[i]$ is invariant. $\mathcal{T}[i]$ consists of at most k^i leaves, so this is a total of at most $k^i b$ subintervals. Now, since the action A selected at each leaf Q of $\mathcal{T}[i]$ is invariant, the set of children $\texttt{children}(\mathcal{T}_\mu, Q, A) = \texttt{children}(\mathcal{T}_Q, Q, A)$ of Q added to the tree is also invariant, using the fact that $\texttt{children}$ is path-wise. This shows that within every subinterval, $\mathcal{T}[i+1]$ is invariant. The total number of subintervals is, by the induction hypothesis, at most $k^{i(i-1)/2}b^i \cdot k^i b = k^{(i+1)i/2}b^{i+1}$, as desired. ◀

B Multiple actions

Let $\texttt{actions}_1, \dots, \texttt{actions}_d$ be path-wise. The multi-action version of Algorithm 1 is given by Algorithm 2. There are two scoring rules $\texttt{ascore}_1^i$ and $\texttt{ascore}_2^i$ for each action type $i \in [d]$. Algorithm 2 can then be parameterized by $(\boldsymbol{\mu}, \lambda)$, where $\boldsymbol{\mu} \in \mathbb{R}^d$ is a vector of parameters controlling each action: the ith action is selected to maximize $\mu_i \cdot \texttt{ascore}_1^i + (1-\mu_i) \cdot \texttt{ascore}_2^i$. As before, we assume there are $b, k \in \mathbb{N}$ such that $|\texttt{actions}_i(\mathcal{T}, Q)| \leq b$ for any i and any $Q \in \mathcal{T}$, and $|\texttt{children}(\mathcal{T}, Q, A_1, \dots, A_d)| \leq k$ for all $Q, A_1, \dots, A_d$.

Let $\mathcal{A}'$, as in the single-action setting, be Algorithm 2 with the evaluations of $\texttt{fathom}$ on line 4 and line 8 suppressed. Then, we may prove a slight generalization of lemma 3.

▶ **Lemma 9.** *Let $\texttt{ascore}_1^i$ and $\texttt{ascore}_2^i$ be two path-wise action-selection scores, for each $i \in \{1, \dots, d\}$. Fix the input root node Q. There are $T \leq k^{d\Delta(\Delta-1)/2}b^{d\Delta}$ boxes of the form $R_t = I_1 \times \dots \times I_d$ partitioning $[0,1]^d$ where for any box R_t, the action-selection scores $\mu_i \cdot \texttt{ascore}_1^i + (1-\mu_i) \cdot \texttt{ascore}_2^i$ results in the same tree built by $\mathcal{A}'$ for all $\boldsymbol{\mu} \in R_t$, no matter what node selection policy is used.*

Algorithm 2 Tree search with multiple actions.

```
Input: Root node $Q$, depth limit $\Delta$
1: Initialize $\mathcal{T} = Q$.
2: while $\mathcal{T}$ contains an unfathomed leaf do
3:     Select a leaf $Q$ of $\mathcal{T}$ that maximizes $\texttt{nscore}(\mathcal{T}, Q)$.
4:     if $\texttt{depth}(Q) = \Delta$ or $\texttt{fathom}(\mathcal{T}, Q, \texttt{None}, \ldots, \texttt{None})$ then
5:         Fathom $Q$.
6:     else
7:         For $i = 1, \ldots, d$, select $A_i \in \texttt{actions}_i(\mathcal{T}, Q)$ that maximizes $\texttt{ascore}_i(\mathcal{T}, Q, A_i)$.
8:         if $\texttt{fathom}(\mathcal{T}, Q, A_1, \ldots, A_d)$ then
9:             Fathom $Q$.
10:        else if $\texttt{children}(\mathcal{T}, Q, A_1, \ldots, A_d) = \emptyset$ then
11:            Fathom $Q$.
12:        else
13:            Add all nodes in $\texttt{children}(\mathcal{T}, Q, A_1, \ldots, A_d)$ to $\mathcal{T}$ as children of $Q$.
```

Proof. Let $\mathcal{T}$ denote the tree built by $\mathcal{A}'$. For $i \in [\Delta]$, let $\mathcal{T}[i]$ denote the restriction of $\mathcal{T}$ to nodes of depth at most i. Let $\texttt{ascore}^i_{\mu_i} = \mu_i \cdot \texttt{ascore}^i_1 + (1 - \mu_i) \cdot \texttt{ascore}^i_2$. We prove the lemma by induction on i. In particular, we show that for each $i \in [\Delta]$, there are $k^{di(i-1)/2} b^{di}$ boxes partitioning $[0,1]^d$ such that $\mathcal{T}[i]$ is invariant over all $\boldsymbol{\mu}$ within any given box. Since $\mathcal{T}[\Delta] = \mathcal{T}$, this implies the lemma statement. The base case of $i = 1$ is trivial since $\mathcal{T}[1]$ consists of only the root, regardless of $\boldsymbol{\mu} \in [0,1]^d$.

Now, suppose the statement holds for some $i \in \{1, \ldots, \Delta - 1\}$. That is, there are $T \le k^{di(i-1)/2} b^{di}$ disjoint boxes $R_1 \cup \cdots \cup I_R = [0,1]^d$ such that $\mathcal{T}[i]$ is invariant over all $\boldsymbol{\mu}$ within any given boxes (our inductive hypothesis). Fix one of these boxes R_t. We subdivide R_t into sub-boxes such that $\mathcal{T}[i+1]$ is invariant within each one of these smaller boxes. Let Q be any leaf of $\mathcal{T}[i]$, and for $\boldsymbol{\mu} \in R_t$ let $\mathcal{T}_{\boldsymbol{\mu}}$ denote the state of the tree using $\texttt{ascore}^i_{\mu_i}$ for each i at the point that Q is selected. Since $i < \Delta$, Q is not fathomed at line 4, regardless of $\boldsymbol{\mu}$. Next, since $\texttt{actions}_i$ is path-wise for each i, the actions available at Q depend only on the path $\mathcal{T}_Q$ from the root of $\mathcal{T}$ to Q, which, by the inductive hypothesis, is invariant over all $\boldsymbol{\mu} \in R_t$. That is, for all i $\texttt{actions}_i(\mathcal{T}_\mu, Q) = \texttt{actions}_i(\mathcal{T}_Q, Q)$ for all $\boldsymbol{\mu} \in R_t$. Then, $\texttt{ascore}^i_{\mu_i}$ will select action $A_i \in \texttt{actions}_i(\mathcal{T}_Q, Q)$ if and only if

$$
\begin{aligned}
A_i &= \operatorname*{argmax}_{A_0 \in \texttt{actions}_i(\mathcal{T}_Q, Q)} \mu \cdot \texttt{ascore}^i_1(\mathcal{T}_{\boldsymbol{\mu}}, Q, A_0) + (1 - \mu_i) \cdot \texttt{ascore}^i_2(\mathcal{T}_{\boldsymbol{\mu}}, Q, A_0) \\
&= \operatorname*{argmax}_{A_0 \in \texttt{actions}_i(\mathcal{T}_Q, Q)} \mu_i \cdot \texttt{ascore}^i_1(\mathcal{T}_Q, Q, A_0) + (1 - \mu_i) \cdot \texttt{ascore}^i_2(\mathcal{T}_Q, Q, A_0),
\end{aligned}
$$

where the second equality follows from the fact that $\texttt{ascore}^i_1$ and $\texttt{ascore}^i_2$ are path-wise. Thus, for a fixed A_0, $\texttt{ascore}^i_{\mu_i}$ is linear in μ_i, so for each A_0 there is at most one subinterval of $[0,1]$ such that for all μ_i in that subinterval, A_0 maximizes $\texttt{ascore}^i_{\mu_i}$. Thus, each leaf of $\mathcal{T}[i]$ contributes at most b subintervals such that for μ_i within a given subinterval, the action of type i selected at each leaf of $\mathcal{T}[i]$ is invariant. $\mathcal{T}[i]$ consists of at most k^i leaves, so this is a total of at most $k^i b$ subintervals. Writing $R_t = I_1 \times \cdots I_d$, we have established that for each i, there are $k^i b$ subintervals partitioning I_i into subintervals such that as μ_i varies over each subinterval, the action of type i selected at every leaf of $\mathcal{T}[i]$ is invariant. These subintervals partition R_t into at most $(k^i b)^d$ boxes. As before, since the actions selected at each leaf Q of $\mathcal{T}[i]$ are invariant, the set of children $\texttt{children}(\mathcal{T}_{\boldsymbol{\mu}}, Q, A_1, \ldots, A_d) = \texttt{children}(\mathcal{T}_Q, Q, A_1, \ldots, A_d)$ of

Q added to the tree is also invariant, using the fact that `children` is path-wise. Therefore, within every sub-box of R_t, $\mathcal{T}[i+1]$ is invariant. The total number of boxes over each possible R_t is, by the induction hypothesis, at most $k^{di(i-1)/2}b^{di} \cdot k^{di}b^d = k^{d(i+1)i/2}b^{d(i+1)}$. ◀

The proof of Lemma 2 is identical in the multi-action setting. The proof of Lemma 4 is also identical: here, we fix a box R in the partition established in Lemma 9, and get an identical partition of $R \times [0,1]$ such that the behavior of Algorithm 2 is invariant as λ varies in each subinterval of $[0,1]$. The number of boxes in the final partition of $[0,1]^{d+1}$ is $k^{d\Delta(\Delta-1)/2}b^{d\Delta} \cdot k^{5\Delta} \leq k^{d\Delta(9+\Delta)}b^{d\Delta}$. Our main pseudo-dimension bound for the multi-action setting follows from the same argument that exploits the framework of Balcan et al. [6].

▶ **Theorem 10.** *Let* $\texttt{cost}(Q)$ *be any tree-constant cost function, and let* $\texttt{cost}_{\boldsymbol{\mu},\lambda}(Q)$ *be the cost of the tree built by Algorithm 1 on input root node* Q *using action-selection scores parameterized by* $\boldsymbol{\mu} \in \mathbb{R}^d$*, where* $d = O(1)$*, and node-selection score parameterized by* λ*. Then,* $\mathrm{Pdim}(\{\texttt{cost}_{\boldsymbol{\mu},\lambda}\}) = O(d\Delta^2 \log k + d\Delta \log b)$*.*

When $d = O(1)$ we get the same pseudo-dimension bound as in the single-action setting: $\mathrm{Pdim}(\{\texttt{cost}_{\boldsymbol{\mu},\lambda}\}) = O(\Delta^2 \log k + \Delta \log b)$, which is the statement of Theorem 6.

Weisfeiler-Leman Invariant Promise Valued CSPs

Libor Barto
Department of Algebra, Faculty of Mathematics and Physics,
Charles University, Prague, Czechia

Silvia Butti
Department of Information and Communication Technologies,
Universitat Pompeu Fabra, Barcelona, Spain

Abstract

In a recent line of work, Butti and Dalmau have shown that a fixed-template Constraint Satisfaction Problem is solvable by a certain natural linear programming relaxation (equivalent to the basic linear programming relaxation) if and only if it is solvable on a certain distributed network, and this happens if and only if its set of Yes instances is closed under Weisfeiler-Leman equivalence. We generalize this result to the much broader framework of fixed-template Promise Valued Constraint Satisfaction Problems. Moreover, we show that two commonly used linear programming relaxations are no longer equivalent in this broader framework.

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

Keywords and phrases Promise Valued Constraint Satisfaction Problem, Linear programming relaxation, Distributed algorithms, Symmetric fractional polymorphisms, Color refinement algorithm

Digital Object Identifier 10.4230/LIPIcs.CP.2022.4

Related Version *Full Version*: https://arxiv.org/abs/2205.04805

Funding *Libor Barto*: Libor Barto has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (Grant Agreement No. 771005, CoCoSym).
Silvia Butti: Silvia Butti was supported by a MICCIN grant PID2019-109137GB-C22 and by a fellowship from "la Caixa" Foundation (ID 100010434). The fellowship code is LCF/BQ/DI18/11660056. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 713673.

Acknowledgements The authors are grateful to Victor Dalmau for his valuable comments.

1 Introduction

The Constraint Satisfaction Problem (CSP) is the problem of deciding whether there is an assignment of values from some domain A to a given set of variables, subject to constraints on the combinations of values which can be assigned simultaneously to certain specified subsets of variables; the allowed combinations of values are specified by relations on A.

Many important computational problems, including various versions of logical satisfiability, graph coloring, and systems of equations, can be obtained by fixing a finite domain and restricting the set of allowed relations [7, 13]. The restrictions can be specified by fixing a relational structure $\mathbf{A}$, called a template. The CSP over $\mathbf{A}$ is then the CSP restricted to instances that use only relations in $\mathbf{A}$. For example, if $\mathbf{A}$ consists of a single binary relation $R^{\mathbf{A}} \subseteq A^2$, an instance of the CSP over $\mathbf{A}$ is, e.g.,

$$R(x_1, x_2),\ R(x_3, x_1),\ R(x_2, x_4),\ R(x_3, x_3). \tag{1}$$

The goal is to decide whether there exists an assignment $h : \{x_1, x_2, \dots\} \to A$ that satisfies all the constraints, that is, $(h(x_1), h(x_2)) \in R^{\mathbf{A}}$, $(h(x_3), h(x_1)) \in R^{\mathbf{A}}$, etc. (see Section 2 for formal definitions). For instance, if $R^{\mathbf{A}}$ is the disequality relation $\neq$ on A, then the CSP over $\mathbf{A}$ is essentially the Graph $|A|$-Coloring Problem.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 4; pp. 4:1–4:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This paper deals with CSPs over fixed templates with finite domains. In particular, the phrase "a CSP" in the following discussion means the CSP over some template.

The (finite-domain, fixed-template) CSP has been a very active research area in the last 20 years, fueled by the tight connection between the complexity of a CSP and the polymorphisms of its template – these are multivariate functions on the domain that preserve all relations in the template (see [2]). The highlight in the area is the dichotomy theorem [3, 27]: every CSP is either solvable in polynomial time or NP-complete (assuming P is not NP); moreover, the polynomial cases are characterized by means of polymorphisms. Other major results include characterizations of applicability of fundamental algorithms, e.g., certain convex relaxations (see [11, 24]).

A natural linear programming relaxation, which is central in this paper, can be obtained by formulating a CSP instance as a feasibility problem for a zero-one integer program and then relaxing the requirement that each variable p is in $\{0, 1\}$ to $p \in [0, 1]$. In fact, there are two widely used relaxations of this form, the Basic Linear Programming (BLP) relaxation (see [14]) and a slightly stronger relaxation, which we denote by SA^1 to highlight its connection to the Sherali-Adams hierarchy [21] for CSPs (see [5]). The difference between the two relaxations is only in how they address repeated variables in a constraint. It turns out that both relaxations (correctly) decide the same CSPs [5] in the sense that, for any template $\mathbf{A}$, all instances of the CSP over $\mathbf{A}$ are decided by the SA^1 relaxation if and only if they are decided by BLP.[1] Moreover, this happens if and only if the template admits symmetric polymorphisms of all arities [16] (see also [1]).

The class of CSPs decided by BLP (SA^1) has reappeared recently in [4], where it was shown that it coincides with the class of CSPs which can be solved on a distributed network. The distributed set-up here is based on the DCSP framework of Yooko et al. [26]; informally, each constraint and each variable is controlled by an agent; the communication is only between a constraint and a variable that participates in it; and the agents are anonymous, they communicate in synchronous rounds, and they all run the same deterministic algorithm.

The papers [4, 5] contribute another interesting characterization, by means of an equivalence akin to the 1-dimensional Weisfeiler-Leman graph isomorphism test [17]. For two CSP instances $\mathbf{I}$, $\mathbf{J}$ we write $\mathbf{I} \equiv_1 \mathbf{J}$ if, very roughly, they cannot be distinguished by considering their local structure around variables (number and type of constraints they participate in, number and type of constraints their adjacent variables participate in, and so on). Now the equivalent conditions discussed above are also equivalent to the CSP being invariant under $\equiv_1$. Altogether, we have the following theorem, which witnesses the significance of this class of CSPs.

▶ **Theorem 1** ([4, 5, 16]). *The following are equivalent for the* CSP *over a finite structure* $\mathbf{A}$.

(i) *There exists a distributed algorithm that solves* $\mathrm{CSP}(\mathbf{A})$. *Moreover, in such a case, there is a polynomial-time distributed algorithm that solves* $\mathrm{CSP}(\mathbf{A})$.

(ii) *If two instances of* $\mathrm{CSP}(\mathbf{A})$ *are* $\equiv_1$*-equivalent, then they are either both* Yes *instances or both* No *instances.*

(iii) SA^1 *decides* $\mathrm{CSP}(\mathbf{A})$.

(iv) BLP *decides* $\mathrm{CSP}(\mathbf{A})$.

(v) $\mathbf{A}$ *has symmetric polymorphisms of every arity.*

Our main result generalizes Theorem 1 to a much broader setting, which we introduce next.

[1] We remark that in the literature the difference between the two relaxations is sometimes neglected, which occasionally leads to unjustified or slightly incorrect claims.

1.1 Promise Valued CSP

The framework of Valued CSP (VCSP) generalizes CSP as follows. Instead of relations we consider valued relations (also known as cost functions) – mappings that assign to tuples rational or positive infinite costs. Returning to the example above, $R^{\mathbf{A}}$ is now a mapping from A^2 to $\mathbb{Q} \cup \{\infty\}$ instead of a subset of A^2. The objective of the search version of the VCSP over $\mathbf{A}$ is to minimize a sum, e.g.,

$$R(x_1, x_2) + \ R(x_3, x_1) + \ R(x_2, x_4) + \ R(x_3, x_3), \tag{2}$$

that is, to find an assignment h such that $R^{\mathbf{A}}(h(x_1), h(x_2)) + R^{\mathbf{A}}(h(x_3), h(x_1)) + \dots$ is minimal. In the decision version, which we consider in this paper, the instance is such a sum together with a rational number τ and we aim to decide whether the minimum is at most τ.

Notice that (the decision version of) VCSP indeed generalizes CSP since relations can be modelled by $\{0, \infty\}$-valued relations. On the other hand, MaxCSP – where the aim is to maximize the number of satisfied constraints given a CSP instance – is exactly the VCSP over $\{0, 1\}$-valued relational structures. The VCSP framework also includes many problems of a mixed optimization and combinatorial nature, such as the Vertex Cover Problem (see [14]). The VCSP area is also well developed; for instance, the approach via an appropriate generalization of polymorphisms still works (see [14]), a dichotomy theorem is available [10], and the equivalence of (iv) and (v) in Theorem 1 can be lifted as well [11].

The more recent framework of Promise CSP (PCSP) generalizes CSP in a different direction. Here the relations are "crisp" but the template is a pair of structures $(\mathbf{A}, \mathbf{B})$ of the same signature. Intuitively, $R^{\mathbf{A}}$ is a "strict" form of R and $R^{\mathbf{B}}$ is its "relaxed" form. The PCSP over $(\mathbf{A}, \mathbf{B})$ is the problem of distinguishing instances solvable in $\mathbf{A}$ from those which are not solvable in $\mathbf{B}$. Note that the problem only makes sense if every instance solvable in $\mathbf{A}$ is also solvable in $\mathbf{B}$ (this is equivalent to $\mathbf{A}$ being homomorphic to $\mathbf{B}$). A well-known family of PCSP examples is the problem of distinguishing k-colorable graphs from those that are not even l-colorable for fixed $l \geq k$; see [1] for further examples. A complete complexity classification for PCSPs seems currently far away. Nevertheless, the algebraic approach via polymorphism works, and the equivalence of (iv) and (v) in Theorem 1 also remains valid [1].

Finally, the Promise Valued CSP (PVCSP) combines both generalizations. A template is a pair of valued structures of the same signature and the problem is, given a sum such as (2) and a rational number τ, to distinguish sums whose minimum computed in $\mathbf{A}$ is at most τ from those whose minimum in $\mathbf{B}$ is greater than τ. Again, the problem only makes sense if the template satisfies certain properties. An exact characterization of when this happens, Proposition 5, is one of the minor contributions of this paper.

We believe that the PVCSP is an extremely promising research direction for two reasons. First, it is very broad: it includes, for example, all constant factor approximation problems for MaxCSP (both the version where the aim is to approximately maximize the number of satisfied constraints, see Example 2; and the version where the aim is to approximately minimize the number of unsatisfied constraints). Second, the approach via generalized polymorphisms, so successful in the above special cases, is still available [9] (the work is not yet published). The only published work on PVCSP that we are aware of is [25] where the authors, among other results, generalize (iv) $\iff$ (v) in Theorem 1 to the PVCSP setting and even consider the more general infinite-domain case.

1.2 Contributions

Our main result, Theorem 8, lifts the equivalence of (i), (ii), and (iii) in Theorem 1 to the PVCSP framework.

The generalization of implication (i) ⇒ (ii) for connected input valued structures follows easily from the nature of the message passing systems we deal with. General, possibly disconnected input valued structures require an additional argument.[2] For the implication (ii) ⇒ (iii) we employ the approach of [5] and, in a sense, "decompose" a solution to the SA^1 relaxation of a PVCSP into three components. One component is a kind of morphism, called here a dual fractional homomorphism, which appeared before in the context of VCSPs with left-hand side (i.e., structural) restrictions [6].[3] The decomposition theorem, stated as Theorem 7, might be of independent interest. We also point out that our construction for this decomposition is much simpler than the construction used in [5] for the less general setting. The distributed algorithm that we design to prove (iii) ⇒ (i) is completely different from the one used for the CSP in [4]. The original algorithm relied on a deep theorem from the algebraic CSP theory [12] about the strength of a certain local propagation algorithm and designed a distributed version of that algorithm. This approach is no longer applicable, even in the (non-valued) PCSP setting. However, we show that a substantially more straightforward and simple idea of directly computing an adjusted form of SA^1 works even in the most general PVCSP framework.

Surprisingly, the implication (iii) ⇒ (iv) is no longer true for PVCSPs: in Example 9 we present a PVCSP template that is decided by SA^1 but not decided by BLP. The converse implication remains valid since SA^1 is a stronger relaxation than BLP.

Recall that the equivalence of (iv) and (v) still holds for PVCSPs [25]; we give a streamlined presentation of the proof using Proposition 5. We also mention, in Example 4, some (P)(V)CSPs that satisfy these conditions, and thus also satisfy the equivalent statements in the main result.

2 Preliminaries

For a tuple $\mathbf{a} \in A^k$, let $\mathbf{a}[i]$ denote the i^{th} entry of $\mathbf{a}$. We say that $\mathbf{a}$ has a *repetition* if there exist $i \neq j \in [k]$ such that $\mathbf{a}[i] = \mathbf{a}[j]$. We use double curly brackets $\{\{\dots\}\}$ to denote multisets. For a non-negative integer n, $n \cdot \{\{\dots\}\}$ stands for the multiset obtained by multiplying the multiplicity of each element in the original multiset by n. Slightly abusing the notation, the set and the multiset of entries of a tuple $\mathbf{a}$ is denoted by $\{\mathbf{a}\}$ and $\{\{\mathbf{a}\}\}$, respectively.

We denote by $\mathbb{Q}_{\geq 0}$ the set of non-negative rational numbers and by $\mathbb{Q}_\infty$ the set $\mathbb{Q} \cup \{\infty\}$, where ∞ is an additional symbol interpreted as a positive infinity. We set $0 \cdot \infty = 0$ and $c \cdot \infty = \infty$ for $c > 0$.

2.1 CSP and PCSP

We present the CSP and PCSP as homomorphism problems. The difference from the presentation in the introduction is merely formal.

[2] This subtle issue was not properly handled in [4]. The present paper thus also fills in a gap in the proof of Theorem 1.

[3] [6] uses the terminology "inverse fractional homomorphism", however we feel that "dual" might better fit the meaning of this concept.

A *signature* σ is a finite collection of relation symbols, each with an associated arity. We shall use $\mathrm{ar}(R)$ to denote the arity of a relation symbol R. Given a set A and a positive integer k, a k-ary *relation* on A is a subset of A^k. A *(relational) structure* $\mathbf{A}$ in the signature σ, or simply a σ-structure, consists of a finite set A called the *universe* of $\mathbf{A}$, and a relation $R^{\mathbf{A}}$ on A of arity $\mathrm{ar}(R)$ for each $R \in \sigma$. Notice that the universe of every structure in this paper is assumed to be finite. Two structures are *similar* if they have the same signature.

Let $\mathbf{I}$, $\mathbf{A}$ be σ-structures. A *homomorphism* from $\mathbf{I}$ to $\mathbf{A}$ is a map $h : I \to A$ such that for every $R \in \sigma$ and every tuple $\mathbf{v} \in R^{\mathbf{I}}$ it holds that $h(\mathbf{v}) \in R^{\mathbf{A}}$, where h is applied to $\mathbf{v}$ component-wise. If there exists a homomorphism from $\mathbf{I}$ to $\mathbf{A}$ we say that $\mathbf{I}$ is homomorphic to $\mathbf{A}$.

For a relational σ-structure $\mathbf{A}$, the *CSP over* $\mathbf{A}$, denoted $\mathrm{CSP}(\mathbf{A})$, is the problem of deciding whether an input σ-structure $\mathbf{I}$ is homomorphic to $\mathbf{A}$. The structure $\mathbf{A}$ is also referred to as a *template* in this context. The translation of the presented definition of $\mathrm{CSP}(\mathbf{A})$ to the formalism used in the introduction is given by defining the set of *constraints* $\mathcal{C}_{\mathbf{I}}$ as the set of formal expressions of the form $R(\mathbf{v})$ where $R \in \sigma$ and $\mathbf{v} \in R^{\mathbf{I}}$.

Given two σ-structures $\mathbf{A}$ and $\mathbf{B}$, the *Promise CSP over* $(\mathbf{A}, \mathbf{B})$, denoted $\mathrm{PCSP}(\mathbf{A}, \mathbf{B})$, is defined as follows: given a σ-structure $\mathbf{I}$, output Yes if $\mathbf{I}$ is homomorphic to $\mathbf{A}$, and output No if $\mathbf{I}$ is not homomorphic to $\mathbf{B}$.[4] This problem makes sense iff the sets of Yes and No instances are disjoint. It is easy to see that this happens exactly when $\mathbf{A}$ is homomorphic to $\mathbf{B}$. Such pairs of structures $(\mathbf{A}, \mathbf{B})$ are called *PCSP templates*.

2.2 PVCSP

We formalize PVCSPs in a similar way to PCSPs. The difference from the presentation in the introduction is slightly more substantial, as we shall briefly discuss later.

A k-ary *valued relation* on A is a function $R : A^k \to \mathbb{Q}_\infty$. A *valued σ-structure* $\mathbf{A}$ consists of a finite universe A, together with a valued relation $R^{\mathbf{A}}$ of arity $\mathrm{ar}(R)$ on A for each $R \in \sigma$. Valued structures are sometimes referred to as *general-valued* in the literature [11, 24] to emphasize that relations in $\mathbf{A}$ may take non-finite values. A σ-structure $\mathbf{A}$ is said to be *non-negative finite-valued* if for every $R \in \sigma$, the range of $R^{\mathbf{A}}$ is contained in $\mathbb{Q}_{\geq 0}$.

Let $\mathbf{I}$, $\mathbf{A}$ be valued σ-structures, where $\mathbf{I}$ is non-negative finite-valued. The *value* of a map $h : I \to A$ for $(\mathbf{I}, \mathbf{A})$, and the *optimum value* for $(\mathbf{I}, \mathbf{A})$ are given by

$$\mathrm{Val}(\mathbf{I}, \mathbf{A}, h) = \sum_{R \in \sigma} \sum_{\mathbf{v} \in I^{\mathrm{ar}(R)}} R^{\mathbf{I}}(\mathbf{v}) R^{\mathbf{A}}(h(\mathbf{v})), \qquad \mathrm{Opt}(\mathbf{I}, \mathbf{A}) = \min_{h : I \to A} \mathrm{Val}(\mathbf{I}, \mathbf{A}, h).$$

For two valued σ-structures $\mathbf{A}$ and $\mathbf{B}$, the *Promise Valued CSP over* $(\mathbf{A}, \mathbf{B})$ [9, 25], denoted $\mathrm{PVCSP}(\mathbf{A}, \mathbf{B})$, is defined as follows: given a pair $(\mathbf{I}, \tau)$, where $\mathbf{I}$ is a non-negative finite-valued σ-structure and $\tau \in \mathbb{Q}$ is a *threshold*, output Yes if $\mathrm{Opt}(\mathbf{I}, \mathbf{A}) \leq \tau$, and output No if $\mathrm{Opt}(\mathbf{I}, \mathbf{B}) > \tau$. We call $(\mathbf{A}, \mathbf{B})$ a *PVCSP template* if the sets of Yes and No instances are disjoint. We show in Proposition 5 that this least restrictive meaningful requirement on a PVCSP template coincides with the choice taken in [25].

Notice that the values of R have a different intended meaning in the template valued structures $\mathbf{A}$, $\mathbf{B}$ and in the input valued structure $\mathbf{I}$. For the template, $R^{\mathbf{A}}(\mathbf{a})$ and $R^{\mathbf{B}}(\mathbf{b})$ should be understood as the *cost* of $\mathbf{a}$ and $\mathbf{b}$: we wish an assignment h to map tuples of variables to tuples of domain elements that are as cheap as possible (and, in fact, $R^{\mathbf{A}}$ or $R^{\mathbf{B}}$

[4] We do not impose any requirements on the algorithm in the case that $\mathbf{I}$ is neither a Yes instance nor a No instance. Alternatively, we are *promised* that the input is a Yes instance or a No instance.

is often referred to as a *cost function*). On the other hand, $R^{\mathbf{I}}(\mathbf{v})$ is the *weight* of the tuple of variables $\mathbf{v}$: we need to be more concerned about heavy tuples of variables, while we may ignore the tuples of zero weight (recall that $0 \cdot \infty = 0$). As an example, observe that the PCSP over a pair of structures $(\mathbf{A}', \mathbf{B}')$ is essentially the same problem as the PVCSP over the pair of $\{0, \infty\}$-valued structures $(\mathbf{A}, \mathbf{B})$, where tuples in the latter template are given zero cost iff they belong to the corresponding relations in the former template; while to an instance $\mathbf{I}'$ of the PCSP corresponds a non-negative finite-valued structure $\mathbf{I}$ where the cost of a tuple is zero iff the tuple does *not* belong to the corresponding relation in $\mathbf{I}'$ (and costs of the remaining tuples are arbitrary positive rationals), together with any threshold $\tau \in \mathbb{Q}_{\geq 0}$.

For a PVCSP input valued σ-structure $\mathbf{I}$ we define the set of *constraints* $\mathcal{C}_{\mathbf{I}}$ as the set of formal expressions of the form $R(\mathbf{v})$ where $R \in \sigma$, $\mathbf{v} \in I^{\operatorname{ar}(R)}$, and $R^{\mathbf{I}}(\mathbf{v}) > 0$; the value $R^{\mathbf{I}}(\mathbf{v})$ is the *weight* of the constraint. This almost translates the presented definition of PVCSP to the version from the introduction: weights of constraints can be emulated by repeating constraints in (2) (and modifying the threshold τ if necessary). However, the repetition can cause an exponential blow up of the instance size. Nevertheless, this difference between the two formalisms is inessential for our purposes.

We say that a valued relation $R^{\mathbf{I}}$ has no repetitions if $R^{\mathbf{I}}(\mathbf{v}) = 0$ whenever $\mathbf{v}$ has a repetition. Similarly, we say that an input valued structure $\mathbf{I}$ has no repetitions if none of its valued relations has a repetition.

▶ **Example 2.** As mentioned in the introduction, the PVCSP framework can be used to model constant factor approximation problems for MaxCSP. More concretely, suppose that we want to find a c-approximation for $\operatorname{CSP}(\mathbf{A})$ for some (non-valued) σ-structure $\mathbf{A}$ and some $c < 1$. One can model this problem as $\operatorname{PVCSP}(\mathbf{A}', \mathbf{B}')$ where $A' = B' = A$ and for all $R \in \sigma$ and $\mathbf{a} \in A^{\operatorname{ar}(R)}$, $R^{\mathbf{A}'}(\mathbf{a}) = -1$ if $\mathbf{a} \in R^{\mathbf{A}}$ and $R^{\mathbf{A}'}(\mathbf{a}) = 0$ otherwise; and $R^{\mathbf{B}'}(\mathbf{a}) = \frac{1}{c} R^{\mathbf{A}'}(\mathbf{a})$. Given an instance $\mathbf{I}$ of $\operatorname{CSP}(\mathbf{A})$ and a parameter $0 < \beta \leq 1$, we turn it into an instance $(\mathbf{I}', -\beta m)$ of $\operatorname{PVCSP}(\mathbf{A}', \mathbf{B}')$ in a natural way, where $\mathbf{I}'$ is a 0-1 valued structure and m is the number of constraints in $\mathbf{I}'$. Then, $\operatorname{Opt}(\mathbf{I}', \mathbf{A}') \leq -\beta m$ if a β-fraction of all constraints of $\mathbf{I}$ can be satisfied in $\mathbf{A}$, and $\operatorname{Opt}(\mathbf{I}', \mathbf{B}') > -\beta m$ if not even a $c\beta$-fraction of the constraints of $\mathbf{I}$ can be satisfied in $\mathbf{A}$.

2.3 Linear programming relaxations

Given two valued σ-structures $\mathbf{I}$ and $\mathbf{A}$ where $\mathbf{I}$ is non-negative finite-valued, the systems of inequalities $\operatorname{BLP}(\mathbf{I}, \mathbf{A})$ and $\operatorname{SA}^1(\mathbf{I}, \mathbf{A})$ contain a variable $p_v(a)$ for every $v \in I$ and every $a \in A$, and a variable $p_{R(\mathbf{v})}(\mathbf{a})$ for every $R(\mathbf{v}) \in \mathcal{C}_{\mathbf{I}}$ and every $\mathbf{a} \in A^{\operatorname{ar}(R)}$. $\operatorname{BLP}(\mathbf{I}, \mathbf{A})$ is the following linear program.

$$\operatorname{Opt}^{\mathrm{BLP}}(\mathbf{I}, \mathbf{A}) := \min \sum_{R(\mathbf{v}) \in \mathcal{C}_{\mathbf{I}}} \sum_{\mathbf{a} \in A^{\operatorname{ar}(R)}} p_{R(\mathbf{v})}(\mathbf{a}) R^{\mathbf{I}}(\mathbf{v}) R^{\mathbf{A}}(\mathbf{a}) \tag{$\star$}$$

subject to:

$$p_{R(\mathbf{v})}(\mathbf{a}) \geq 0 \qquad R(\mathbf{v}) \in \mathcal{C}_{\mathbf{I}},\ \mathbf{a} \in A^{\operatorname{ar}(R)} \tag{3}$$

$$\sum_{a \in A} p_v(a) = 1 \qquad v \in I \tag{4}$$

$$p_v(a) = \sum_{\mathbf{a} \in A^{\operatorname{ar}(R)}, \mathbf{a}[i] = a} p_{R(\mathbf{v})}(\mathbf{a}) \qquad a \in A,\ R(\mathbf{v}) \in \mathcal{C}_{\mathbf{I}},\ i \in [\operatorname{ar}(R)] \text{ s.t. } \mathbf{v}[i] = v \tag{5}$$

$$p_{R(\mathbf{v})}(\mathbf{a}) = 0 \qquad R(\mathbf{v}) \in \mathcal{C}_{\mathbf{I}},\ \mathbf{a} \in A^{\operatorname{ar}(R)} \text{ s.t. } R^{\mathbf{A}}(\mathbf{a}) = \infty \tag{6}$$

As for the program $\mathrm{SA}^1(\mathbf{I},\mathbf{A})$, the objective function, denoted $\mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I},\mathbf{A})$, is given by the same objective function as in $\mathrm{BLP}(\mathbf{I},\mathbf{A})$. The variables are subject to all the constraints in $\mathrm{BLP}(\mathbf{I},\mathbf{A})$, but in addition, they are also subject to the following constraint.

$$\boxed{\begin{array}{ll} p_{R(\mathbf{v})}(\mathbf{a}) = 0 & R(\mathbf{v}) \in \mathcal{C}_{\mathbf{I}},\ \mathbf{a} \in A^{\mathrm{ar}(R)} \\ & \exists i,j \in [\mathrm{ar}(R)] \text{ such that } \mathbf{v}[i] = \mathbf{v}[j] \text{ and } \mathbf{a}[i] \neq \mathbf{a}[j] \end{array}} \tag{7}$$

Notice that in general $\mathrm{Opt}^{\mathrm{BLP}}(\mathbf{I},\mathbf{A}) \leq \mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I},\mathbf{A})$. Moreover, in the particular case where $\mathbf{I}$ has no repetitions, BLP and SA^1 are the same linear program and so $\mathrm{Opt}^{\mathrm{BLP}}(\mathbf{I},\mathbf{A}) = \mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I},\mathbf{A})$.

For a linear program $\mathrm{L} \in \{\mathrm{BLP}, \mathrm{SA}^1\}$ we say that $\mathrm{L}(\mathbf{I},\mathbf{A})$ is *feasible* if there exists a rational solution to the system $\mathrm{L}(\mathbf{I},\mathbf{A})$. Notice that then $(\star)$ makes sense since $R^{\mathbf{A}}(\mathbf{a}) = \infty$ implies $p_{R(\mathbf{v})} = 0$ and $0 \cdot \infty = 0$ (formally, one should skip these summands in $(\star)$). If the linear program is infeasible, then we set $\mathrm{Opt}^{\mathrm{L}}(\mathbf{I},\mathbf{A}) = \infty$.

The LP constraints (3)–(5) ensure that, for each $R(\mathbf{v}) \in \mathcal{C}_{\mathbf{I}}$, the values of $p_{R(\mathbf{v})}(\mathbf{a})$ form a probability distribution on $A^{\mathrm{ar}(R)}$ (which is additionally consistent with $p_v(a)$'s). The inner sum in $(\star)$ is equal to the expected "cost" of the constraint $R(\mathbf{v})$ with weight $R^{\mathbf{I}}(\mathbf{v})$ when $\mathbf{v}$ is evaluated according to this distribution. From this observation it is apparent that $\mathrm{Opt}^{\mathrm{L}}(\mathbf{I},\mathbf{A}) \leq \mathrm{Opt}(\mathbf{I},\mathbf{A})$. We say that L *decides* $\mathrm{PVCSP}(\mathbf{A},\mathbf{B})$ if, for every input structure $\mathbf{I}$, we have $\mathrm{Opt}(\mathbf{I},\mathbf{B}) \leq \mathrm{Opt}^{\mathrm{L}}(\mathbf{I},\mathbf{A})$. Note that in such a case the algorithm for $\mathrm{PVCSP}(\mathbf{A},\mathbf{B})$ that answers Yes iff $\mathrm{Opt}^{\mathrm{L}}(\mathbf{I},\mathbf{A}) \leq \tau$ (where τ is the input threshold) is correct, so the definition makes sense.[5]

2.4 Polymorphisms

An n-ary *polymorphism* of a pair of similar structures $(\mathbf{A},\mathbf{B})$ is an n-ary operation $f : A^n \to B$ such that for every relation symbol R in the signature of $\mathbf{A}$ and $\mathbf{B}$, the coordinate-wise application of f to any list of n tuples from $R^{\mathbf{A}}$ results in a tuple in $R^{\mathbf{B}}$. Note that a unary polymorphism of $(\mathbf{A},\mathbf{B})$ is just a homomorphism from $\mathbf{A}$ to $\mathbf{B}$. An n-ary operation $f : A^n \to B$ is said to be *symmetric* if for every $a_1,\dots,a_n \in A$ and every permutation ρ on $[n]$ we have that $f(a_1,\dots,a_n) = f(a_{\rho(1)},\dots,a_{\rho(n)})$.

An n-ary *fractional polymorphism* [25] of two valued σ-structures $(\mathbf{A},\mathbf{B})$ is a probability distribution ω on the set $B^{A^n} := \{f : A^n \to B\}$ such that for every $R \in \sigma$ and every list of n tuples $\mathbf{a}_1,\dots,\mathbf{a}_n \in A^{\mathrm{ar}(R)}$ we have that

$$\sum_{f \in B^{A^n}} \omega(f) R^{\mathbf{B}}(f(\mathbf{a}_1,\dots,\mathbf{a}_n)) \leq \frac{1}{n}\sum_{i=1}^{n} R^{\mathbf{A}}(\mathbf{a}_i)$$

where f is applied to $\mathbf{a}_1,\dots,\mathbf{a}_n \in A^{\mathrm{ar}(R)}$ component-wise.[6]

The *support* of ω is the set of functions $f : A^n \to B$ such that $\omega(f) > 0$. We say that ω is *symmetric* if every operation in its support if symmetric.

[5] We remark that in [5], the feasibility of the program $\mathrm{SA}^1(\mathbf{I},\mathbf{A})$ was alternatively phrased as the existence of a "fractional homomorphism" from $\mathbf{I}$ to $\mathbf{A}$, to stress that the linear system $\mathrm{SA}^1(\mathbf{I},\mathbf{A})$ is a (fractional) relaxation of homomorphism in the same way as the equivalence relation $\equiv_1$ defined below is a relaxation of isomorphism. Nonetheless, in this paper we avoid this terminology as it clashes with the notion of fractional homomorphism defined in Section 3 as a unary fractional polymorphism.

[6] We use here a simpler concept than fractional polymorphism as defined in [25], which will be sufficient for our purposes.

The following theorem was proved in [25]; we provide a somewhat streamlined argument in the spirit of [1] in Section 3.

▶ **Theorem 3.** *Let* $(\mathbf{A}, \mathbf{B})$ *be a promise valued template of signature* σ. *Then the following are equivalent.*

(iv) BLP *decides* $\mathrm{PVCSP}(\mathbf{A}, \mathbf{B})$*;*

(v) $(\mathbf{A}, \mathbf{B})$ *has symmetric fractional polymorphisms of every arity.*

▶ **Example 4.** A CSP that can be decided by BLP is e.g. the Horn-3-Sat, where the template has domain $\{\text{true}, \text{false}\}$ and two relations defined by $\neg x \vee \neg y \vee \neg z$ and $\neg x \vee \neg y \vee z$. A well-known class of templates with BLP-decidable VCSPs are those that contain only submodular valued relations (see [14]). Finally, the 2-approximation of the Vertex Cover problem [14] is a PVCSP decidable by BLP. In all the mentioned examples, it is not hard to find symmetric (fractional) polymorphisms of every arity.

2.5 Graph of an input, iterated degree, distributed model

We represent an input σ-structure $\mathbf{I}$ to a PVCSP as a labeled bipartite graph $\mathrm{G}_{\mathbf{I}}$, also known as the *factor graph* of $\mathbf{I}$ in the non-valued setting [8]. This representation allows us to define iterated degrees of variables and constraints as well as our distributed model.

$\mathrm{G}_{\mathbf{I}}$ has one vertex for each constraint $R(\mathbf{v}) \in \mathcal{C}_{\mathbf{I}}$, labeled (R, q) where $q = R^{\mathbf{I}}(\mathbf{v})$ (> 0), and one vertex for each variable, with empty label. Vertex $v \in I$ is adjacent to a vertex $R(\mathbf{v}) \in \mathcal{C}_{\mathbf{I}}$ if $v \in \{\mathbf{v}\}$; the edge is labeled $S = \{i : \mathbf{v}[i] = v\}$. The label of a vertex x is denoted ℓ_x, the label of an edge $\{x, y\}$ is denoted $\ell_{\{x,y\}}$.

We call $\mathbf{I}$ *connected* if $\mathrm{G}_{\mathbf{I}}$ is. Similarly, we say that $\mathbf{I}'$ is a connected component of $\mathbf{I}$ if $\mathrm{G}_{\mathbf{I}'}$ is a connected component of $\mathrm{G}_{\mathbf{I}}$.

The k^{th} *iterated degree* of a vertex x, where x is a variable or a constraint, is defined inductively by $\delta_0^{\mathbf{I}}(x) = \ell_x$, and $\delta_{k+1}^{\mathbf{I}}(x) = \{\{(\ell_{\{x,y\}}, \delta_k^{\mathbf{I}}(y)) \mid y \text{ is adjacent to } x \text{ in } \mathrm{G}_{\mathbf{I}}\}\}$. The *iterated degree* of a vertex x is defined as $\delta^{\mathbf{I}}(x) = (\delta_0^{\mathbf{I}}(x), \delta_1^{\mathbf{I}}(x), \delta_2^{\mathbf{I}}(x), \ldots)$. For vertices x and y we write $x \equiv_1 y$ if they have the same iterated degrees. Note that the iterated degrees are analogues of colors in the 1-dimensional Weisfeiler-Leman color refinement algorithm [17] for graph isomorphism test. The *iterated degree sequence* of $\mathbf{I}$ is defined as $\delta(\mathbf{I}) = \{\{\delta^{\mathbf{I}}(x) \mid x \in I \cup \mathcal{C}_{\mathbf{I}}\}\}$; for two σ-structures $\mathbf{I}$, $\mathbf{J}$, we write $\mathbf{I} \equiv_1 \mathbf{J}$ if they have the same iterated degrees sequence.[7] Notice that in order to prove that $\mathbf{I} \equiv_1 \mathbf{J}$ it is sufficient to show that $\{\{\delta^{\mathbf{I}}(x) \mid x \in I\}\} = \{\{\delta^{\mathbf{J}}(x) \mid x \in J\}\}$.

The computational model for solving $\mathrm{PVCSP}(\mathbf{A}, \mathbf{B})$ on a distributed network is as follows. An input valued structure $\mathbf{I}$ is represented as a bipartite message passing network designed as $\mathrm{G}_{\mathbf{I}}$: we have an agent $\alpha(x)$ for every vertex $x \in I \cup \mathcal{C}_{\mathbf{I}}$ and the communication channels exactly correspond to edges in $\mathrm{G}_{\mathbf{I}}$ and have the same labels. Every agent in the network knows only the template, the threshold, the number of variables ($|I|$), the number of constraints ($|\mathcal{C}_{\mathbf{I}}|$), and the labels of their controlled variable and of the adjacent channels. The agents are anonymous, they all run the same deterministic algorithm, and the communication proceeds in synchronous rounds. For a more detailed discussion on the distributed set-up, we refer the reader to [4].

We say that a distributed algorithm solves an instance $(\mathbf{I}, \tau)$ of $\mathrm{PVCSP}(\mathbf{A}, \mathbf{B})$ if the algorithm terminates and the terminating state of every process is Yes if $(\mathbf{I}, \tau)$ is a Yes instance of $\mathrm{PVCSP}(\mathbf{A}, \mathbf{B})$, and No if $(\mathbf{I}, \tau)$ is a No instance of $\mathrm{PVCSP}(\mathbf{A}, \mathbf{B})$. We say

[7] The degree sequence is often defined to be a list. However, when looking at iterated degree it is common [18, 19] and more practical to use multisets instead of lists, while maintaining the terminology *sequence* to highlight that we are dealing with a generalisation of the classical concept of degree sequence.

that a distributed algorithm solves PVCSP$(\mathbf{A}, \mathbf{B})$ if it solves every connected instance of PVCSP$(\mathbf{A}, \mathbf{B})$ (note here that it makes little sense to run a distributed algorithm on a disconnected network).

3 Fractional homomorphisms and SA^1

We start by stating the characterization of PVCSP templates in terms of fractional homomorphisms. The result will also be useful in the proof of Theorem 3.

A *fractional homomorphism* [22, 25] from $\mathbf{A}$ to $\mathbf{B}$ is a unary fractional polymorphism of $(\mathbf{A}, \mathbf{B})$, or equivalently, a probability distribution μ over B^A such that for every $R \in \sigma$ and every $\mathbf{a} \in A^{\mathrm{ar}(R)}$ we have that

$$\sum_{f \in B^A} \mu(f) R^{\mathbf{B}}(f(\mathbf{a})) \leq R^{\mathbf{A}}(\mathbf{a}). \tag{8}$$

If there exists a fractional homomorphism from $\mathbf{A}$ to $\mathbf{B}$, we say that $\mathbf{A}$ is fractionally homomorphic to $\mathbf{B}$ and we write $\mathbf{A} \to_f \mathbf{B}$.

The implication (1) ⇒ (2) in the following proposition is a well-known and easy calculation (see e.g. [22]). The converse implication appears to be new, although the proof technique via Farkas' Lemma [20] is standard in the VCSP area.

▶ **Proposition 5.** *For any two valued σ-structures $\mathbf{A}$ and $\mathbf{B}$, the following are equivalent.*

1. *There exists a fractional homomorphism from $\mathbf{A}$ to $\mathbf{B}$.*
2. *For all non-negative finite-valued σ-structures $\mathbf{I}$, $\mathrm{Opt}(\mathbf{I}, \mathbf{B}) \leq \mathrm{Opt}(\mathbf{I}, \mathbf{A})$.*

Proof. (1) ⇒ (2) Let μ be a fractional homomorphism from $\mathbf{A}$ to $\mathbf{B}$, let $g : I \to A$ be such that $\mathrm{Opt}(\mathbf{I}, \mathbf{A}) = \mathrm{Val}(\mathbf{I}, \mathbf{A}, g)$, and let $f \in B^A$ be some map that minimizes $\mathrm{Val}(\mathbf{I}, \mathbf{B}, f \circ g)$. Then

$$\begin{aligned}\mathrm{Opt}(\mathbf{I}, \mathbf{B}) \leq \mathrm{Val}(\mathbf{I}, \mathbf{B}, f \circ g) &\leq \sum_{f' \in B^A} \mu(f') \,\mathrm{Val}(\mathbf{I}, \mathbf{B}, f' \circ g) \\ &= \sum_{R \in \sigma} \sum_{\mathbf{v} \in I^{\mathrm{ar}(R)}} R^{\mathbf{I}}(\mathbf{v}) \sum_{f' \in B^A} \mu(f') R^{\mathbf{B}}(f' \circ g(\mathbf{v})) \\ &\leq \sum_{R \in \sigma} \sum_{\mathbf{v} \in I^{\mathrm{ar}(R)}} R^{\mathbf{I}}(\mathbf{v}) R^{\mathbf{A}}(g(\mathbf{v})) = \mathrm{Val}(\mathbf{I}, \mathbf{A}, g) = \mathrm{Opt}(\mathbf{I}, \mathbf{A}).\end{aligned}$$

(2) ⇒ (1). The idea for this proof is to assume that there is no fractional homomorphism from $\mathbf{A}$ to $\mathbf{B}$, formulate this fact as infeasibility of a system of linear inequalities, and then use a version of Farkas' Lemma to find $\mathbf{I}$ with $\mathrm{Opt}(\mathbf{I}, \mathbf{B}) > \mathrm{Opt}(\mathbf{I}, \mathbf{A})$.

The existence of a fractional homomorphism from $\mathbf{A}$ to $\mathbf{B}$ can be reformulated as the following system of linear inequalities, where there is a rational-valued variable μ_f for every $f \in B^A$.

$$\boxed{\begin{array}{rl} \text{variables:} & \mu_f \text{ for all } f \in B^A \\ \text{constraints:} & \displaystyle\sum_{f \in B^A} \mu_f R^{\mathbf{B}}(f(\mathbf{a})) \leq R^{\mathbf{A}}(\mathbf{a}) \text{ for all } R \in \sigma \text{ and } \mathbf{a} \in A^{\mathrm{ar}(R)} \\ & \displaystyle\sum_{f \in B^A} \mu_f \geq 1 \\ & \mu_f \geq 0 \text{ for all } f \in B^A. \end{array}} \tag{9}$$

If there is no fractional homomorphism from $\mathbf{A}$ to $\mathbf{B}$, this system is infeasible.

We now deal with infinite coefficients. Define $B^A_{<\infty} = \{f \in B^A : \forall R \in \sigma, \forall \mathbf{a} \in A^{\mathrm{ar}(R)},\ R^{\mathbf{A}}(\mathbf{a}) < \infty \text{ implies } R^{\mathbf{B}}(f(\mathbf{a})) < \infty\}$. Now consider the new linear system obtained from the above by first removing all the inequalities in (9) where $R^{\mathbf{A}}(\mathbf{a}) = \infty$ (since these inequalities are always satisfied), and second, by removing the variable μ_f for all $f \in B^A \setminus B^A_{<\infty}$ and changing (9) so that the sums run over $B^A_{<\infty}$ only (since we need to have $\mu_f = 0$ for $f \in B^A \setminus B^A_{<\infty}$ in any feasible solution). Clearly, the system of linear inequalities resulting from this procedure remains infeasible and does not contain infinite coefficients.

This system of linear inequalities can be rewritten in matrix form as $M\mathbf{f} \leq \mathbf{a}$ subject to $\mathbf{f} \geq 0$, where $\mathbf{f} \in \mathbb{Q}_{\geq}^{B^A_{<\infty}}$ is the vector of unknowns, and M is a real-valued matrix. By Farkas' Lemma, the system of inequalities $M^T\mathbf{y} \geq 0$ subject to $\mathbf{a}^T\mathbf{y} < 0$ and $\mathbf{y} \geq 0$ is feasible. Explicitly, the latter system is the following.

$$\boxed{\begin{array}{rl}
\text{variables:} & y, x_{R,\mathbf{a}} \text{ for every } R \in \sigma \text{ and } \mathbf{a} \in A^{\mathrm{ar}(R)} \text{ with } R^{\mathbf{A}}(\mathbf{a}) < \infty \\
\text{constraints:} & \displaystyle\sum_{R\in\sigma} \sum_{\substack{\mathbf{a}\in A^{\mathrm{ar}(R)} \\ R^{\mathbf{A}}(\mathbf{a})<\infty}} x_{R,\mathbf{a}} R^{\mathbf{B}}(f(\mathbf{a})) \geq y \text{ for all } f \in B^A_{<\infty} \\
 & \displaystyle\sum_{R\in\sigma} \sum_{\substack{\mathbf{a}\in A^{\mathrm{ar}(R)} \\ R^{\mathbf{A}}(\mathbf{a})<\infty}} x_{R,\mathbf{a}} R^{\mathbf{A}}(\mathbf{a}) < y \\
 & x_{R,\mathbf{a}} \geq 0 \text{ for all } R \in \sigma, \mathbf{a} \in A^{\mathrm{ar}(R)} \\
 & y \geq 0.
\end{array}} \tag{10}$$

Eliminating y, and adding trivially satisfied constraints to (10) for all $f \in B^A \setminus B^A_{<\infty}$, we get that the following system is feasible.

$$\boxed{\begin{array}{rl}
\text{variables:} & x_{R,\mathbf{a}} \text{ for every } R \in \sigma \text{ and } \mathbf{a} \in A^{\mathrm{ar}(R)} \text{ with } R^{\mathbf{A}}(\mathbf{a}) < \infty \\
\text{constraints:} & \displaystyle\sum_{R\in\sigma} \sum_{\substack{\mathbf{a}\in A^{\mathrm{ar}(R)} \\ R^{\mathbf{A}}(\mathbf{a})<\infty}} x_{R,\mathbf{a}} R^{\mathbf{B}}(f(\mathbf{a})) > \sum_{R\in\sigma} \sum_{\substack{\mathbf{a}\in A^{\mathrm{ar}(R)} \\ R^{\mathbf{A}}(\mathbf{a})<\infty}} x_{R,\mathbf{a}} R^{\mathbf{A}}(\mathbf{a}) \text{ for all } f \in B^A \\
 & x_{R,\mathbf{a}} \geq 0 \text{ for all } R \in \sigma, \mathbf{a} \in A^{\mathrm{ar}(R)}
\end{array}} \tag{11}$$

Let $x_{R,\mathbf{a}}$ for $R \in \sigma$, $\mathbf{a} \in A^r$ be a feasible solution to (11), and consider the structure $\mathbf{I}$ with domain $I = A$ and relations given by $R^{\mathbf{I}}(\mathbf{a}) = x_{R,\mathbf{a}}$ for $\mathbf{a} \in A^{\mathrm{ar}(R)}$ with $R^{\mathbf{A}}(\mathbf{a}) < \infty$ and $R^{\mathbf{I}}(\mathbf{a}) = 0$ whenever $R^{\mathbf{A}}(\mathbf{a}) = \infty$. Notice that $\mathbf{I}$ is non-negative finite-valued, that the right-hand side in the first inequality is equal to $\mathrm{Val}(\mathbf{I}, \mathbf{A}, \mathrm{id})$, (where id denotes the identity function) and that the left-hand side is equal to $\mathrm{Val}(\mathbf{I}, \mathbf{B}, f)$. Therefore $\mathrm{Opt}(\mathbf{I}, \mathbf{B}) > \mathrm{Opt}(\mathbf{I}, \mathbf{A})$, as required. ◀

Sketch of proof of Theorem 3. For an integer $m \geq 1$, let $\mathrm{LP}^m(\mathbf{A})$ be the structure whose universe consists of A-multisets of size m and whose valued relations are defined by the following formula where $R \in \sigma$ and $s_1, \ldots, s_r$ are from the universe.

$$R^{\mathrm{LP}^m(\mathbf{A})}(s_1, \ldots, s_r) := \frac{1}{m} \min_{\substack{\mathbf{t}_1,\ldots,\mathbf{t}_r \in A^m \\ \{\{\mathbf{t}_i\}\} = s_i}} \sum_{i=1}^{m} R^{\mathbf{A}}(\mathbf{t}_1[i], \ldots, \mathbf{t}_r[i]).$$

Variants of such structures have been defined in the literature both for (P)CSP [16, 1] and for VCSP [22, 25]. These papers also explicitly or implicitly observe the following properties.

1. $\mathrm{Opt}^{\mathrm{BLP}}(\mathbf{I},\mathbf{A}) = \min_{m\geq 1}\mathrm{Opt}(\mathbf{I},\mathrm{LP}^m(\mathbf{A}))$ for all non-negative finite-valued $\mathbf{I}$.
2. For all $m \geq 1$, $\mathrm{LP}^m(\mathbf{A}) \to_f \mathbf{B}$ if and only if $(\mathbf{A},\mathbf{B})$ has an m-ary symmetric fractional polymorphism.

The proof can be now finished using Proposition 5. For (iv) ⇒ (v) suppose that $(\mathbf{A},\mathbf{B})$ does not have a symmetric polymorphism of some arity m. Then, there is no fractional homomorphism from $\mathrm{LP}^m(\mathbf{A})$ to $\mathbf{B}$. It follows from Proposition 5 that there exists some structure $\mathbf{I}$ such that $\mathrm{Opt}(\mathbf{I},\mathbf{B}) > \mathrm{Opt}(\mathbf{I},\mathrm{LP}^m(\mathbf{A})) \geq \mathrm{Opt}^{\mathrm{BLP}}(\mathbf{I},\mathbf{A})$. Hence, BLP does not decide $\mathrm{PVCSP}(\mathbf{A},\mathbf{B})$. On the other hand, for (v) ⇒ (iv), assume that $(\mathbf{A},\mathbf{B})$ has symmetric fractional polymorphisms of every arity. Let $m \geq 1$ be such that $\mathrm{Opt}(\mathbf{I},\mathrm{LP}^m(\mathbf{A}))$ is minimal. We know that $\mathrm{LP}^m(\mathbf{A})$ is fractionally homomorphic to $\mathbf{B}$ and therefore for all finite-valued structures $\mathbf{I}$, $\mathrm{Opt}(\mathbf{I},\mathbf{B}) \leq \mathrm{Opt}(\mathbf{I},\mathrm{LP}^m(\mathbf{A})) = \mathrm{Opt}^{\mathrm{BLP}}(\mathbf{I},\mathbf{A})$. Hence, BLP decides $\mathrm{PVCSP}(\mathbf{A},\mathbf{B})$. ◀

The decomposition theorem mentioned in the introduction uses a concept that is "dual" to fractional homomorphism, as suggested by the following Proposition 6. Here we only present the proof of the implication that is needed for the decomposition theorem. The proof of the other implication uses techniques similar to the ones deployed in Proposition 5, and we refer the reader to [6] for the details.

We define a *dual fractional homomorphism* from $\mathbf{I}$ to $\mathbf{J}$ ($\mathbf{I} \to_{df} \mathbf{J}$) to be a probability distribution η over J^I such that for every $R \in \sigma$ and every $\mathbf{u} \in J^{\mathrm{ar}(R)}$ we have that

$$R^{\mathbf{J}}(\mathbf{u}) \geq \sum_{f\in J^I} \eta(f) \sum_{\substack{\mathbf{v}\in I^{\mathrm{ar}(R)}\\ \mathbf{u}=f(\mathbf{v})}} R^{\mathbf{I}}(\mathbf{v}). \tag{12}$$

▶ **Proposition 6.** *For any two non-negative finite-valued σ-structures $\mathbf{I}$ and $\mathbf{J}$, the following are equivalent.*
1. *There exists a dual fractional homomorphism from $\mathbf{I}$ to $\mathbf{J}$.*
2. *For all valued σ-structures $\mathbf{A}$, $\mathrm{Opt}(\mathbf{I},\mathbf{A}) \leq \mathrm{Opt}(\mathbf{J},\mathbf{A})$.*

Proof. (1) ⇒ (2). Let η be a dual fractional homomorphism from $\mathbf{I}$ to $\mathbf{J}$, and $g : J \to A$ be such that $\mathrm{Opt}(\mathbf{J},\mathbf{A}) = \mathrm{Val}(\mathbf{J},\mathbf{A},g)$. Then

$$\begin{aligned}\mathrm{Opt}(\mathbf{J}{,}\mathbf{A}) &= \sum_{R\in\sigma}\sum_{\mathbf{u}\in J^{\mathrm{ar}(R)}} R^{\mathbf{J}}(\mathbf{u})R^{\mathbf{A}}(g(\mathbf{u}))\\ &\geq \sum_{R\in\sigma}\sum_{\mathbf{u}\in J^{\mathrm{ar}(R)}}\sum_{f\in J^I}\eta(f)\sum_{\substack{\mathbf{v}\in I^{\mathrm{ar}(R)}\\ \mathbf{u}=f(\mathbf{v})}} R^{\mathbf{I}}(\mathbf{v})R^{\mathbf{A}}(g\circ f(\mathbf{v})) = \sum_{f\in J^I}\eta(f)\,\mathrm{Val}(\mathbf{I},\mathbf{A},g\circ f),\end{aligned}$$

which implies that there exists some function $f' : I \to J$ such that $\mathrm{Val}(\mathbf{I},\mathbf{A},g\circ f') \leq \mathrm{Opt}(\mathbf{J},\mathbf{A})$, hence $\mathrm{Opt}(\mathbf{I},\mathbf{A}) \leq \mathrm{Opt}(\mathbf{J},\mathbf{A})$ as required. Notice that this holds regardless of whether $\mathbf{A}$ is finite-valued or general-valued. ◀

4 The decomposition theorem

In this section we state and prove the decomposition theorem. This provides a connection between the combinatorial and the LP-based characterizations of the class of PVCSP templates that are the subject of our main result, and thus is a fundamental step in the proof of Theorem 8, namely the implication (ii) ⇒ (iii). We refer to [5] for a more detailed discussion about (a weaker form of) this result.

▶ **Theorem 7.** *Let* $\mathbf{I}$*,* $\mathbf{A}$ *be a pair of similar valued structures, where* $\mathbf{I}$ *is non-negative and finite-valued. Then there exist non-negative finite-valued structures* $\mathbf{Y}_1, \mathbf{Y}_2$ *such that*

1. $\mathbf{I} \to_{df} \mathbf{Y}_1$*,*
2. $\mathbf{Y}_1 \equiv_1 \mathbf{Y}_2$*, and*
3. $\mathrm{Opt}(\mathbf{Y}_2, \mathbf{A}) \leq \mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I}, \mathbf{A})$.

Proof. If $\mathrm{SA}^1(\mathbf{I}, \mathbf{A})$ is not feasible, then we can take $\mathbf{Y}_1 = \mathbf{Y}_2 = \mathbf{I}$, and the statement follows trivially, so from now on we shall assume that $\mathrm{SA}^1(\mathbf{I}, \mathbf{A})$ is feasible. Let $p_v(a)$, $p_{R(\mathbf{v})}(\mathbf{a})$ form an optimal solution of $\mathrm{SA}^1(\mathbf{I}, \mathbf{A})$ and let $m > 0$ be an integer such that all the values $mp_v(a)$ and $mp_{R(\mathbf{v})}(\mathbf{a})$ are integers. Note that these integers are non-negative by (3) and (5).

We define the universe of both valued structures $\mathbf{Y}_1$ and $\mathbf{Y}_2$ as $Y_1 = Y_2 = [m] \times I$. The valued structure $\mathbf{Y}_1$ is simply a "scaled disjoint union" of m copies of $\mathbf{I}$: we set $R^{\mathbf{Y}_1}((k, \mathbf{v}[1]), (k, \mathbf{v}[2]), \ldots, (k, \mathbf{v}[\mathrm{ar}(R)])) = 1/m \cdot R^{\mathbf{I}}(\mathbf{v})$ for every $k \in [m]$, $\mathbf{v} \in I^{\mathrm{ar}(R)}$, and the weight of the remaining tuples is set to 0. Observe that $\mathbf{I} \to_{df} \mathbf{Y}_1$ by the dual fractional homomorphism given by the uniform distribution over f_k, $k \in [m]$, where $f_k : I \to Y_1$ is defined by $f_k(v) = (k, v)$ for all $v \in I$. Also notice that the iterated degree of each (k, v) is obtained from the iterated degree of v by scaling down each constraint label (R, q) to $(R, q/m)$.

The structure $\mathbf{Y}_2$ is a "twisted" version of $\mathbf{Y}_1$ (the construction is a version of the twisted product from [15]). For every $v \in I$, fix a tuple $\mathbf{p}_v \in A^m$ in which $a \in A$ appears exactly $mp_v(a)$ times – note that this is possible since the $mp_v(a)$ sum up to m by (4). We define $h : Y_2 \to A$ by $h(k, v) = \mathbf{p}_v[k]$ for all $k \in [m]$ and $v \in I$. The structure $\mathbf{Y}_2$ is constructed so that the value of h for $(\mathbf{Y}_2, \mathbf{A})$ is $\mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I}, \mathbf{A})$, as follows. For every $R(\mathbf{v}) \in \mathcal{C}_{\mathbf{I}}$, denote $r = \mathrm{ar}(R)$, and consider an $m \times r$ matrix Q that has, for each $\mathbf{a} \in A^r$, exactly $mp_{R(\mathbf{v})}(\mathbf{a})$ rows equal to $\mathbf{a}$. Note that the i^{th} column contains $a \in A$ exactly $mp_{\mathbf{v}[i]}(a)$ times by (5), in other words, the multiset of elements of this columns is equal to $\{\{\mathbf{p}_{\mathbf{v}[i]}\}\}$; in particular, Q indeed has m rows. Moreover, if $\mathbf{v}[i] = \mathbf{v}[j]$, then the columns i and j are identical by (7). It follows that there are permutations $\rho_1, \ldots, \rho_r : [m] \to [m]$ such that

- for every $k \in [m]$, the k^{th} row of Q is equal to $(\mathbf{p}_{\mathbf{v}[1]}[\rho_1(k)], \mathbf{p}_{\mathbf{v}[2]}[\rho_2(k)], \ldots, \mathbf{p}_{\mathbf{v}[r]}[\rho_r(k)])$;
- for every $i, j \in [r]$, if $\mathbf{v}[i] = \mathbf{v}[j]$ then $\rho_i = \rho_j$.

We set $R^{\mathbf{Y}_2}((\rho_1(k), \mathbf{v}[1]), (\rho_2(k), \mathbf{v}[2]), \ldots, (\rho_r(k), \mathbf{v}[r])) = 1/m \cdot R^{\mathbf{I}}(\mathbf{v})$ for every $k \in [m]$. After running through all $R(\mathbf{v}) \in \mathcal{C}_{\mathbf{I}}$ we set the remaining weights to 0. The weights of those tuples that correspond to $R(\mathbf{v})$ were selected so that their contribution to $\mathrm{Val}(\mathbf{Y}_2, \mathbf{A}, h)$ is equal to the inner sum in the SA^1 objective function ($\star$); therefore, the total value of h is equal to $\mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I}, \mathbf{A})$. It follows that $\mathrm{Opt}(\mathbf{Y}_2, \mathbf{A}) \leq \mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I}, \mathbf{A})$. Moreover, the iterated degree of a pair (k, v) in $\mathbf{Y}_2$ is the same as in $\mathbf{Y}_1$ (note here that the second item above guarantees that repeated entries are handled correctly). It follows that $\mathbf{Y}_1 \equiv_1 \mathbf{Y}_2$, and the proof is concluded. ◀

The dual fractional homomorphism $\mathbf{I} \to_{df} \mathbf{Y}_1$, the equivalence $\mathbf{Y}_1 \equiv_1 \mathbf{Y}_2$, and assignments $Y_2 \to A$ that witness that $\mathrm{Opt}(\mathbf{Y}_2, \mathbf{A}) \leq \mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I}, \mathbf{A})$ from the proof of Theorem 7 can all be naturally associated with rational matrices (of dimensions $I \times Y_1$, $Y_1 \times Y_2$, and $Y_2 \times A$, respectively). It can be calculated that the product of these matrices is a matrix associated to a solution to the $\mathrm{SA}^1(\mathbf{I}, \mathbf{A})$ linear program. This is why we regard Theorem 7 as a decomposition theorem.

5 Main result

We are ready to prove the main result. The appropriate generalization of invariance under $\equiv_1$ (item (ii) in Theorem 1) is that if $\mathbf{I} \equiv_1 \mathbf{J}$ and $\tau \in \mathbb{Q}$, then it cannot happen that $(\mathbf{I}, \tau)$ is a Yes-instance while $(\mathbf{J}, \tau)$ is a No-instance. Item (ii) in the following theorem is a reformulation of this requirement.

▶ **Theorem 8.** *Let* $(\mathbf{A}, \mathbf{B})$ *be a promise valued template of signature* σ*. Then the following are equivalent.*

(i) *There exists a distributed algorithm that solves* $\mathrm{PVCSP}(\mathbf{A}, \mathbf{B})$*. Moreover, in such a case, there is a polynomial-time distributed algorithm that solves* $\mathrm{PVCSP}(\mathbf{A}, \mathbf{B})$*.*

(ii) *For all finite-valued* σ*-structures* $\mathbf{I}, \mathbf{J}$*, if* $\mathbf{I} \equiv_1 \mathbf{J}$ *then* $\mathrm{Opt}(\mathbf{J}, \mathbf{B}) \leq \mathrm{Opt}(\mathbf{I}, \mathbf{A})$*.*

(iii) SA^1 *decides* $\mathrm{PVCSP}(\mathbf{A}, \mathbf{B})$*.*

Proof. (i) ⇒ (ii). From the nature of the distributed model, it follows that agents with the same iterated degree will be in the same state at any time during the execution of any distributed algorithm. Therefore, if (i) holds and $\mathbf{I} \equiv_1 \mathbf{J}$ are connected, then a terminating distributed algorithm will report the same decision when run on input $(\mathbf{I}, \tau)$ or $(\mathbf{J}, \tau)$ (see Proposition 2.2 and Corollary 2.3 in [4]), so by setting $\tau = \mathrm{Opt}(\mathbf{I}, \mathbf{A})$ we obtain that (ii) holds for all connected $\mathbf{I}$, $\mathbf{J}$. We now show how (ii) in its full generality follows from (ii) restricted to connected $\mathbf{I}$ and $\mathbf{J}$.

Let us call two finite-valued σ-structures $\mathbf{I}$ and $\mathbf{J}$ *weakly congruent* if $|J| \cdot \delta(\mathbf{I}) = |I| \cdot \delta(\mathbf{J})$. We claim that $\mathrm{Opt}(\mathbf{J}, \mathbf{B})/|J| \leq \mathrm{Opt}(\mathbf{I}, \mathbf{A})/|I|$ whenever $\mathbf{I}$ and $\mathbf{J}$ are weakly congruent and connected. The claim clearly holds when $|I| = 1$ or $|J| = 1$, so assume $|I|, |J| \geq 2$. For any positive integer k, we define connected finite-valued σ-structures $\mathbf{I}'^{(k)}$ and $\mathbf{I}^{(k)}$ (and similarly $\mathbf{J}'^{(k)}$, $\mathbf{J}^{(k)}$) as follows. Let $I = \{v_0, v_1, \ldots, v_{|I|-1}\}$ and let the universe of $\mathbf{I}'^{(k)}$ be $\{0, 1, \ldots, k-1\} \times I$. Let η be the probability distribution over the mappings $I \to I^{(k)}$ assigning probability $1/2k$ to each of the $2k$ mappings f_j, f'_j, $j \in \{0, 1, \ldots, k-1\}$, where $f_j(v_i) = (j, v_i)$ and $f'_j(v_i) = (i + j \mod k, v_i)$ for each $v_i \in I$. We define the weights in $\mathbf{I}'^{(k)}$ in the unique way so that (12) holds for η with equality instead of inequality. Then η is a dual fractional homomorphism from $\mathbf{I}$ to $\mathbf{I}'^{(k)}$, and the probability distribution which assigns probability 1 to the projection onto I is a dual fractional homomorphism in the opposite direction. By Proposition 6, $\mathrm{Opt}(\mathbf{I}, \mathbf{C}) = \mathrm{Opt}(\mathbf{I}'^{(k)}, \mathbf{C})$ for any valued σ-structure $\mathbf{C}$. Finally, let $\mathbf{I}^{(k)}$ be the valued σ-structure obtained from $\mathbf{I}'^{(k)}$ by multiplying weights by $2k$; clearly, $\mathrm{Opt}(\mathbf{I}^{(k)}, \mathbf{C}) = 2k\,\mathrm{Opt}(\mathbf{I}'^{(k)}, \mathbf{C})$ for any $\mathbf{C}$. It follows from the construction that $\mathbf{I}^{(k)}$ is connected. Moreover, if k is large enough ($k \geq |I|$ suffices), then the iterated degree of (j, v_i) in $\mathbf{I}^{(k)}$ is obtained from the iterated degree of v_i in $\mathbf{I}$ by multiplying all the variable multisets in each of the elements of $\delta^{\mathbf{I}}(v_i)$ by 2 (in each inductive step in the definition of iterated degree). It follows that, for all k', the valued structures $\mathbf{J}^{(k'|I|)}$ and $\mathbf{I}^{(k'|J|)}$ are connected and, when k' is large enough, have the same iterated degree. By item (ii) for connected valued structures, we get $\mathrm{Opt}(\mathbf{J}^{(k'|I|)}, \mathbf{B}) \leq \mathrm{Opt}(\mathbf{I}^{(k'|J|)}, \mathbf{A})$ and the claim follows using the equalities above and rearranging.

Before finishing the proof, notice a simple consequence of the definition of iterated degrees. For a "variable vertex" x of $\mathrm{G}_{\mathbf{I}}$, a label S, and a "constraint vertex" y such that x and y are adjacent in $\mathrm{G}_{\mathbf{I}}$, denote $x[S, y] = \{y' \mid \delta(y') = \delta(y), \ell_{\{x,y'\}} = S\}$. Observe that if there exists an edge between x and y labeled S, then $\{x'[S, y] \mid \delta(x') = \delta(x)\}$ is a collection of mutually disjoint sets of equal size, which cover $\{y' \mid \delta(y) = \delta(y')\}$; and, moreover, the same claim holds when x' and y' are restricted to the connected component containing x (or y). It follows that for a component $\mathbf{I}'$ of $\mathbf{I}$ and a component $\mathbf{J}'$ of $\mathbf{J}$, where $\mathbf{I} \equiv_1 \mathbf{J}$, either the iterated degrees $\delta(\mathbf{I}')$ and $\delta(\mathbf{J}')$ are disjoint, or $\mathbf{I}'$ and $\mathbf{J}'$ are weakly congruent.

This observations allows us to finish the proof as follows. Let $\mathbf{I}$ and $\mathbf{J}$ be finite-valued σ-structures such that $\mathbf{I} \equiv_1 \mathbf{J}$ and let $n = |I| = |J|$. Then there are sequences $(\mathbf{I}_1, \ldots, \mathbf{I}_n)$ and $(\mathbf{J}_1, \ldots, \mathbf{J}_n)$ such that the first (resp., second) sequence contains each connected component $\mathbf{I}'$ of $\mathbf{I}$ (resp., $\mathbf{J}'$ of $\mathbf{J}$) exactly $|I'|$ times (resp., $|J'|$ times), and $\mathbf{I}_i$ and $\mathbf{J}_i$ are weakly congruent for every $i \in [n]$. From the claim above, we get $\mathrm{Opt}(\mathbf{J}_i, \mathbf{B})/|J_i| \leq \mathrm{Opt}(\mathbf{I}_i, \mathbf{A})/|I_i|$ for every $i \in [n]$. Summing up these inequalities and observing that $\mathrm{Opt}(\mathbf{I}, \mathbf{A})$ is equal to the sum of $\mathrm{Opt}(\mathbf{I}', \mathbf{A})$ over all connected components $\mathbf{I}'$ of $\mathbf{I}$ (and similarly for $\mathbf{J}$), item (ii) now follows.

(ii) ⇒ (iii). We need to show that for every non-negative finite-valued σ-structure $\mathbf{I}$, $\mathrm{Opt}(\mathbf{I}, \mathbf{B}) \leq \mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I}, \mathbf{A})$. Let $\mathbf{Y}_1, \mathbf{Y}_2$ be the structures obtained from Theorem 7, i.e., $\mathbf{Y}_1 \equiv_1 \mathbf{Y}_2$, $\mathrm{Opt}(\mathbf{Y}_2, \mathbf{A}) \leq \mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I}, \mathbf{A})$, and there is a dual fractional homomorphism from $\mathbf{I}$ to $\mathbf{Y}_1$. Then, by (ii) we have that $\mathrm{Opt}(\mathbf{Y}_1, \mathbf{B}) \leq \mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I}, \mathbf{A})$, and by Proposition 6, $\mathrm{Opt}(\mathbf{I}, \mathbf{B}) \leq \mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I}, \mathbf{A})$ too, as required.

(iii) ⇒ (i). From Theorem 3.2 in [4] (adapted to the valued setting), if $\mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I}, \mathbf{A}) < \infty$, then there is a solution to the linear program that assigns the same value to every class of variables and constraints of $\mathbf{I}$ that have the same iterated degree.[8] This allows us to reduce the linear program as follows. Let $I/{\equiv_1}$ and $\mathcal{C}_{\mathbf{I}}/{\equiv_1}$ denote the sets of equivalence classes of variables and constraints, respectively, under the equivalence $\equiv_1$. The new linear program, denoted $\mathrm{SA}^1_{\equiv}(\mathbf{I}, \mathbf{A})$, contains one variable $p_{[v]}(a)$ for every class $[v] \in I/{\equiv_1}$ and one variable $p_{[R(\mathbf{v})]}(\mathbf{a})$ for every class $[R(\mathbf{v})] \in \mathcal{C}_{\mathbf{I}}/{\equiv_1}$. The variables of the new program $\mathrm{SA}^1_{\equiv}(\mathbf{I}, \mathbf{A})$ are subject to the same constraints as in $\mathrm{SA}^1(\mathbf{I}, \mathbf{A})$, except they use the new reduced set of variables. The new objective function is

$$\mathrm{Opt}^{\mathrm{SA}^1_{\equiv}}(\mathbf{I}, \mathbf{A}) := \min \sum_{[R(\mathbf{v})] \in \mathcal{C}_{\mathbf{I}}/{\equiv_1}} k_{[R(\mathbf{v})]} \sum_{\mathbf{a} \in A^{\mathrm{ar}(R)}} p_{[R(\mathbf{v})]}(\mathbf{a}) R^{\mathbf{I}}(\mathbf{v}) R^{\mathbf{A}}(\mathbf{a}), \tag{13}$$

where $k_{[R(\mathbf{v})]} = |[R(\mathbf{v})]|$ is the number of constraints equivalent to $R(\mathbf{v})$. By the above discussion, we have $\mathrm{Opt}^{\mathrm{SA}^1_{\equiv}}(\mathbf{I}, \mathbf{A}) = \mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I}, \mathbf{A})$. Therefore, since SA^1 decides $\mathrm{PVCSP}(\mathbf{A}, \mathbf{B})$, so does $\mathrm{SA}^1_{\equiv}$. (We remark here that two input structures with the same iterated degree have the same reduced $\mathrm{SA}^1_{\equiv}$ up to renaming of variables; this can be used e.g. to show that (iii) implies (ii).)

In order to show that (iii) implies (i), assume that $\mathbf{I}$ is a connected input structure. We show that every agent in the distributed network can obtain the reduced linear program of $\mathrm{SA}^1_{\equiv}$ via a polynomial-time distributed algorithm. As $\mathrm{SA}^1_{\equiv}$ decides $\mathrm{PVCSP}(\mathbf{A}, \mathbf{B})$, this will conclude the proof.

The agents can calculate their iterated degree (or, rather, a finite and effectively computable representation thereof) in polynomial time using a simple distributed version of the color refinement algorithm. Each agent $\alpha(x), x \in I \cup \mathcal{C}_{\mathbf{I}}$ can then use the representation of the iterated degree as an identifier, see [4, Lemma 4.8.] for a more detailed discussion. Every agent can obtain sufficient information from its neighbours to compute the equations in (4), (5), (6) and (7) that constrain its relevant LP variables of the reduced system (and use the identifiers to name the LP variables), and can subsequently broadcast these along the network. We are left to show that every agent can also compute the objective function of $\mathrm{SA}^1_{\equiv}(\mathbf{I}, \mathbf{A})$. In fact, it is sufficient that every agent $\alpha(R(\mathbf{v}))$ computes the summand of $\mathrm{Opt}^{\mathrm{SA}^1_{\equiv}}(\mathbf{I}, \mathbf{A})$ that corresponds to $[R(\mathbf{v})]$ and then broadcasts it in order to obtain the complete objective function. The only nontrivial piece of information to compute is the value of the coefficients $k_{[R(\mathbf{v})]}$.

[8] We remark here that this theorem also has a substantially simpler proof – it is enough to observe that averaging over variables and constraints with the same iterated degrees does not increase the objective function.

By the observation made in the proof of (i) ⇒ (ii), for each $R(\mathbf{v}) \in \mathcal{C}_{\mathbf{I}}$, a participating variable $v \in \{\mathbf{v}\}$, and $S = \ell_{\{v,R(\mathbf{v})\}}$, the coefficient $k_{[R(\mathbf{v})]}$ is equal to the number of S-labeled edges from v into members of $[R(\mathbf{v})]$ (denoted $v[S, R(\mathbf{v})]$ above) multiplied by the size of $[v]$. The former value can be computed by $\alpha(v)$, so $\alpha(v)$ can compute the ratio $k_{[R(\mathbf{v}_1)]} : k_{[R(\mathbf{v}_2)]}$ for any two constraints $R(\mathbf{v}_1), R(\mathbf{v}_2)$ that v participates in. After broadcasting all these ratios, each agent can compute the ratios between any two $k_{[R(\mathbf{v})]}$ and, since the sum of these coefficients is $|\mathcal{C}_{\mathbf{I}}|$ (which is known to the agents), they can compute the coefficients. ◀

Clearly, the implication (iv) ⇒ (iii) in Theorem 1 remains true for PVCSP (so the equivalent statements in Theorem 8 are satisfied in, e.g., the PVCSPs in Example 4). The following example shows that, unlike for PCSPs, the converse implication does not hold in general: we provide an example of a PVCSP template that is decided by SA^1 but not by BLP.

▶ **Example 9.** Let $\mathbf{A}$, $\mathbf{B}$ be σ-structures where σ contains a single binary relation symbol R. Let $A = B = \{0, 1\}$, $R^{\mathbf{A}}(a, a) = R^{\mathbf{B}}(a, a) = 3$ for $a \in \{0, 1\}$, and $R^{\mathbf{A}}(a, b) = 2$, $R^{\mathbf{B}}(a, b) = 0$ for $a \neq b \in \{0, 1\}$. The probability distribution which assigns probability 1 to the identity function is a fractional homomorphism, and so $(\mathbf{A}, \mathbf{B})$ is a PVCSP template.

We claim that BLP does not decide $\mathrm{PVCSP}(\mathbf{A}, \mathbf{B})$. Indeed, let $\mathbf{I}$ be the PVCSP input structure given by $I = \{v\}$ and $R^{\mathbf{I}}(v, v) = 1$. Then, there is a feasible solution to $\mathrm{BLP}(\mathbf{I}, \mathbf{A})$ given by $p_v(a) = 1/2$ for $a \in \{0, 1\}$ and $p_{R(v,v)}(a, a) = 0$, $p_{R(v,v)}(a, b) = 1/2$ for $a \neq b \in \{0, 1\}$. This solution witnesses that $\mathrm{Opt}^{\mathrm{BLP}}(\mathbf{I}, \mathbf{A}) \leq 2$, however, it is easy to see that $\mathrm{Opt}(\mathbf{I}, \mathbf{B}) = 3$ and so BLP does not decide $\mathrm{PVCSP}(\mathbf{A}, \mathbf{B})$.

On the other hand, we show that $\mathrm{Opt}(\mathbf{I}, \mathbf{B}) \leq \mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I}, \mathbf{A})$ for any input valued structure $\mathbf{I}$. Let $V_l(\mathbf{I}) = \sum_{v \in I} R^{\mathbf{I}}(v, v)$ and $V_e(\mathbf{I}) = \sum_{u \neq v} R^{\mathbf{I}}(u, v)$ be the total weight of the constraints in $\mathbf{I}$ with and without repetitions, respectively. We choose an assignment $h : I \to B$ at random: each $h(v)$ is chosen independently and uniformly (both 0 and 1 with probability 1/2). The expected value of $\mathrm{Val}(\mathbf{I}, \mathbf{B}, h)$ is $3V_l(\mathbf{I}) + 3/2V_e(\mathbf{I})$, which implies that $\mathrm{Opt}(\mathbf{I}, \mathbf{B}) \leq 3V_l(\mathbf{I}) + 3/2V_e(\mathbf{I})$. As for SA^1, we know that any feasible solution must have $p_{(v,v)}(a, b) = 0$ whenever $a \neq b$. Therefore, we get

$$\mathrm{Opt}^{\mathrm{SA}^1}(\mathbf{I}, \mathbf{A}) = \min \Big[\sum_{v \in I} \sum_{a \in A} p_{R(v,v)}(a, a) R^{\mathbf{I}}(v, v) R^{\mathbf{A}}(a, a) + \sum_{u \neq v \in I} \sum_{a,b \in A} p_{R(u,v)}(a, b) R^{\mathbf{I}}(u, v) R^{\mathbf{A}}(a, b) \Big] \geq 3V_l(\mathbf{I}) + 2V_e(\mathbf{I}) > \mathrm{Opt}(\mathbf{I}, \mathbf{B}).$$

6 Conclusion

We have shown that solvability of a PVCSP by the SA^1 relaxation is equivalent to invariance under the Weisfeiler-Leman-like equivalence $\equiv_1$, and also to solvability in a natural distributed model. The distributed algorithm for the narrower CSP setting from [4] worked also for the search version of the problem, but this is unfortunately not the case for the algorithm presented in this paper. Is there an algorithm solving the search version of $\mathrm{PVCSP}(\mathbf{A}, \mathbf{B})$ whenever the PVCSP is solvable by SA^1? Note that in the search version an instance consists only of $\mathbf{I}$ and the goal is to find an assignment $h : I \to B$ such that $\mathrm{Val}(\mathbf{I}, \mathbf{B}, h) \leq \mathrm{Opt}(\mathbf{I}, \mathbf{A})$.

Another open problem emerges from Example 9 which shows that BLP and SA^1 are not equivalent for PVCSPs. It follows from [5] that BLP and SA^1 are equivalent for PCSPs and from [23] that they are also equivalent for finite-valued VCSPs. Are these relaxations equivalent for general-valued VCSPs?

References

1 Libor Barto, Jakub Bulín, Andrei A. Krokhin, and Jakub Opršal. Algebraic approach to promise constraint satisfaction. *J. ACM*, 68(4):28:1–28:66, 2021. doi:10.1145/3457606.

2 Libor Barto, Andrei Krokhin, and Ross Willard. Polymorphisms, and How to Use Them. In Andrei Krokhin and Stanislav Živný, editors, *The Constraint Satisfaction Problem: Complexity and Approximability*, volume 7 of *Dagstuhl Follow-Ups*, pages 1–44. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017. doi:10.4230/DFU.Vol7.15301.1.

3 A. A. Bulatov. A dichotomy theorem for nonuniform CSPs. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 319–330, October 2017. doi:10.1109/FOCS.2017.37.

4 Silvia Butti and Victor Dalmau. The complexity of the distributed constraint satisfaction problem. In Markus Bläser and Benjamin Monmege, editors, *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*, volume 187 of *LIPIcs*, pages 20:1–20:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.STACS.2021.20.

5 Silvia Butti and Víctor Dalmau. Fractional Homomorphism, Weisfeiler-Leman Invariance, and the Sherali-Adams Hierarchy for the Constraint Satisfaction Problem. In Filippo Bonchi and Simon J. Puglisi, editors, *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021, August 23-27, 2021, Tallinn, Estonia*, volume 202 of *LIPIcs*, pages 27:1–27:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.MFCS.2021.27.

6 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.

7 Tomás Feder and Moshe Y Vardi. The computational structure of monotone monadic SNP and constraint satisfaction: A study through datalog and group theory. *SIAM Journal on Computing*, 28(1):57–104, 1998.

8 Ferdinando Fioretto, Enrico Pontelli, and William Yeoh. Distributed constraint optimization problems and applications: A survey. *J. Artif. Int. Res.*, 61(1):623–698, January 2018.

9 Alexander Kazda. Minion homomorphisms give reductions between promise valued CSPs, 2021. In preparation.

10 Vladimir Kolmogorov, Andrei A. Krokhin, and Michal Rolínek. The complexity of general-valued CSPs. *SIAM J. Comput.*, 46(3):1087–1110, 2017. doi:10.1137/16M1091836.

11 Vladimir Kolmogorov, Johan Thapper, and Stanislav Živný. The power of linear programming for general-valued CSPs. *SIAM J. Comput.*, 44(1):1–36, 2015. doi:10.1137/130945648.

12 Marcin Kozik. Solving CSPs Using Weak Local Consistency. *SIAM Journal on Computing*, 50(4):1263–1286, 2021. doi:10.1137/18M117577X.

13 Andrei Krokhin and Stanislav Živný. *The Constraint Satisfaction Problem: Complexity and Approximability*, volume 7. Schloss Dagstuhl, 2017.

14 Andrei Krokhin and Stanislav Živný. The Complexity of Valued CSPs. In Andrei Krokhin and Stanislav Živný, editors, *The Constraint Satisfaction Problem: Complexity and Approximability*, volume 7 of *Dagstuhl Follow-Ups*, pages 233–266. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017. doi:10.4230/DFU.Vol7.15301.233.

15 Gábor Kun. Constraints, MMSNP and expander relational structures. *Comb.*, 33(3):335–347, 2013. doi:10.1007/s00493-013-2405-4.

16 Gabor Kun, Ryan O'Donnell, Suguru Tamaki, Yuichi Yoshida, and Yuan Zhou. Linear programming, width-1 CSPs, and robust satisfaction. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 484–495, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2090236.2090274.

17 AA Leman and B Weisfeiler. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsiya*, 2(9):12–16, 1968.

18 Motakuri V. Ramana, Edward R. Scheinerman, and Daniel Ullman. Fractional isomorphism of graphs. *Discrete Mathematics*, 132(1):247–265, 1994. doi:10.1016/0012-365X(94)90241-0.

19 Edward R Scheinerman and Daniel H Ullman. *Fractional graph theory: a rational approach to the theory of graphs*. Courier Corporation, 2011.

20 Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., USA, 1986.

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

22 Johan Thapper and Stanislav Živný. The power of linear programming for valued CSPs. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 669–678. IEEE Computer Society, 2012. doi:10.1109/FOCS.2012.25.

23 Johan Thapper and Stanislav Živný. The complexity of finite-valued CSPs. *J. ACM*, 63(4):37:1–37:33, 2016. doi:10.1145/2974019.

24 Johan Thapper and Stanislav Živný. The Power of Sherali-Adams Relaxations for General-Valued CSPs. *SIAM J. Comput.*, 46(4):1241–1279, 2017. doi:10.1137/16M1079245.

25 Caterina Viola and Stanislav Živný. The combined basic LP and affine IP relaxation for promise VCSPs on infinite domains. *ACM Trans. Algorithms*, 17(3):21:1–21:23, 2021. doi:10.1145/3458041.

26 Makoto Yokoo, Toru Ishida, Edmund H Durfee, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 614–621. IEEE, 1992.

27 Dmitriy Zhuk. A proof of the CSP dichotomy conjecture. *J. ACM*, 67(5):30:1–30:78, August 2020. doi:10.1145/3402029.

Trajectory Optimization for Safe Navigation in Maritime Traffic Using Historical Data

Chaithanya Basrur ✉
Singapore Management University, Singapore

Arambam James Singh ✉
National University of Singapore, Singapore

Arunesh Sinha ✉
Singapore Management University, Singapore

Akshat Kumar ✉
Singapore Management University, Singapore

T. K. Satish Kumar ✉
University of Southern California, Los Angeles, CA, USA

Abstract

Increasing maritime trade often results in congestion in busy ports, thereby necessitating planning methods to avoid close quarter risky situations among vessels. Rapid digitization and automation of port operations and vessel navigation provide unique opportunities for significantly improving navigation safety. Our key contributions are as follows. *First*, given a set of future candidate trajectories for vessels in a traffic hotspot zone, we develop a multiagent trajectory optimization method to choose trajectories that result in the best overall close quarter risk reduction. Our novel MILP-based optimization method is more than an order-of-magnitude faster than a standard MILP for this problem, and runs in near real-time. *Second*, although automation has improved in maritime operations, current vessel traffic systems (in our case study of a busy Asian port) predict only a *single* future trajectory of a vessel based on linear extrapolation. Therefore, using historical data we learn a *generative model* that predicts *multiple* possible future trajectories of each vessel in a given traffic hotspot, reflecting past vessel movement patterns, and integrate it with our trajectory optimizer. *Third*, we validate our trajectory optimization and generative model extensively using a real world maritime traffic dataset containing 6 million Automated Identification System (AIS) data records detailing vessel movements over 1.5 years from one of the world's busiest ports, demonstrating effective risk reduction.

2012 ACM Subject Classification Computing methodologies → Multi-agent planning

Keywords and phrases Multi-Agent Path Coordination, Maritime Traffic Control

Digital Object Identifier 10.4230/LIPIcs.CP.2022.5

Supplementary Material
Audiovisual (Videos and Appendix): `https://github.com/rlr-smu/TrajOpt`

Funding This research/project is supported by the National Research Foundation Singapore and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-RP-2020-016).

1 Introduction

Increasing maritime vessel traffic in some of the busiest ports of world such as Tokyo bay and Singapore creates traffic hotspots and increases the risk of closer quarter near-miss situations [17]. Recently, disruptions in global supply chains, and adverse weather events have further endangered the navigational safety by causing unexpected traffic spikes in busy waterways such as Singapore's port [31]. Vessel collisions endanger not only human lives,

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 5; pp. 5:1–5:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

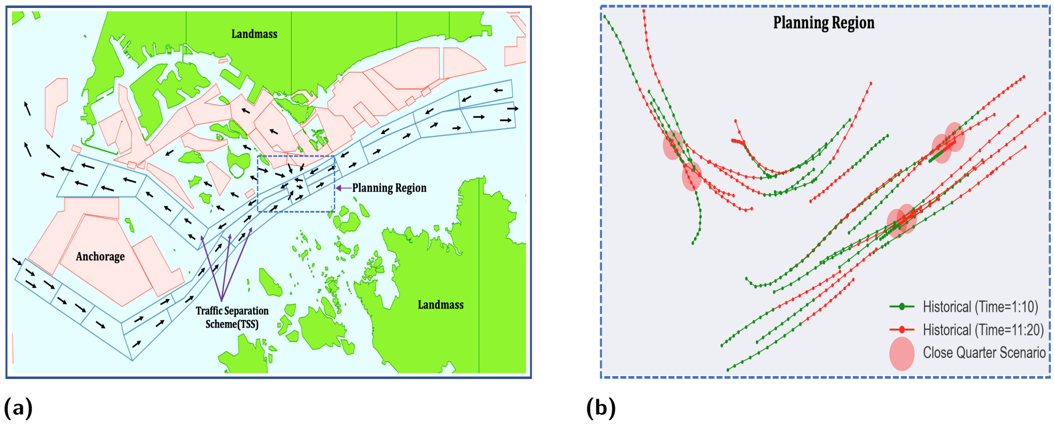


Figure 1 (a) Electronic navigation chart(ENC) of Singapore strait. ENC is used by vessels for navigation through the strait. Traffic separation scheme (TSS) are the sea lanes through which vessels enter and leave the strait. (b) Enlarged view of planning region from ENC, an instance of a congested scenario in this region is shown. Green and red line denote first 10-step and next 10-step trajectories from historical data respectively. This is the most congested region in the whole strait because many vessels enter and come out of the port through this junction point.

but also endanger the environment by causing oil spills [20]. Therefore, our goal in this work is to study and develop maritime traffic coordination techniques to mitigate close quarter risky situations that may develop in near future, and improve safety of navigation.

Current model of operations, automation in maritime traffic. Most busy ports, such as Singapore's, have a vessel traffic information system (VTIS) that is manned by port watch operators [19]. Operators keep a close watch over the vessel traffic 24x7 via radars and other sensors, and take action if a risky navigation situation is about to develop in the near future (e.g. in the next 10-15 mins). A key challenge is how to proactively advise involved vessels in a traffic hotspot to avoid the close quarter situation. Based on our discussions with Singapore's port authority and our physical observation of their VTIS control center, watch operators' traffic monitoring software predicts how vessels would move over the next 10-15 mins by *linearly* extrapolating their course. And if this linear prediction based trajectories suggest a close-quarter situation developing in the next 10-15 mins, it alerts the watch operator. However, it is left to the watch operator to decide how to advise vessels (e.g., to alter course) to avoid such a close quarter situation. This lack of automated trajectory optimization support creates high cognitive burden for watch operators, and is prone to human error. Therefore, our automated trajectory optimization tool can act as a decision support system for improved safety of maritime navigation.

In addition to improving current VTIS operations, developing automated trajectory optimization methods would also be highly impactful for the future of maritime traffic. E-Navigation [12] introduced by International Maritime Organisation (IMO) aims to improve maritime industry operations by digitizing both vessel navigation and port-based operations including digitizing communications between vessels and VTIS. Such digitization can further enable the usage of automated tools for improved safety of navigation. There are also recent advancements in the space of autonomous ships that have the potential to improve safety in navigation and also reduce costs to the industry [8, 24, 23]. Maritime Autonomous Surface Ships (MASS) [21] is an initiative by the IMO which provides regulations and guidelines on the advancement of technologies in this space. An example use-case of our tool would be (autonomous) vessels, which are in a traffic hotspot, propose a set of candidate trajectories which they can take in the near future and transmit it to the port authority. Using our trajectory optimization tool, the port authority can then advise vessels to take the least-risky trajectory.

Electronic navigation chart and planning region. Figure 1a shows the electronic navigation chart of Singapore strait. The traffic separation scheme (TSS) are the sea lanes through which vessels enter and leave the strait. Each smaller polygon represents a sea zone. Other areas of interests are marked in different colors such as anchorages, landmass among others. Our area of focus (or *planning* region) highlighted in dotted square is a typical hotspot region. In this region, vessels enter port waters (in pink color) towards berths, outgoing vessels from berths enter TSS, and some vessels transit through the TSS. As a result, this planning region experiences heavy cross traffic with vessels often navigating across traffic separation areas to avoid hotspots.

Figure 1b shows a collection of real historical trajectories for different vessels (tankers and cargos). Each dot in a trajectory shows the corresponding vessel position 1 minute apart. Red circles highlight those locations where vessels are in a close quarter situation (distance between them is less than 500 meters).

Generative trajectory modeling using historical data. Our trajectory optimization tool requires a candidate set of possible future trajectory for each vessel involved in a hotspot region. In the future with increased automation, vessels themselves can digitally compute a number of feasible trajectories they can follow in the near future (next 10-15 mins) and transmit them to the port authority. There are existing tools such as ECDIS [13] for vessel route planning. However, the current vessel-port operations are unable to provide such information. Similarly, current VTIS (such as Singapore port's) predicts only a single trajectory based on linear extrapolation.

To address this, we develop a generative model takes as input the past 10 minute trajectory of each vessel in a specific hotspot area ('time=1:10'), and then predicts a number of possible trajectories for each vessel for the next 10 minutes ('time=11:20'). Our generative model is trained on a large historical dataset, which implies that generated future trajectories are feasible (that is, they do not involve unrealistic manoeuvres such as taking U-turns, or making vessels fully stop). Instead of vessels deciding independently their future trajectory ('time=11:20'), our trajectory optimization module can pick safest possible trajectories, among the ones generated by our generative model, to decrease the risk by maximizing the closest point of approach (CPA) between any two vessels. Note that all the information required for this system is available with the port authority as they monitor movements of all vessels, and based on our optimization tool, can advise vessels to follow a particular trajectory. As traffic is dynamic, another property our tool must have is to produce results in near real-time, and be able to run on a rolling horizon basis.

Contributions. Our key contributions are as follows. First, given a set of candidate future trajectories of vessels in a hotspot region, we develop a mixed-integer linear programming (MILP) based optimization method that can optimize over all possible combinations of future vessel trajectories to minimize the risk of close quarter situations developing. Second, using historical data of vessel movements in Singapore strait, one of busiest port in the world, we learn a deep conditional generative model based on LSTM [11] that can predict multiple possible future trajectories of vessels in a traffic hotspot region. Empirically, our generative model is trained and evaluated on a real world historical data containing 6 million data points (AIS records detailing vessel traffic data) over 1.5 years. We extensively test several properties of our generative model, such as its ability to generate realistic and diverse trajectories, and show that our method is significantly superior to another generative model called social GAN [10]. We also show that our novel MILP-based trajectory optimization method is more than an order-of-magnitude faster than a standard MILP model for this problem, and can provide solutions in near real-time, a key requirement for the solver.

Generative modeling. The learning of probability distributions from data and ability to sample from them is a fundamental learning task known as generative modelling. Within generative models, a prominent sub-class are techniques that do not explicitly learn the probability distribution (or density) function but are able to generate samples from it. These include the classical Markov chain Monte Carlo methods [3] as well as modern Generative Adversarial Networks [9] (GANs). GAN-based models have also been used for generative trajectory modeling of pedestrians [10, 2, 15]. However, vessel traffic has movement characteristics which are unlike pedestrians' (such no sharp or uturns, prediction over longer time duration among others). Our proposed generative model is also computationally efficient than previous approaches such as SocialGAN [10] which take much longer to train, and often produce worse predictions as we show empirically.

Multi-agent path finding (MAPF). Given a set of agents with unique start and goal locations in a shared environment, the MAPF problem [28] is to find collision-free paths for all agents from their respective start locations to their respective goal locations. MAPF has many real-world applications, including in video games [25], automated warehousing [32], multi-drone delivery [5], and aircraft-towing vehicles [18]. Solving the MAPF problem optimally for either the minimum sum-of-costs or the minimum makespan is NP-hard [34, 16]. Although many MAPF solvers exist, they are not directly applicable in our problem domain for the following reasons. First, MAPF solvers require a discrete search space. They discretize continuous spaces even when they use motion primitives and thereby generate only piecewise smooth paths uncharacteristic of trajectories in our problem domain. Second, MAPF solvers don't constrain branching decisions at intermediate locations and therefore don't reason about historical data required for capturing the complex kinodynamic constraints on trajectories in our problem domain. Third, MAPF solvers are typically interested in avoiding collisions as hard constraints rather than optimizing close quarter risk reduction.

Maritime traffic optimization. Previous works [27, 26] have proposed a reinforcement learning based approach to address the maritime traffic control problem. Their main focus is to optimize the traffic for the entire Singapore Strait. Whereas, our focus is more on the *micro-level* traffic optimization, that is, minimizing the close quarter incidents in the near future. [1, 4, 35, 29] have also addressed the traffic control problem at the micro-level. However the solution methodologies in these approaches *do not* model the uncertainty in the maritime environment, which is an important real world feature. In this work, we capture the uncertainty in vessel movements using our proposed generative model.

2 Problem Formulation and Statement

We tackle the problem of recommending seafaring vessels a navigation route in and around congested ports (e.g., the planning region in figure 1a). The increasing number of vessels over the years has led to an increase in the frequency of collisions and close quarter situations.

Thus, the need of the hour is some intervention from a central port authority to encourage safe navigation around crowded ports. We envision a route recommendation system that suggest routes to vessels involved in a traffic hotspot. A number of competing considerations need to be taken into account for such a system: the recommendations should follow typical paths traversed historically, the path suggested must be easy to execute by the vessel, and achieve a global objective of minimizing the close-quarter risk among vessels.

Problem Statement. To specify the problem formally, we consider a planning horizon H and planning area $\mathcal{Z}$; a polygon in sea space. The planning area is typically the port area prone to traffic hotspots, such as where vessels either enter the port waters for berthing and anchoring or pass through to open seas. Vessels enter and leave $\mathcal{Z}$ during the planning period H. At a planning epoch t, we observe a snapshot of the whole planning region which includes information such as, total number of vessels M and their previous trajectories until time t. We use $[M]$ as a shorthand for $\{1, \dots, M\}$. $v \in [M]$ denotes a vessel, $\tau_v^{\text{prv}} = \langle l_t, l_{t-1}, \dots, l_{t-(n-1)} \rangle$ denotes the past n-step trajectory of vessel v at time t, where $l_t = (x_t, y_t)$ is the location on 2d plane. We also assume that time is discretized (e.g., 30 second intervals). The objective of a maritime traffic controller is to recommend future m-steps trajectories τ_v^{rec} for each vessel v so that it minimizes a risk function $\text{risk}(\tau_1^{\text{rec}}, \tau_2^{\text{rec}}, \dots, \tau_M^{\text{rec}})$ given by

$$\text{risk}(\tau_1^{\text{rec}}, \dots, \tau_M^{\text{rec}}) = -\min_{v,v' \in [M], v \neq v'} \Big\{ \text{dist}(\tau_v^{\text{rec}}, \tau_{v'}^{\text{rec}}) \Big\} \tag{1}$$

where the function dist provides the closest distance between the two input trajectories. Thus, in words, the risk measures the *negative* of the closest point of approach between any two vessels for the recommended trajectories. Minimizing the risk means maximizing the closest point of approach (CPA), which is a standard notion for maritime safety [6].

3 Approach

Our approach to solve the trajectory recommendation problem has two parts. First, given a set of candidate future trajectories for each vessel in a hotspot area, we develop an optimization method that selects trajectories to minimize the risk. Second, we develop a trajectory generation model that generates multiple plausible trajectories for each vessel that can be recommended.

3.1 Vessel Trajectory Optimization

We first formulate the path planning problem as a trajectory optimization problem. Suppose that $\{\tau_v^1, \tau_v^2, \dots \tau_v^K\}$ are the K future plausible trajectories for a vessel v. There are multiple ways in which such future trajectories can be collected – vessels themselves send future possible trajectories they can follow (e.g., using route planning tools such as ECDIS as noted in Section 1), VTIS can use their own prediction methods, or as in our case, trajectory generation module can be used (described in next sub-section). Importantly, our trajectory optimization method is not dependent on the manner in which such future trajectories are collected.

Let x_v^k be a binary decision variable that denotes whether the trajectory k is selected as the recommendation. Thus, a natural constraint is $\sum_{k=1}^{K} x_v^k = 1, \quad \forall v \in [M]$ which enforces that only one trajectory is selected per vessel.

Table 1 RiskOPT: Mixed-integer non-linear program for trajectory optimization.

$$\begin{aligned}
\max_{\mathbf{x}} \min_{v\in[M],v'\in[M],v\neq v'} & \left\{ \sum_{k\in[K],k'\in[K]} x_v^k x_{v'}^{k'} \operatorname{dist}(\tau_v^k, \tau_{v'}^{k'}) \right\} \\
\text{subject to} & \sum_{k=1}^{K} x_v^k = 1, \quad \forall v \in [M] \\
& x_v^k \in \{0,1\} \quad \forall v \in [M], k \in [K]
\end{aligned}$$

We re-write the risk in terms of all the binary variables x_v^k's. Let $\mathbf{x}$ denote all the binary variables for all vessels. For defining risk($\mathbf{x}$) only those trajectories must be considered that are selected, which we enforce by the bilinear term $x_v^k x_{v'}^{k'}$ below:

$$-\min_{v\in[M],v'\in[M],v\neq v'} \left\{ \sum_{k\in[K],k'\in[K]} x_v^k x_{v'}^{k'} \operatorname{dist}(\tau_v^k, \tau_{v'}^{k'}) \right\} \tag{2}$$

We want to minimize risk, which given the negative sign in risk becomes the integer bilinear optimization problem RiskOPT in Table 1.

Naive Formulation. The problem RiskOPT is non-linear because of the bilinear terms. A naive and standard way of removing the bilinearity is to introduce additional continuous variables $z_{k,k',v,v'} = x_v^k x_{v'}^{k'}$ and constraints $z_{k,k',v,v'} \geq x_v^k + x_{v'}^{k'} - 1$ and $z_{k,k',v,v'} \leq x_v^k$ and $z_{k,k',v,v'} \leq x_{v'}^{k'}$. It can be readily checked that this re-formulation is equivalent to the original one. This reformulation uses K^2M^2 extra variables and $3K^2M^2$ extra constraints over the original bilinear formulation. However, our planning needs to be almost real time (solve within one minute) and as observed in experiments, this naive reformulation does not meet this requirement. Hence, we present a more compact reformulation that is orders of magnitude faster than the naive one.

Improved Formulation. Observe that the key part of expression (2) can be re-written as:

$$\sum_{k\in[K]} x_v^k \Big(\sum_{k'\in[K]} x_{v'}^{k'} \operatorname{dist}(\tau_v^k, \tau_{v'}^{k'}) \Big) \tag{3}$$

We use the shorthand:

$$f_{k,v}(\mathbf{x}_{v'}) = \sum_{k'\in[K]} x_{v'}^{k'} \operatorname{dist}(\tau_v^k, \tau_{v'}^{k'}).$$

Here $\mathbf{x}_{v'} = \langle x_{v'}^1, \dots, x_{v'}^K \rangle$ is the vector of variables for vessel v'. Note that $f_{k,v}(\mathbf{x}_{v'})$ is linear in $\mathbf{x}_{v'}$. The expression (3) now simplifies to:

$$\sum_{k\in[K]} x_v^k f_{k,v}(\mathbf{x}_{v'}) \tag{4}$$

We now replace $x_v^k f_{k,v}(\mathbf{x}_{v'})$ with a real valued variable $z_{k,v,v'}$ to get a reformulation of (4) as:

$$\sum_{k\in[K]} z_{k,v,v'} = \sum_{k\in[K]} x_v^k f_{k,v}(\mathbf{x}_{v'}).$$

Table 2 CompactRiskOPT: Compact Mixed-integer linear program for trajectory optimization.

$$\begin{aligned}
&\max_{\mathbf{x},\mathbf{z},y} \quad y \\
&\text{subject to } \sum_{k=1}^{K} x_v^k = 1, \quad \forall v \in [M] \\
&\qquad\qquad \text{Constraint set from Eq. 7} \\
&\qquad\qquad \text{Constraint set from Eq. 5} \\
&\qquad\qquad \text{Constraint set from Eq. 6} \\
&\qquad\qquad x_v^k \in \{0,1\} \quad \forall v \in [M], k \in [K]
\end{aligned}$$

Additionally, we also show that $f_{k,v}(\mathbf{x}_{v'})$ can be easily lower and upper bounded so that the relationship between $z_{k,v,v'}$ and $x_v^k f_{k,v}(\mathbf{x}_{v'})$ can be expressed as linear constraints. Let lower bound:

$$L_{k,v,v'} = \min_{\mathbf{x}_{v'}} f_{k,v}(\mathbf{x}_{v'}) = \min_{k'} \operatorname{dist}(\tau_v^k, \tau_{v'}^{k'}).$$

The second equality above follows from the definition of $f_{k,v}(\mathbf{x}_{v'})$ and the constraint that $\sum_{k' \in [K]} x_{v'}^{k'} = 1$. Similarly, let upper bound:

$$U_{k,v,v'} = \max_{\mathbf{x}_{v'}} f_{k,v}(\mathbf{x}_{v'}) = \max_{k'} \operatorname{dist}(\tau_v^k, \tau_{v'}^{k'}).$$

Lower bounds L and U can be easily computed for each tuple $\langle k, v, v' \rangle$ before we setup the optimization problem. To replace $x_v^k f_{k,v}(\mathbf{x}_{v'})$ with a real valued $z_{k,v,v'}$, we first add the constraints:

$$L_{k,v,v'} x_v^k \le z_{k,v,v'} \le U_{k,v,v'} x_v^k \quad \forall k \in [K], v \in [M], v' \in [M], v \neq v' \tag{5}$$

This constraint ensures that $z_{k,v,v'} = 0$ if $x_v^k = 0$. We still need to ensure that if $x_v^k = 1$ then $z_{k,v,v'} = f_{k,v}(\mathbf{x}_{v'})$. Towards this end, we add the constraints:

$$\begin{aligned} &f_{k,v}(\mathbf{x}_{v'}) - U_{k,v,v'}(1 - x_v^k) \le z_{k,v,v'} \le f_{k,v}(\mathbf{x}_{v'}) - \\ &\quad L_{k,v,v'}(1 - x_v^k) \quad \forall k \in [K], v \in [M], v' \in [M], v \neq v' \end{aligned} \tag{6}$$

In the above constraint, if $x_v^k = 1$ then $z_{k,v,v'} = f_{k,v}(\mathbf{x}_{v'})$ and this value of $z_{k,v,v'}$ is also feasible for the previous constraint. Also, when $x_v^k = 0$ then the previous constraint gives $z_{k,v,v'} = 0$ which is still feasible for the above constraint.

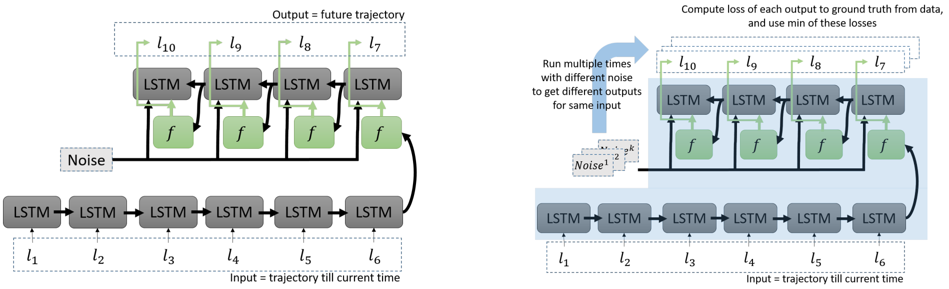
This adds KM^2 continuous variables and $4KM^2$ inequalities; note the reduction in the number of these additional variables and constraints as compared to the naive approach. Also, note that the new objective $\sum_{k \in [K]} z_{k,v,v'}$ is now completely continuous. By introducing an additional variable y which is to stand for $\min_{v \in [M], v' \in [M], v \neq v'} \left\{ \sum_{k \in [K]} z_{k,v,v'} \right\}$ and the constraints

$$\sum_{k \in [K]} z_{k,v,v'} \ge y \; \forall k \in [K], v \in [M], v' \in [M], v \neq v' \tag{7}$$

the max-min optimization becomes the Mixed Integer Linear Program (MILP) CompactRiskOPT in Table 2.

The arguments presented till now directly leads to the following formal claim of correctness

▶ **Propostion 1.** *Optimization* CompactRiskOPT *is equivalent to* RiskOPT.


Output = future trajectory
l_{10}
l_9
l_8
l_7
LSTM
f
Noise
l_1
l_2
l_3
l_4
l_5
l_6
Input = trajectory till current time
Compute loss of each output to ground truth from data, and use min of these losses
Run multiple times with different noise to get different outputs for same input
$Noise^1$
$Noise^2$
$Noise^k$

(a) Architecture of generator.

(b) Illustration of diversity loss.

Figure 2 A LSTM based generative model (left) with diversity loss computation (right). This instance of the generative model is shown with $n = 6$ and $m = 4$. The exact diversity loss formula is in text in Equation 8.

3.2 Trajectory Generation

Our aim is to use historical data to generate the K plausible trajectories. In practice, such generation would be performed by port authority using human expertise, their own prediction methods or vessels themselves can compute it using routing tools such as ECDIS, as mentioned in the introduction. However, these possible trajectories are not recorded in the maritime traffic dataset that are commercially available[1], which only record historical movement of vessels. Clearly, randomly generating trajectories produces very unrealistic trajectories. Instead, we use a generative adversarial networks (GAN) [9] like set-up to learn from historic data and generate multiple future trajectories; the GAN-like learning ensures realism by generating trajectories close to observed ground truth trajectories in the data. We generate a set of K trajectories $\{\tau_v^1, \tau_v^2, \ldots \tau_v^K\}$ for each vessel v. The selection of one trajectory among this set for each vessel is done in the path planning part as described in the previous sub-section. We take inspiration from a GAN to build a simpler architecture for trajectory generation that is easier to train and achieves better results in experiments.

More formally, the goal is to output multiple future trajectories $\{\tau_v^1, \tau_v^2, \ldots \tau_v^K\}$ for each of the M vessels starting from current time step t, where each trajectory τ_v^i is of m time steps. Each time step is 1 minute in wall-clock time. The input to this task is the previous n-step trajectory τ_v^{prv}. During training the future trajectory (ground truth) is known and specified as $\tau_v^{\text{true}} = \langle l_{t+1}^{\text{true}}, \ldots, l_{t+m}^{\text{true}} \rangle$, where $l_t^{\text{true}} = (x_t^{\text{true}}, y_t^{\text{true}})$ is the location of the vessel on the 2D plane (this is available from the historical data).

LSTM architecture. Our architecture for this generative task is shown in Figure 2a. It is essentially a LSTM layer (we call this the generator g_θ). LSTMs are a special kind of recurrent neural networks, capable of learning long-term dependencies [11]. All recurrent neural networks have the form of a chain of repeating modules of neural network. A LSTM layer also has this chain like structure where each repeating structure is called a LSTM cell. Each LSTM cell takes in an input (one element of a sequence which is a location l_i in our case) and outputs a hidden value h_i that is fed to the next LSTM cell. The chain structure ensures that h_i captures the information about all the inputs l_j with $j \leq i$. The last cells in a LSTM layer output the predicted future elements of the sequence.

[1] https://www.marinetraffic.com

In our architecture, the first n LSTM cells take in as input the past n locations given by τ_v^{prv}. The next m cells output the future prediction with a sequential structure where the prediction at a time-step forms the input for the LSTM cell that predicts the location for the next time step. The predicted location output is formed by transforming the h_i value from the output LSTM cells by passing h_i through a fully connected layer represented by f in Figure 2a.

Stochastic predictions. Importantly, all the future predictor cells also take in Gaussian noise z_1 of dimension D as input, which enables stochastic predictions that provide the multiple future trajectories we need. Multiple trajectories provide flexibility for the optimizer to reach better solutions.

Next, we describe the loss function. During training, for any given predicted sequence output $\widehat{\tau}_v^j = \langle \widehat{l}_{t+1}^j, \ldots, \widehat{l}_{t+m}^j \rangle$ we define a loss $L(\widehat{\tau}_v^j, \tau_v^{\text{true}}) = \sum_{i=1}^m ||l_{t+i}^{\text{true}} - \widehat{l}_{t+i}^j||_2$. However, instead of using just one predicted sequence we invoke the generator S times with different noise samples (and same past location input) to obtain S distinct predicted sequences and form the overall loss $\mathcal{L}$ as follows:

$$\mathcal{L}\big(\{\widehat{\tau}_v^1, \ldots \widehat{\tau}_v^S\}, \tau_v^{\text{true}}\big) = \min_{j \in \{1,\ldots,S\}} L(\widehat{\tau}_v^j, \tau_v^{\text{true}}) \tag{8}$$

This loss is illustrated in Figure 2b. The above loss function is known as Minimum over N (MoN) loss [7] in prior literature and has been used as an additional loss term for diverse samples in SocialGAN [10] for pedestrian trajectory prediction. To understand this loss, note that replacing the min with average or max will force all generated trajectories to collapse to that single trajectory that provides the lowest loss, thereby producing a deterministic prediction instead of the desired stochastic prediction. The min allows for diverse samples while still ensuring that the distribution that generates these samples is able to generate samples close to the ground truth.

Observe that our stochastic prediction is history dependent, which implicitly takes into account the speed of the vessel (which in turn depends on external but unknown factors such as weather and vessel type). In particular, we use only the information that the current VTIS system uses in Singapore port, which simply linearly extrapolates the vessel's current trajectory for prediction.

Also note the distinct aspect that unlike a GAN (e.g., SocialGAN) there is no discriminator network in our architecture, but the loss function of the generator g_θ uses the diversity loss to generate required trajectories. The absence of a discriminator removes the need for adversarial training process of typical GANs, making our training process much more stable and computationally faster, which is critical for us given the large data size. Moreover, we demonstrate experimentally that our approach outperforms SocialGAN as well as a simple linear extrapolation which is the current approach followed by Singapore port's VTIS. In particular, we use three prior proposed metrics to demonstrate the superiority of our approach; these include two common metrics in trajectory prediction, namely Average Displacement Error (ADE) and Final Displacement Error (FDE), and a metric named discriminative score proposed in time-series generation [33]. The ADE and FDE compare the generated trajectories with the actual historical trajectory, and also showcase the diversity in our stochastic predictions. The discriminative score metric ensures that our generated trajectories are realistic (i.e., similar to trajectories in the historical dataset). These metrics are explained in the experiment section.

4 Experiments

We evaluate our proposed learning and planning based system on real world maritime dataset. We use 1.5 years of historical Automatic Identification System (AIS) data (spanning the months between January 2018 -June 2019) of vessels voyaging in the Singapore Strait purchased from the company MarineTraffic. Each AIS record contains information such as timestamp, vessel unique id, lat-long (GPS) positions, course over ground (COG), speed over ground (SOG) and navigation status (e.g anchored/sailing etc). The vital vessel navigation information such as lat-long positions are logged every few seconds interval resulting in total of around 6 million records. Our evaluation is mainly for tankers and cargo vessels because majority of traffic involved in hotspot formation belong to these two types. They are also generally considered as *high-risk* category vessels due to type and size of cargo they carry.

We further process the data to get about 1.6 million individual vessel trajectories for our proposed method in the planning region (shown in figure 1a) . Each vessel trajectory includes 20 latitude-longitude reported at intervals of one minute. These trajectories are used to train our generative model as explained later. Additional experimental details are in the supplemental material.

4.1 Trajectory Generation

In addition to the maritime data we also evaluate our trajectory generation model on three publicly available human pedestrian trajectory datasets (ETH, Hotel, Zara1) [14, 22]. The data includes 2200 trajectories of human movement behaviour in congested environments. The results are in the supplement and are provided mainly to showcase that our proposed approach is competitive with socialGAN even on datasets socialGAN is optimized for.

Evaluation metrics. We use commonly adopted metrics – ADE and FDE [15, 2] and discriminative score [33] for evaluating generated trajectories:

- **Average displacement error (ADE)**: Average L2 distance between the ground truth τ_v^{true} and the k^{th} predicted trajectory $\widehat{\tau}_v^k$ over all predicted locations in $\widehat{\tau}_v^k$.

$$\text{ADE}(\widehat{\tau}_v^k, \tau_v^{\text{true}}) = \frac{\sum_{i=1}^m ||l_{t+i}^k - l_{t+i}^{\text{true}}||_2}{m} \tag{9}$$

  For a given trajectory τ_v^{prv} , we sample K future trajectories from the generator. The best and mean ADEs are given by:

$$(best)\,\text{ADE} = \min_{k\in[K]} \text{ADE}(\widehat{\tau}_v^k, \tau_v^{\text{true}}) \tag{10}$$

$$(mean)\,\text{ADE} = \frac{\sum_{k=1}^K \text{ADE}(\widehat{\tau}_v^k, \tau_v^{\text{true}})}{K} \tag{11}$$

  We compare our approach with Social GAN on both best and mean ADE.

- **Final displacement error (FDE)**: It is the L2 distance between the ground truth and the k^{th} prediction at the final predicted location for this trajectory.

$$\text{FDE}(\widehat{\tau}_v^k, \tau_v^{\text{true}}) = ||l_{t+m}^k - l_{t+m}^{\text{true}}||_2 \tag{12}$$

  The calculations for best and mean FDE is similar to the ones of the ADE in equation (10) and (11).

  The ADE and FDE metric show both the quality of predictions and diversity of trajectory generation. If the best ADE (and FDE) is low, then it implies, there is at least one

Table 3 ADE and FDE comparison between our approach and SocialGAN for the Maritime navigational data (lower is better).

Metric (in meters)	SocialGAN	Ours
(best) ADE	491.6	**281.2**
(best) FDE	975.5	**496.3**
(avg) ADE	698.3	**463.7**
(avg) FDE	1340.6	**814.6**

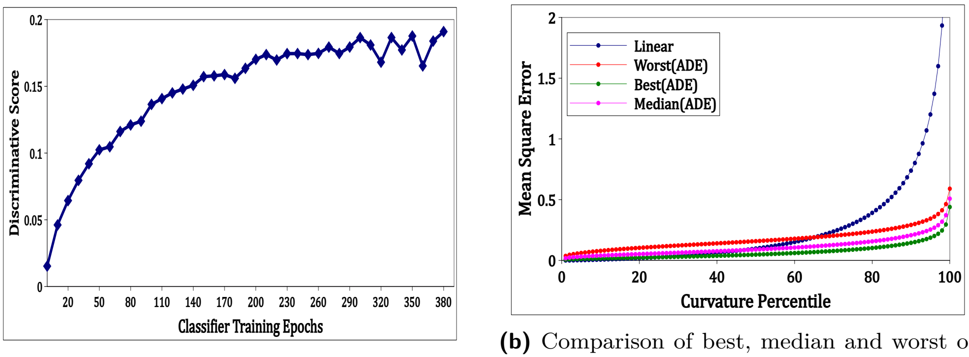
trajectory that is close to the actual ground truth trajectory. We also observe empirically that average ADE and FDE are different than the best ADE and FDE. This implies that there is diversity in predictions, which is incorporated by the MoN loss in (8).

- **Discriminative score** [33]: It is a well adopted measure to validate the quality of generated samples from a generator. Given a generator, we use a test trajectory dataset (which is not used in training the generator) with N trajectories, each of length $n+m$: $\{(\tau^{\text{prv}}_{v_1}, \tau^{\text{true}}_{v_1}), \ldots, (\tau^{\text{prv}}_{v_N}, \tau^{\text{true}}_{v_N})\}$. We generate (using our already trained generator) N future trajectories corresponding to each $\tau^{\text{prv}}_{v_i}$ for $i \in [N]$ to obtain $\{\widehat{\tau}_{v_i}, \ldots, \widehat{\tau}_{v_N}\}$. Then, we have a dataset of $2N$ trajectories, half of which are true trajectories $\{\tau^{\text{true}}_{v_1}, \ldots, \tau^{\text{true}}_{v_N}\}$ (labelled 1) and the other half generated using our generator $\{\widehat{\tau}_{v_i}, \ldots, \widehat{\tau}_{v_N}\}$ (labelled 0). We train a classifier on this dataset and measure its accuracy. A *perfect* generator would generate data indistinguishable from real ones and hence the classifier would have 50% accuracy. Any deviation from this 50% is a measure of how *inaccurate* the generator is. The discriminative score measures this deviation and is defined as abs$(0.5 - \mathit{accuracy})$. A lower discriminative score quantitatively indicates a better generator. Empirically, our generator achieves a low discriminative score, which implies that our generator generates trajectories that are representative of the typical vessel movement patterns found in the historical dataset.

Maritime data results. We divide the whole vessel trajectories data into training and testing set in a 80/20 ratio. Each vessel trajectory consists of 20 locations, first 10 locations (i.e., $n = 10$) are used as input to the model and next 10 locations (i.e., $m = 10$) as the labels. This corresponds to using last 10 mins of trajectory to generate trajectories for next 10 mins. We use the same number of model parameters for both SocialGAN and our approach, additional details on hyper-parameter settings are provided in supplementary material. Table 3 shows the ADE and FDE measures of both approaches. We observe empirically that in all four metrics, our approach is able to achieve better solution quality than SocialGAN. This result shows effectiveness of our proposed generative model on the maritime data.

Note that tanker and cargo vessels are about 200-300 meter in length. Therefore, ADE and FDE achieved by our approach are small relative to the size of tankers. Furthermore, best ADE/FDE in our case are quite different than the average ADE/FDE. This demonstrates that there is diversity in the generated trajectories.

Discriminative score. Our generator achieves a discriminative score of 0.19 as shown in Figure 3a. As the classifier is trained the score steadily increases but hits a plateau of 0.19. As reported in past work [33], 0.19 is competitive (better in some cases) with the scores obtained for other time series generation tasks. Having a low discriminative score ensures that the trajectories generated are realistic, and reflect typical movement patterns observed in the historical dataset. However, this doesn't necessarily imply that our samples are not



(a) Discriminative score to distinguish between real or fake trajectory sample. A score between (0.0 – 0.2) is reasonable.

(b) Comparison of best, median and worst of the generated trajectories against linear extrapolation as a function of increasing curvature in the real trajectories.

Figure 3

diverse. The FDE values for the generated trajectories differ by a significant amount as shown in Table 3; the average FDE is significantly higher than the best FDE. The same argument can be made for the metric ADE as well.

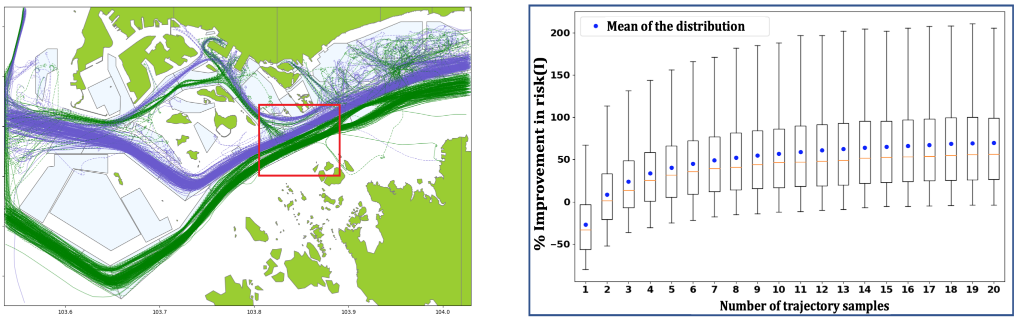
Varying curvatures. To demonstrate the robustness of our trajectory generator, we test it on trajectories with varying curvatures. We assign a curvature percentile to a trajectory where having a higher percentile implies that the vessel trajectory is more curved. In figure 3b, x-axis is curvature percentile, and y-axis is the average error of all trajectories in that curvature bucket. Results in figure 3b show that while the generator's performance is comparable to linear extrapolation in the case of low curvature percentiles (vessels almost moving in a straight line). It does much better with vessels that are changing their direction. Even the worst of the trajectory samples start doing better than the linear extrapolation as the curvature increases. This shows that our generator is a much better predictor in challenging scenarios when vessels are turning, than the current linear extrapolation method used by Singapore port's VTIS. We also emphasize that linear extrapolation is a very good metric in most cases, as large vessels typically are unable to turn sharply. Therefore, these results show that our generator has learned much better movements patterns found in the historical data than the linear prediction.

4.2 Path Planning

Here we present our experimental results for path planning module on the maritime data. The path planning part requires a generative model for generating trajectory samples. As empirically observed, SocialGAN performs worse on the maritime data. Therefore, we use our proposed model as the generative model for trajectory generation.

The evaluation of path planning part is mainly for close quarter scenarios where two or more vessels come very close to each other (less than 500 meters). This number (500 meters) was set after our discussions with maritime domain experts; however, it is configurable and does not affect our algorithmic methods. We use the objective in Table 2 as our evaluation criterion which essentially measures the minimum CPA between any any two vessels.

We use the naive formulation in Section 3.1 as baseline and refer to it as the naive solver. We refer to the improved formulation as the accelerated solver. This is our main proposed solver. We also test against the linearly extrapolated trajectories, which shows what would be the risk if vessels moved over this trajectory (linear extrapolation is the current prediction method Singapore port's VTIS uses).



(a) Ship routes in the Singapore strait with highlighted region in red is used as our planning region.

(b) Improvement (I) in risk of trajectories recommended by accelerated solver compared to historical trajectories using boxplots.

Figure 4

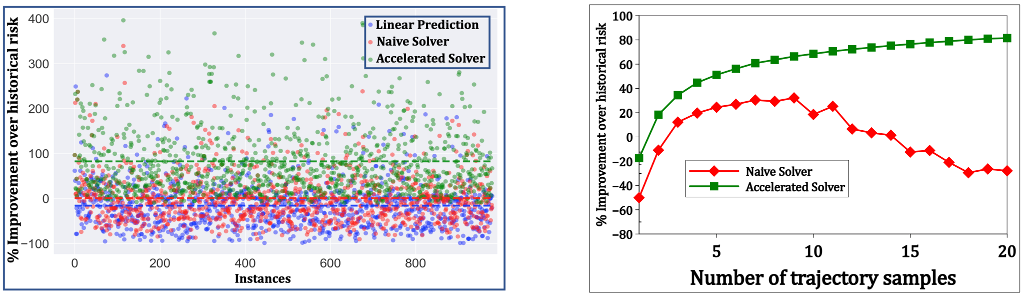
Planning instance generation. For each day we generate instances from peak hour period (7 AM - 9 AM) . Majority of close quarter incidents occur during this period. We first select a planning region near the port waters that has high traffic activity based on historical data as shown in Figure 4a. We also select an instance window of 20 minutes because our complete trajectory is of 20 locations at one minute intervals. So a planning instance includes a set of vessel trajectories that have at least one location present within the given planning region and the time window. A snapshot of a planning instance is shown in Figure 1b. For each instance we compute a risk value based on historical data as defined in Section 2. We evaluate our path planning approach on 1000 different instances with the highest risk.

Distributional result. In Figure 4b, we show distributional information about improvement of risk values in 1000 different instances . The x-axis denotes the number of samples (K) and y-axis shows percentage improvement of risk using accelerated solver. For a given instance the percentage improvement of risk (I) is given by

$$I = 100.\frac{\text{risk}(\tau_1^{\text{true}}, \ldots, \tau_M^{\text{true}}) - \left[\text{risk}(\tau_1^{\text{rec}}, \ldots, \tau_M^{\text{rec}})\right]}{abs(\text{risk}(\tau_1^{\text{true}}, \ldots, \tau_M^{\text{true}}))} \tag{13}$$

Note that risk, as defined in Equation 1, is always negative. Thus, the absolute value in the denominator is needed to show the percentage improvement [30]. We set optimization time limit to one minute to test the near real-time performance of the trajectory optimization module. We observe that the mean (in blue circle) values are higher than medians (orange) thus indicating a positively skewed distribution with long tail. The boxes cover the data range from 25th to 75th percentile. And the fences around the boxes cover the whole range of data. There exist some rare outliers where recommended trajectories are slightly worse off than historical trajectories. Based on our investigation, it was because a ship captain performed an atypical maneuver (such as taking sharp turns) which is rarely observed in the dataset. We also observe an overall good improvement (around 50%) of solution quality across 1000 instances starting from 7 samples. This result show robustness of our proposed accelerated solver across different instances in near real-time.

Different instances. In Figure 5a we demonstrate the performance of linear prediction, naive and accelerated solvers on 1000 different instances. The x-axis denotes instance id and y-axis denotes percentage improvement of risk compared to historical trajectories. Green color



(a) Comparison of linear prediction, naive and accelerated solver on improvement of risk compared to historical trajectories.

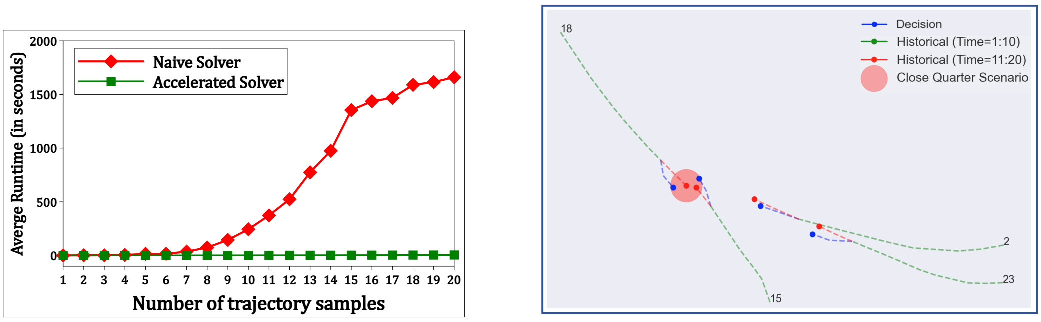
(b) Comparison of naive and accelerated solver on improvement of risk over historical trajectories with varying number of samples.

Figure 5

points denote the improvement using the accelerated solver, computed as per equation 13. Similarly, blue and red color denotes the improvement using the linear prediction and naive solver. We set an optimization time limit of one minute for solving each instance. We use a sample size of $K = 20$ for each trajectory. For all the instances, the accelerated solver achieves equal or better solution quality than both the linear prediction baseline and naive solver. This is because optimization time limit of 1 minute is limited for the naive solver to achieve good quality solution. On average the accelerated solver (green dotted line) achieves around 80% improvement of risk. Linear prediction in blue color perform poorly than both the solvers as it is just the linear extrapolation of data from previous time steps.

Varying samples. In our path planning optimization sample size is an important parameter. Therefore, in this experiment we test both the solvers with varying number of sample sizes. Results in Figure 5b show the comparison of naive and accelerated solver on solution quality with varying number of samples K. The x-axis denotes number of samples used in the optimization and y-axis denotes average percentage improvement of risk over historical data. The results shown are averaged over 15 instances. For this experiment also we set the optimization time limit to 1 minute. We observe that for accelerated solver solution quality improve with increasing number of samples, and quality does not change much after 15 samples. This is an expected result because at low number of samples the solution space is small. As the number of samples increase the solution space also increase which leads to better solution quality. But beyond a certain point there are upper limits to maximum possible distance between ships in a region with finite space. Thus the risk plateaus out with increasing number of samples. In this experiment also we observe that the accelerated solver is able to provide better solution quality than both historical data and the naive solver. In case of naive solver after about 7-8 samples the effect of optimization time limit kicks in. More number of samples would require longer optimization time to get the same solution quality, and thus we see a drop in solution quality. This experiment provides vital information about how to choose the sample size parameter in our approach.

Runtime comparison. Results in Figure 6a shows comparison of naive and accelerated solver on optimization time with varying number of samples K. The x-axis denotes number of samples and y-axis denotes average optimization runtime. The results shown are averaged over 20 instances. For this experiment, we set the optimality gap of the solver to 10%. We observe that runtime of naive solver rise almost exponentially with increasing number of



(a) Comparison of naive and accelerated solver on runtime with varying sample count.

(b) Close quarter scenario.

Figure 6

samples. However, accelerated solver is able to maintain a constant runtime irrespective of sample size. The solver has a runtime of around 4 seconds at 20 sample size. We also observed that accelerated solver is able to achieve a runtime of around 5 seconds at 20 sample size even for 5% optimality gap (not shown in the figure). For any system to be used in real-time scenario, a decision time of few seconds is very crucial. The empirical result shows our proposed system is well adapted for a real-time safe trajectory recommendation system.

Close quarter scenario. Figure 6b shows an instance of close quarter situation. Green and red dotted line denote previous and next 10-step trajectories from historical data respectively. Blue dotted line is the recommended trajectory from our learning and planning based system. It is one of the predicted samples from the generative model. In the figure vessels with id 15 and 18 are heading in opposite direction. They come very close to each other (less than 500 meters) which is a close quarter situation as highlighted in the big red circle. We observe that our recommended trajectories (in blue) are able to maintain a safe distance and thus avoid the close quarter incident. We provide four video files for such instances of close quarter situations using our maritime traffic simulator on our GitHub repo.

Close quarter scenario. Figure 7 shows qualitative results of some of the generated trajectories from our generative model. Trajectories in green and red are complete historical trajectories with time=1:10 and time=11:20 respectively. Trajectories in cyan color are generated sample future trajectories (time=11:20). Trajectories in blue color are the selected trajectories for time=11:20 from the path planning solver. Here we observe that the generated trajectories in cyan are a good representative sample of the historical trajectories in red.

5 Conclusion

We have presented a multiagent path planning approach to the problem of alleviating close quarter incidents in a highly congested maritime traffic environment. We proposed a data-driven based optimization methodology to the problem. We first learn a generative model of vessel movement behaviors from historical data. Empirically, we have shown the superior quality of our generative model over the baseline model. The trajectory samples generated from our model are then used in our proposed novel and efficient MILP solver to reduce close quarter incidents. Empirically, we have shown that our solver is able to provide high quality safe trajectory recommendations in near real-time in a variety of real-world close quarter situations mined from past data.

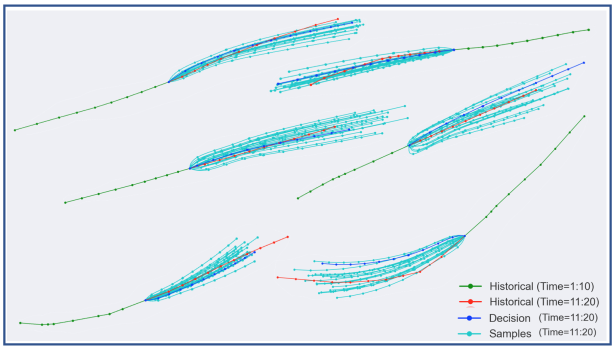


Figure 7 Qualitative result for generative model.

References

1 Lucas Agussurja, Akshat Kumar, and Hoong Chuin Lau. Resource-constrained scheduling for maritime traffic management. In *AAAI Conference*, 2018.

2 Alexandre Alahi, Kratarth Goel, Vignesh Ramanathan, Alexandre Robicquet, Li Fei-Fei, and Silvio Savarese. Social lstm: Human trajectory prediction in crowded spaces. In *IEEE conference on CVPR*, pages 961–971, 2016.

3 Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. An introduction to mcmc for machine learning. *Machine learning*, 50(1):5–43, 2003.

4 Saumya Bhatnagar, Akshat Kumar, and Hoong Chuin Lau. Decision making for improving maritime traffic safety using constraint programming. In *Proceedings of the 28th IJCAI*, 2019.

5 S. Choudhury, K. Solovey, M. J. Kochenderfer, and M. Pavone. Efficient large-scale multi-drone delivery using transit networks. In *IEEE ICRA*, pages 4543–4550, 2020.

6 Lei Du, Floris Goerlandt, and Pentti Kujala. Review and analysis of methods for assessing maritime waterway risk based on non-accident critical events detected from ais data. *Reliability Engineering and System Safety*, 200, 2020.

7 Haoqiang Fan, Hao Su, and Leonidas J Guibas. A point set generation network for 3d object reconstruction from a single image. In *IEEE conference on CVPR*, pages 605–613, 2017.

8 Futurenautics. Autonomous ships | white paper. `https://www.sipotra.it/old/wp-content/uploads/2017/05/Autonomous-Ships.pdf`, 2016.

9 Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

10 Agrim Gupta, Justin Johnson, Li Fei-Fei, Silvio Savarese, and Alexandre Alahi. Social gan: Socially acceptable trajectories with generative adversarial networks. In *IEEE Conference on CVPR*, pages 2255–2264, 2018.

11 Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

12 IMO. E-navigation. `https://www.imo.org/en/OurWork/Safety/Pages/eNavigation.aspx`, 2019.

13 International Maritime Organization. Electronic Nautical Charts (ENC) and Electronic Chart Display and Information Systems (ECDIS). `https://www.imo.org/en/OurWork/Safety/Pages/ElectronicCharts.aspx`, 2022.

14 Laura Leal-Taixé, Michele Fenzi, Alina Kuznetsova, Bodo Rosenhahn, and Silvio Savarese. Learning an image-based motion context for multiple people tracking. In *IEEE Conference on CVPR*, pages 3542–3549, 2014.

15 Namhoon Lee, Wongun Choi, Paul Vernaza, Christopher B Choy, Philip HS Torr, and Manmohan Chandraker. Desire: Distant future prediction in dynamic scenes with interacting agents. In *IEEE Conference on CVPR*, pages 336–345, 2017.

16 H. Ma, C. Tovey, G. Sharon, T. K. S. Kumar, and S. Koenig. Multi-agent path finding with payload transfers and the package-exchange robot-routing problem. In *AAAI Conference*, pages 3166–3173, 2016.

17 Faris Mokhtar. Busy shipping lane's narrow passageway hard for vessels to navigate. `https://www.todayonline.com/singapore/busy-shipping-lanes-narrow-passageway-hard-vessels-navigate`, 2017.

18 Robert Morris, Corina S. Pasareanu, Kasper Søe Luckow, Waqar Malik, Hang Ma, T. K. Satish Kumar, and Sven Koenig. Planning, scheduling and monitoring for airport surface operations. In *AAAI Workshop on Planning for Hybrid Systems*, 2016.

19 MPA. Vessel Traffic Information System. `https://www.mpa.gov.sg/web/portal/home/port-of-singapore/operations/vessel-traffic-information-system-vtis`, 2021.

20 MPA Singapore. Over 250 participate in Joint Oil Spill Exercise to Test Responsiveness to Oil Spills at Sea. `https://www.mpa.gov.sg/web/portal/home/media-centre/news-releases/mpa-news-releases/detail/091cd124-ca60-4f34-bdb6-a0967f82defd`, 2018.

21 International Maritime Organization. Autonomous shipping. `https://www.imo.org/en/MediaCentre/HotTopics/Pages/Autonomous-shipping.aspx`.

22 Stefano Pellegrini, Andreas Ess, and Luc Van Gool. Improving data association by joint modeling of pedestrian trajectories and groupings. In *European conference on computer vision*, pages 452–465. Springer, 2010.

23 Henrik Ringbom. Regulating autonomous ships – concepts, challenges and precedents. *Ocean Development & International Law*, 50(2-3):141–169, 2019.

24 Rolls-Royce. Remote and autonomous ship – The next steps. `https://www.rolls-royce.com/~/media/Files/R/Rolls-Royce/documents/customers/marine/ship-intel/aawa-whitepaper-210616.pdf`, 2016.

25 David Silver. Cooperative pathfinding. In *AIIDE*, pages 117–122, 2005.

26 Arambam James Singh, Akshat Kumar, and Hoong Chuin Lau. Hierarchical multiagent reinforcement learning for maritime traffic management. In *Proceedings of the 19th AAMAS*, 2020.

27 Arambam James Singh, Duc Thien Nguyen, Akshat Kumar, and Hoong Chuin Lau. Multiagent decision making for maritime traffic management. In *AAAI Conference*, 2019.

28 Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Roman Barták, and Eli Boyarski. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *SoCS*, pages 151–159, 2019.

29 Teck-Hou Teng, Hoong Chuin Lau, and Akshat Kumar. Coordinating vessel traffic to improve safety and efficiency. In *Proceedings of the 16th AAMAS*, pages 141–149. ACM, 2017.

30 Leo Törnqvist, Pentti Vartia, and Yrjö O Vartia. How should relative changes be measured? *The American Statistician*, 39(1):43–46, 1985.

31 Kevin Varley. Ships Queues Worsen Port Delays From Singapore to Piraeus. `https://www.bloomberg.com/news/articles/2021-11-02/ships-queues-worsen-port-delays-from-singapore-to-piraeus`, 2021.

32 Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine*, 29(1):9–20, 2008.

33 Jinsung Yoon, Daniel Jarrett, and Mihaela van der Schaar. Time-series generative adversarial networks. *Advances in Neural Information Processing Systems*, 32:5508–5518, 2019.

34 J. Yu and S. M. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *AAAI Conference*, pages 1443–1449, 2013.

35 Jinfen Zhang, Tiago A Santos, C Guedes Soares, and Xinping Yan. Sequential ship traffic scheduling model for restricted two-way waterway transportation. *Proceedings of the Institution of Mechanical Engineers, Part M: Journal of Engineering for the Maritime Environment*, 231(1):86–97, 2017.

Acquiring Maps of Interrelated Conjectures on Sharp Bounds

Nicolas Beldiceanu
IMT Atlantique, LS2N (TASC), Nantes, France

Jovial Cheukam-Ngouonou
IMT Atlantique, LS2N (TASC), Nantes, France, and Université Laval, Québec, Canada

Rémi Douence
IMT Atlantique, LS2N, Inria, (Gallinette), Nantes, France

Ramiz Gindullin
IMT Atlantique, LS2N (TASC), Nantes, France

Claude-Guy Quimper
Université Laval, Québec, Canada

Abstract

To automate the discovery of conjectures on combinatorial objects, we introduce the concept of a *map of sharp bounds* on characteristics of combinatorial objects, that provides a set of interrelated sharp bounds for these combinatorial objects. We then describe a Bound Seeker, a CP-based system, that gradually acquires maps of conjectures. The system was tested for searching conjectures on bounds on characteristics of digraphs: it constructs sixteen maps involving 431 conjectures on sharp lower and upper-bounds on eight digraph characteristics.

2012 ACM Subject Classification Computing methodologies → Heuristic function construction; Mathematics of computing → Combinatorial optimization

Keywords and phrases Acquisition of conjectures, digraphs, bounds

Digital Object Identifier 10.4230/LIPIcs.CP.2022.6

Supplementary Material *Software (Source Code)*: `https://github.com/cquimper/MapSeekerCP2022`
archived at `swh:1:dir:e25840f81f3be49d17b827efeab9a5a285595703`

Funding *Nicolas Beldiceanu*: partially founded by the EU-funded ASSISTANT project no. 101000165.
Jovial Cheukam-Ngouonou: founded by the ANR AI@IMT project and by Laval University.
Ramiz Gindullin: founded by the EU-funded ASSISTANT project.

Acknowledgements Thanks to Hervé Grall for his participation in the definition of the map concept, and to Samir Loudni and Helmut Simonis for their comments on a preliminary version of this paper.

1 Introduction

Research on conjectures making systems in the context of discrete mathematics is a topic that goes back to the late 1950s and the 1980s [8, 14, 32] and got renewed interest [20, 21, 29, 31]. Within CP, some initial research on the generation of implied constraints was done by Charley et al. [11] and the most recent work focuses on model and constraint acquisition [4, 7, 10, 19, 27, 28] rather than on conjecture making. Within OR, Hansen's AutoGraphiX system [1, 17] focuses on finding unrelated bounds using Variable Neighbourhood Search.

Four reasons motivate our work: (i) to highlight that CP can contribute to the automatic discovery of conjectures, (ii) to systematically search sharp bounds on characteristics of objects that show up in combinatorial problems, (iii) to stress the need to develop conjecture discovery programs that build up a body of strongly interrelated knowledge rather than unrelated conjectures as it has been the case so far, (iv) by the fact that bounds are an essential feature of branch-and-bound methods in optimisation but also a weakness of CP [16, 22]: the development of sharp bounds that consider several interrelated characteristics is still a

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 6; pp. 6:1–6:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

manual process [3, 6]. Our approach is unique among all works for conjectures generation, as the result is not a set, but rather a *graph of conjectures*, linked by projection (i.e. variable elimination) operators. Our contributions are:

- We introduce the concept of *map of sharp bounds* as a set of interrelated conjectures providing sharp lower and upper-bounds wrt the characteristic of a combinatorial object.
- For each conjecture on a sharp bound, the map gives some *extremal characteristics* i.e., the characteristic values common to all combinatorial objects achieving the bound.
- By introducing secondary characteristics and by permitting the use of common sub-expressions in a polynomial, as well as simple Boolean and conditional formulae, we tend to produce *explainable conjectures.* This also reveals *unified conjectures* across different subsets of characteristics.
- We demonstrate the usefulness of CP for acquiring such maps: using digraphs as combinatorial objects, the system produces 431 conjectures distributed in 16 maps obtained from 8 characteristics combined with lower and upper bounds. It retrieves a set of known results, enhances some known bounds, and comes up with new conjectures, some of which we proved to be true.

The significance of maps is twofold. Beyond sharp bounds, a map brings together the relations between several sharp bounds and the structure of combinatorial objects reaching each bound under the same edifice. A map can be used to test the mutual consistency of independently acquired bounds by verifying that one bound can be derived from another bound.

In Sect. 2, we introduce the concept of a *map* that presents a set of conjectures for sharp bounds and their logical relations. In Sect. 3.1, we provide the workflow of our acquisition system. We introduce, in Sect. 3.2, a parameterised CP conjecture generator. We evaluate the produced conjectures in Sect. 4, discuss related work in Sect. 5, and conclude in Sect. 6.

2 Conjectures map as a symbolic piece of knowledge

After providing an informal overview of maps of conjectures, and a first example of a map, we motivate, define and illustrate the map concept. Then we show how the use of secondary characteristics permits both acquiring formulae sharing common sub-expressions, and sometimes come up with the same bound for different subsets of input characteristics.

Informal overview of maps. Consider digraphs as an example of combinatorial objects. It is well known that any digraph $\mathcal{G}$ satisfies the following invariant: the number of arcs a of $\mathcal{G}$ is less than or equal to the square of the number of vertices v^2 of $\mathcal{G}$, and the maximum value v^2 is only reached when the number of vertices of the smallest connected component of $\mathcal{G}$ is equal to v, i.e. $\mathcal{G}$ consists of a single connected component of v vertices.

We are interested in systematically generating such candidate invariants, a.k.a. conjectures, for a richer set of characteristics, e.g. the number of connected components c of $\mathcal{G}$, the number $\underline{c}$ of vertices of the smallest connected component of $\mathcal{G}$.

Our conjectures have one of the following forms: (i) sharp bounds of a digraph characteristic wrt other digraph characteristics, e.g. $a \leq v^2$, or (ii) implication showing that, when a sharp bound is reached, some characteristics are fixed or functionally determined by some other characteristics, e.g. $a = v^2 \Rightarrow c = 1$, and $a = v^2 \Rightarrow \underline{c} = v$.

Finally, we are interested in connecting sharp bounds, revealing that the right-hand side of an implication of type (ii) can be used to eliminate a characteristic of a sharp bound and retrieve a sharp bound with one less characteristic. For instance, replacing $\underline{c}$ by v in the sharp bound $a \leq \underline{c}^2 + (v - \underline{c})^2$, we retrieve the sharp bound $a \leq v^2$. We call these different conjectures and the links connecting sharp bounds "map".

A first example of map. As an example of combinatorial objects, we use in this paper digraphs with these characteristics: the number v of vertices, the number a of arcs, the number c (resp. s) of connected components (resp. strongly connected components), the number $\underline{c}$ (resp. $\overline{c}$) of vertices of the smallest (resp. largest) connected component, the number $\underline{s}$ (resp. $\overline{s}$) of vertices of the smallest (resp. largest) strongly connected component. To compare the bounds obtained by the Bound Seeker with the database of invariants of the global constraint catalogue, see Sect.4.3 of [2], we assume that each vertex of a digraph has at least one incoming or outgoing arc.

▶ **Example 1.** Fig. 1 illustrates the map concept with a map containing three conjectures labelled as ❶, ❷, and ③:

- Two conjectures about the sharp bounds ❶ $a \leq (v-(c-1))^2+(c-1)$, and ❷ $a \leq v^2$ on the maximum number of arcs a in a digraph $\mathcal{G}$ wrt the number of vertices v, and the number of connected components c of $\mathcal{G}$.
- The conjecture ③ of node (B) indicates that the bound v^2 is reached only when $c = 1$.

The arrow going from node (A) to node (B) is labelled by ③ as the bound v^2 is obtained by replacing c by 1 in the bound $(v-(c-1))^2+(c-1)$. The leftmost and rightmost parts of Fig. 1 show, in brown, two digraphs achieving these bounds.

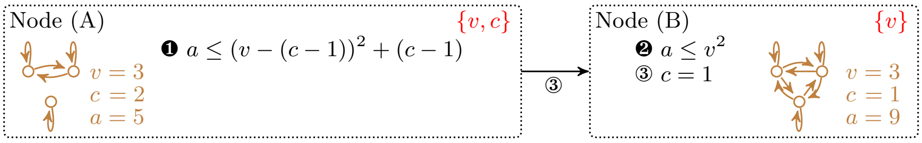


Figure 1 Map of two sharp bounds on the maximum number of arcs of a digraph.

In this paper, all maps of conjectures are presented in the same way as the map in Fig. 1: (i) the upper left corner of a node gives a *node label* in black, (ii) the upper right corner provides the *parameters* used in the sharp bound of this node in red, (iii) a dark label of the form ❶ refers to the *sharp bound* itself, (iv) a light label of the form ① designates an *equation* which must hold to reach the sharp bound given in (iii), (v) a brown illustration shows a *witness to the sharpness of the bound.* Finally, an arrow from a first node to a second node indicates which equation(s) in the second node should be used to substitute some parameters used in the first node's bound to retrieve the bound given in the second node. For space reasons, some large maps, e.g. Fig 4, may omit the elements (i) and (v).

Motivating and defining the concept of map. We introduce the concept of a *map of conjectures* as a way to reveal the links between a set of conjectures related to sharp bounds for a characteristic of a combinatorial object. Our goal is to describe conjectures on sharp bounds of characteristics of a combinatorial object, e.g. a digraph, a tree, and to organise these conjectures into a single structure, a *map of sharp bounds*, which (i) systematically interconnects these conjectures, and which (ii) describes the structure of the combinatorial objects for which the bounds are reached. In the map in Fig. 1, we consider for digraphs three characteristics, a, v and c for the number of arcs, of vertices, and of connected components.

▶ **Definition 2.** *Given a finite set of input characteristics $\mathcal{P}$ and an output characteristic* $\mathrm{o} \notin \mathcal{P}$, *a* map of sharp upper bounds $\mathcal{M}_{\mathcal{P}}^{\mathrm{o}\leq}$ *is defined as a digraph where:*

- *Each node of the map is associated with a subset $P \subseteq \mathcal{P}$ of input characteristics and corresponds to a* maximum conjecture *of the form $\mathrm{o} \leq f(P)$. This inequality is tight, i.e. there exist values that can be given to the parameters P in order to reach the equality.*

In addition, a node contains maximality conjectures*, at most one per characteristic q in the complement of P wrt $\mathcal{P}$, represented by the symbolic equality $q = g_q(P)$, where g_q is a function defined over realisable parameters values of P and called a* maximum characterisation*, and expressing the following property: for any combination of parameters P reaching the maximum $f(P)$, the characteristic q is equal to $g_q(P)$.*

- *Each arc from conjecture $\mathrm{o} \leq f_i(P_i)$ to conjecture $\mathrm{o} \leq f_j(P_j)$ corresponds to a projection from a subset P_i of input characteristics to a subset P_j of input characteristics, by eliminating a characteristic $q_{i,j}$, i.e. $P_j = P_i \setminus \{q_{i,j}\}$. The arc is labelled with an equality $q_{i,j} = g_{q_{i,j}}(P_j)$ where $g_{q_{i,j}}(P_j)$ is the value given to $q_{i,j}$ to reach the equality in the conjecture $\mathrm{o} \leq f_j(P_j)$. The equality $q_{i,j} = g_{q_{i,j}}(P_j)$ is called a* maximality conjecture*.*

In a map, there is a single output characteristic that we bound using the other characteristics called input characteristics. The output characteristic is the *bounded characteristic*, while the input characteristics are the *bounding characteristics*. While the maximum conjecture provides a bound on the output characteristic wrt the characteristics in P, the maximality conjectures indicate the values taken by the characteristics not in P when the bound is reached. Similarly to $\mathcal{M}^{\mathrm{o}\leq}_{\mathcal{P}}$, a map $\mathcal{M}^{\mathrm{o}\geq}_{\mathcal{P}}$ provides a collection of sharp lower bounds as a set of minimum conjectures of the form $\mathrm{o} \geq f_j(P_j)$, and a set of minimality conjectures.

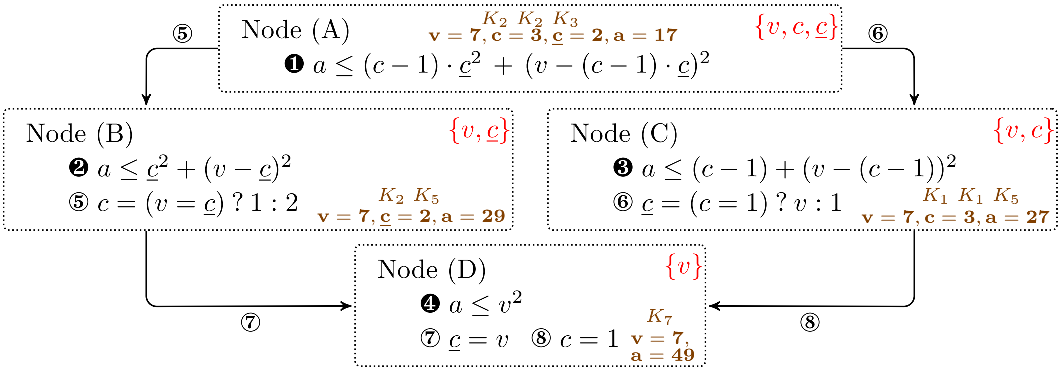

⑤
Node (A)
$K_2\ K_2\ K_3$
$\mathbf{v = 7, c = 3, \underline{c} = 2, a = 17}$
$\{v, c, \underline{c}\}$
⑥
❶ $a \leq (c-1)\cdot\underline{c}^2 + (v-(c-1)\cdot\underline{c})^2$
Node (B)
$\{v, \underline{c}\}$
❷ $a \leq \underline{c}^2 + (v-\underline{c})^2$
⑤ $c = (v = \underline{c})\ ?\ 1 : 2$
$K_2\ K_5$
$\mathbf{v = 7, \underline{c} = 2, a = 29}$
Node (C)
$\{v, c\}$
❸ $a \leq (c-1) + (v-(c-1))^2$
⑥ $\underline{c} = (c = 1)\ ?\ v : 1$
$K_1\ K_1\ K_5$
$\mathbf{v = 7, c = 3, a = 27}$
Node (D)
$\{v\}$
❹ $a \leq v^2$
⑦ $\underline{c} = v$ ⑧ $c = 1$
K_7
$\mathbf{v = 7,}$
$\mathbf{a = 49}$
⑦
⑧

Figure 2 Map $\mathcal{M}^{a\leq}_{\{v,c,\underline{c}\}}$ with the sharp upper-bounds ❶, ❷, ❸, ❹ for the number of arcs in a digraph; each node presents an example in brown: given a value for the characteristics attached to the node, a graph reaching the maximum is described, as a union of cliques K_i, with i vertices, e.g. in node (B), given the assignments $v = 7$ and $\underline{c} = 2$, the digraph with 2 cliques K_2, K_5 reaches the maximum 29 for the number a of arcs; *cond* ? $x : y$ denotes x if condition *cond* holds, y otherwise.

▶ **Example 3** (Extending Ex. 1 to a map of four nodes)**.** Fig. 2 presents Map $\mathcal{M}^{a\leq}_{\{v,c,\underline{c}\}}$, where we consider the following characteristics of digraphs: as input characteristics, the number v of vertices, the number c of connected components, and the number $\underline{c}$ of vertices of the smallest connected component; as output characteristic, the number a of arcs. In Map $\mathcal{M}^{a\leq}_{\{v,c,\underline{c}\}}$, there are four nodes, corresponding to the subsets $\{v, c, \underline{c}\}$, $\{v, \underline{c}\}$, $\{v, c\}$ and $\{v\}$, shown in red, whereas the power set of $\{v, c, \underline{c}\}$ contains eight subsets. For the four other subsets, namely $\{c, \underline{c}\}$, $\{\underline{c}\}$, $\{c\}$ and $\emptyset$, no conjecture can be found, as the number of arcs is not upper bounded wrt these characteristics. In the nodes (A), (B), (C) and (D), the items labelled with ❶, ❷, ❸ and ❹ indicate a maximum conjecture wrt the number a of arcs, while the elements marked with ⑤, ⑥, ⑦ and ⑧ show maximality conjectures wrt c and $\underline{c}$. For instance, in Node (B), the maximum conjecture ❷ $a \leq \underline{c}^2 + (v-\underline{c})^2$ really means: among all digraphs with v nodes and whose smallest component contains $\underline{c}$ nodes, the digraph with most arcs has exactly $\underline{c}^2 + (v-\underline{c})^2$ arcs. Each arc is labelled with a maximality conjecture giving the value of the characteristic that is eliminated. For instance, from Node (A) to Node (B), the characteristic c that is eliminated from ❶ satisfies this maximality conjecture ⑤: when the maximum of number of arcs is reached, the value of c is 1 if $v = \underline{c}$, 2 otherwise.

Capturing more bounds with secondary characteristics. As the number of input characteristics grows, the bound formulae can get rather complicated. Consequently, we introduce a set $\mathcal{A}$ of auxiliary characteristics to obtain simpler formulae. Examples of such auxiliary characteristics are, for instance, (i) $c_{>1}$, (ii) $s_{>1}$, and (iii) $c_{\in\{2,3\}}$ which correspond to (i) the number of connected components with more than one vertex, (ii) to the number of strongly connected components with more than one vertex, and (iii) to the number of connected components with two or three vertices and for which all strongly connected components have only one vertex. Also initially introduced when searching for lower bounds on the number of arcs, such characteristics have proved useful for many other bounds. We introduce the notion of secondary characteristics of the node of a map, which will be illustrated in Ex. 5 and 6.

▶ **Definition 4.** *Given a node of a map that is associated to a subset $P \subseteq \mathcal{P}$ of input characteristics, to an output characteristics* o*, to a maximum conjecture of the form* $\mathrm{o} \leq f(P)$*, and a set of auxiliary characteristics $\mathcal{A}$, the* set of secondary characteristics *of the node is defined as the characteristics of the set $\mathcal{A} \cup (\mathcal{P} - P - \{\mathrm{o}\})$ which are functionally determined by the set P when $\mathrm{o} = f(P)$.*

To test that a secondary characteristic is functionally determined by P, we check for each generated combination of values for P that the value of the secondary characteristic is unique. This test is performed while generating our dataset used for acquiring conjectures.

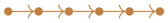
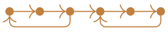
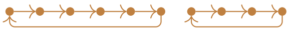
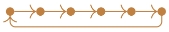
To find bounds that exploit these secondary characteristics, we use a multi-level approach: (i) first, we look for a formula for each secondary characteristic; (ii) then we try to catch a sharp bound also considering the secondary characteristics for which we could find a formula. Both in (i) and (ii) a formula can either use input characteristics and secondary characteristics for which we already found a formula. As a result, we obtain formulae that are easier to interpret, as we can associate a straightforward meaning to the sub-terms that appear in a bound. Ex. 5 illustrates this point.

▶ **Example 5** (Bound expressed wrt several secondary characteristics)**.** This example shows the only lower bound found by the Bound Seeker on the number of arcs a of a digraph $\mathcal{G}$ wrt the size $\overline{c}$ of its largest connected component and the size $\underline{s}$ of its smallest strongly connected component. We have $\mathcal{P} = \{v, a, c, \underline{c}, \overline{c}, s, \underline{s}, \overline{s}\}$, the bound parameters $P = \{\overline{c}, \underline{s}\}$, the output characteristic $\mathrm{o} = a$, and the auxiliary characteristics $\mathcal{A} = \{c_{>1}, s_{>1}\}$. All potential secondary characteristics $\mathcal{A} \cup (\mathcal{P} - P - \{\mathrm{o}\}) = \{v, c, \underline{c}, s, \overline{s}, c_{>1}, s_{>1}\}$ are functionally determined by $\overline{c}$ and $\underline{s}$. The lower bound found by the Bound Seeker is $a \geq s_{>1} - c_{>1} + v$ with:

- $s_{>1} = \min(-\underline{s} + \overline{c} + 1, 2 \cdot (\underline{s} \geq 2))$,
- $c_{>1} = (\overline{c} = c\,?\,0 : c)$, where $c = 1 + (((\overline{c} - 2 \cdot \underline{s}) \leq 0) \wedge ((\overline{c} \mod \underline{s}) \geq 1))$,
- $v \;\; = ((\overline{c} - \underline{c}) = 0\,?\,\overline{c} : \overline{c} + \underline{c})$, where $\underline{c} = ((2 \cdot \underline{s} - \overline{c}) \leq 0\,?\,\overline{c} : \underline{s})$,

where a Boolean expression such as $(\underline{s} \geq 2)$ is used as an integer, i.e. either 0 for false or 1 for true. While the main formula $s_{>1} - c_{>1} + v$ is simple, it uses a secondary characteristic $s_{>1}$ which is expressed directly wrt $\overline{c}$ and $\underline{s}$, and two other secondary characteristics $c_{>1}$ and v which mention the two extra secondary characteristics c and $\underline{c}$ for which two formulae involving only $\overline{c}$ and $\underline{s}$ could be found. The occurrence of Boolean expressions reflects slight variations in the structure of *witness digraphs*, i.e. digraphs reaching a sharp bound, as shown in Table 1.

Table 1 Digraphs minimising the number of arcs for four values of the bound parameters $\overline{c}$ and $\underline{s}$.

$\overline{c}$	$\underline{s}$	a	v	c	$\underline{c}$	$c_{>1}$	$s_{>1}$	witness digraph	$s_{>1}-c_{>1}+v$
6	1	5	6	1	6	1	0		$0-1+6$
6	3	7	6	1	6	1	2		$2-1+6$
6	4	10	10	2	4	2	2		$2-2+10$
6	6	6	6	1	6	1	1		$1-1+6$

Within a same map, expressing bounds in terms of secondary characteristics may reveal a same bound formula for several subsets of input characteristics. We observed this phenomenon in the majority of the acquired maps. Ex. 6 illustrates this for the acquired map giving the upper bound on the number of vertices of the largest connected component of a digraph.

▶ **Example 6** (Map example illustrating how bounds can be unified by using secondary characteristics)**.** In the appendix, Fig. 4 depicts the maximum and maximality conjectures of the map $\mathcal{M}^{\overline{c}\le}_{\{v,c,\underline{c},s,\underline{s},\overline{s}\}}$ found by the Bound Seeker for the upper-bound on the size of the largest connected component $\overline{c}$ with the related links. Note that v needs to be an input characteristic, as otherwise the upper-bound of $\overline{c}$ is unbounded. Part (A) shows the 16 bounds found when using only the input characteristics: these bounds are defined by 5 maximum conjectures ❶,. . . ,❺ and 4 maximality conjectures ⑥,. . . ,⑨. Each link illustrates how a maximum conjecture is projected onto an other maximum conjecture via a maximality conjecture: e.g., the link ❺ $\xrightarrow{\text{⑦}}$ ❶ shows how the bound ❺ $\overline{c}\le\underline{s}-c\cdot\underline{s}+v$ is rewritten as ❶ $\overline{c}\le v$ as we have ⑦ $c=1$. Part (B) shows the bounds found when also using the secondary characteristics r and $\underline{c}$, where r is a secondary characteristic corresponding to $v-c\cdot\underline{c}$. We only have 2 maximum conjectures ❶ $\overline{c}\le v$ and ❷ $\overline{c}\le r+\underline{c}$, where r and $\underline{c}$ are defined by the 5 maximality conjectures ③,. . . ,⑦ shown on Part (B). The natural upper-bound of $\overline{c}$ is the number of vertices of the digraph (see ❶), unless c or $\underline{c}$ are part of the input characteristics (see ❷), which requires to consider the feasibility conditions induced by the use of such inputs.

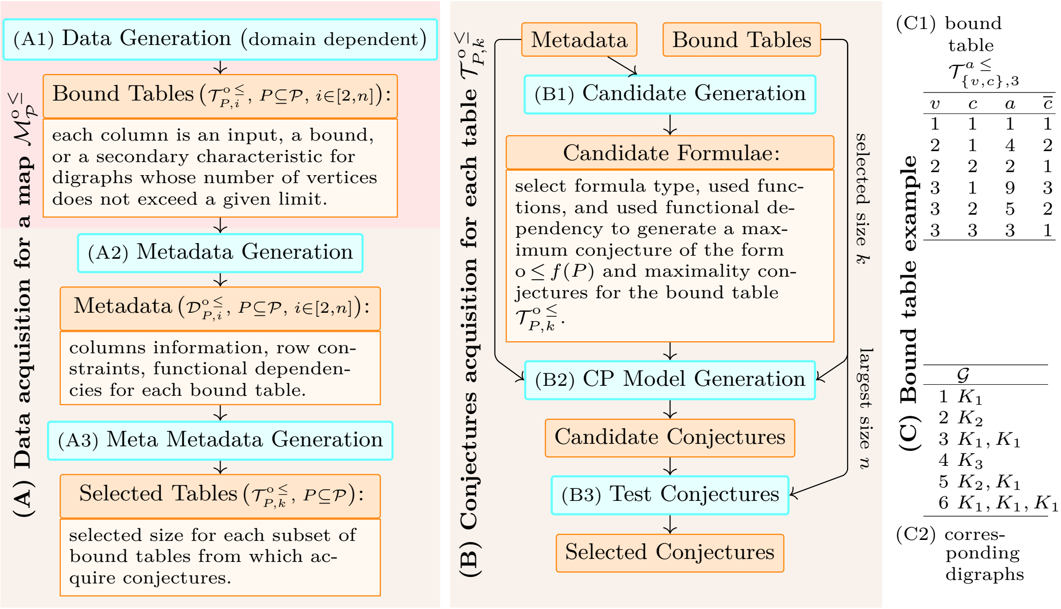
Missing arcs are due to the lack of functional dependencies. For instance, in Part (A), we have no arc from $\{v,s\}$ to $\{v\}$, as the number of strongly connected components s is not functionally determined by the number of vertices v when the sharp bound ❶ is reached, i.e. when $\overline{c}=v$: e.g., for $\overline{c}=v=2$ we both have $s=2$ and $s=1$ as shown by and .

3 A Bound Seeker

3.1 Overview of the map acquisition system

Parts (A) and (B) of Fig. 3 gives the different phases for generating a map: software components are shown in cyan and labelled with capital letters, while data is displayed in orange. We now detail the phases (A1), (A2), (A3), (B1), (B2), and (B3). To illustrate each phase, we use the bound table $\mathcal{T}^{a\le}_{\{v,c\},3}$ provided in Part (C1) of Fig. 3.

(A1) Generating data. To learn valid conjectures for any digraph of at most k vertices, we produce all parameter combinations of interest for digraphs up to a maximum number n of vertices. An exhaustive generation of such data is not a problem, as a program is used for this purpose. However, the issue is to select the appropriate value of k, neither too small to create invalid conjectures for digraphs with more than k vertices, nor too large to limit


(A) Data acquisition for a map $\mathcal{M}_{\mathcal{P}}^{o\leq}$
(A1) Data Generation (domain dependent)
Bound Tables ($\mathcal{T}_{P,i}^{o\leq}$, $P\subseteq\mathcal{P}$, $i\in[2,n]$):
each column is an input, a bound, or a secondary characteristic for digraphs whose number of vertices does not exceed a given limit.
(A2) Metadata Generation
Metadata ($\mathcal{D}_{P,i}^{o\leq}$, $P\subseteq\mathcal{P}$, $i\in[2,n]$):
columns information, row constraints, functional dependencies for each bound table.
(A3) Meta Metadata Generation
Selected Tables ($\mathcal{T}_{P,k}^{o\leq}$, $P\subseteq\mathcal{P}$):
selected size for each subset of bound tables from which acquire conjectures.
(B) Conjectures acquisition for each table $\mathcal{T}_{P,k}^{o\leq}$
Metadata
Bound Tables
(B1) Candidate Generation
Candidate Formulae:
select formula type, used functions, and used functional dependency to generate a maximum conjecture of the form $o\leq f(P)$ and maximality conjectures for the bound table $\mathcal{T}_{P,k}^{o\leq}$.
(B2) CP Model Generation
Candidate Conjectures
(B3) Test Conjectures
Selected Conjectures
selected size k
largest size n
(C) Bound table example
(C1) bound table $\mathcal{T}_{\{v,c\},3}^{a\leq}$
(C2) corresponding digraphs

v	c	a	$\bar{c}$
1	1	1	1
2	1	4	2
2	2	2	1
3	1	9	3
3	2	5	2
3	3	3	1

	$\mathcal{G}$
1	K_1
2	K_2
3	K_1, K_1
4	K_3
5	K_2, K_1
6	K_1, K_1, K_1

Figure 3 Workflow in the Bound Seeker: (A) data and (B) conjecture acquisition phases; Phase (A1) with a red background depends on the combinatorial objects we consider (digraphs in our case), while Phases (A2), (A3), ..., (B3) are domain independent; (C1) example of an upper-bound table for digraphs of at most 3 vertices with the *input characteristic* v, c, the *output characteristics* a, and the *secondary characteristic* $\bar{c}$ corresponding to the number of vertices of the largest connected component; (C2) digraphs corresponding to each entry of the bound table shown in (C1).

the number of generated constraints to acquire the conjectures in Phase (B2). To this end, Phase (A1) produces a table $\mathcal{T}$ with the characteristics values for digraphs of at most n vertices in such a way that the size of the table $\mathcal{T}$ does not exceed a given memory limit. With this table $\mathcal{T}$, Phase (A1) extracts for each i between 2 and n, for each subset of input characteristics P of $\mathcal{P}$, and each output characteristic o, a bound table $\mathcal{T}_{P,i}^{o\leq}$ based only on the entries of $\mathcal{T}$ corresponding to digraphs with at most i vertices. Each row of a bound table represents a feasible combination of values for P, with the corresponding bound value for o, and the values of the secondary characteristics.

Unlike all the next steps, Phase (A1) depends on the type of combinatorial objects for which we generate conjectures. For digraphs, our data generation phase uses a CP model to produce a set of bound tables that is used by the acquisition process. As illustrated in Part (C1) of Fig. 3, the bound table $\mathcal{T}_{\{v,c\},3}^{a\leq}$ provides a sharp upper bound of the output characteristic a wrt the input characteristics v and c. A bound table may also mention secondary characteristics, e.g. $\bar{c}$ in $\mathcal{T}_{\{v,c\},3}^{a\leq}$, which are functionally determined by the input characteristics. Each column of the table $\mathcal{T}_{\{v,c\},3}^{a\leq}$ refers to a characteristic, i.e. v, c, a, $\bar{c}$, while each row corresponds to a combination of parameter values for v, c with the associated maximum number of arcs a and the value of the secondary characteristic $\bar{c}$.

(A2) Generating metadata. For each bound table $\mathcal{T}_{P,i}^{o\leq}$ (with $P\subseteq\mathcal{P}$ and $i\in[2,n]$), with *nrows* rows, where $\mathcal{T}_{P,i}^{o\leq}[r,j]$ denotes the value of the r-th row and the j-th column, Phase (A2) calculates the aggregated information $\mathcal{D}_{P,i}^{o\leq}$ (with $P\subseteq\mathcal{P}$ and $i\in[2,n]$) used to select the size k employed when searching for the conjectures of the subset P and the output characteristics o, such as:

Table 2 Examples of candidates formulae and corresponding generated formulae for the bound table $\mathcal{T}_{\{v,c\},3}^{a\leq}$ in Part (C1) of Fig 3.

Candidate formulae generated by Phase (B1)	Formulae found by Phase (B2)
polynomial of degree 1 parameterised by v and c to determine $\overline{c}$	$\overline{c} = v - c + 1$
polynomial of degree 1 parameterised by v and c to determine a	none
polynomial of degree 2 parameterised by v and c to determine a	$a = c^2 - 2 \cdot v \cdot c + v^2 - c + 2 \cdot v$
polynomial of degree 1 parameterised by v and $\overline{c}$ to determine a	none
polynomial of degree 2 parameterised by v and $\overline{c}$ to determine a	$a = \overline{c}^2 - \overline{c} + v$

- The minimum/maximum values of each column and the number of distinct values.
- The minimal functional dependencies [24] that determine in the table $\mathcal{T}_{P,k}^{\mathrm{o}\leq}$ the output characteristic and the secondary characteristics. Each functional dependency gives a subset of characteristics that functionally determine another characteristic. For instance, in the bound table $\mathcal{T}_{\{v,c\},3}^{a\leq}$, columns a and $\overline{c}$ are functionally determined by columns v and c. But column a is also functionally determined by columns v and $\overline{c}$.
- Binary constraints between two distinct columns i and j of the table $\mathcal{T}_{P,k}^{\mathrm{o}\leq}$, i.e. constraints of the form $\forall r \in [1, \mathit{nrows}], \mathcal{T}_{P,k}^{\mathrm{o}\leq}[r,i]\ \mathit{op}\ \mathcal{T}_{P,k}^{\mathrm{o}\leq}[r,j]$ (with $\mathit{op} \in \{\leq, <, >, \geq\}$). In $\mathcal{T}_{\{v,c\},3}^{a\leq}$ we have for each row that the number of vertices is greater than or equal to the number of connected components, i.e. $v \geq c$, and similarly $v \geq \overline{c}$, $a \geq v$, $a \geq c$, $a \geq \overline{c}$.

Such knowledge is used to focus the search for conjectures: first by selecting promising subsets of input parameters for a formula, and second by providing information that avoids producing meaningless formulae. For instance, we do not generate a formula with a term $\min(v, c)$ as $v \geq c$ is true. The generated metadata is also the input of the next phase.

(A3) Generating meta metadata to find the relevant size of the training dataset. Based on the information computed by Phase (A2), Phase (A3) determines for the subset P and the output characteristic o, the size k used when searching for conjectures. To select the size k in the datasets $\mathcal{T}_{P,i}^{\mathrm{o}\leq}$ (with $i \in [2, n]$) from which we acquire the conjectures, we operate as follows. As a functional dependency or a binary constraint of a table $\mathcal{T}_{P,i}^{\mathrm{o}\leq}$ may become invalid for a table $\mathcal{T}_{P,j}^{\mathrm{o}\leq}$ with $j > i$, we identify the smallest size k from which the set of minimal functional dependencies and the set of binary constraints of the tables $\mathcal{T}_{P,k}^{\mathrm{o}\leq}, \dots, \mathcal{T}_{P,n}^{\mathrm{o}\leq}$ remain identical. In practice, for space reason, we generated digraphs with up to $n = 26$ vertices. To avoid overfitting when the number of rows of table $\mathcal{T}_{P,k}^{\mathrm{o}\leq}$ is too small, we select the smallest size corresponding to the table with at least 200 rows: on average, conjectures were produced using digraphs with up to 18 vertices.

(B1) Generating candidate formulae. This phase generates for a bound table $\mathcal{T}_{P,k}^{\mathrm{o}\leq}$, partially instantiated candidate formulae to acquire the corresponding maximal and maximality conjectures. Given the parameters P, the output characteristic o, the set of secondary characteristics of the selected bound table $\mathcal{T}_{P,k}^{\mathrm{o}\leq}$, Phase (B1) produces on request the next candidate formula to find a conjecture. The set of potential characteristics that the formula may mention, and the formula itself, are restricted by the functional dependencies and the binary constraints that were identified by the metadata generation phase. Table 2 shows some candidates formulae that are successively produced for table $\mathcal{T}_{\{v,c\},3}^{a\leq}$.

(B2) Generating a CP model linking a parameterised formula with the data. This phase uses a candidate formula generated by Phase (B1) to post an equational constraint for each entry in a bound table $\mathcal{T}^{\text{o}\leq}_{P,k}$ to obtain a formula where all input parameters and coefficients are fixed and thus produce a conjecture. Phase (B2) queries Phase (B1) for the next candidate parameterised formula, tries to instantiate it, and asks again for a next candidate formula. To find a value for each coefficient of a candidate formula, we use a constraint model to link a candidate formula to (i) the functional dependencies and binary constraints identified by the metadata generation phase, and (ii) all the bound table entries of the selected size. Many constraints break different symmetry types and force all sub-terms of a formula to be meaningful. The second column of Table 2 shows for each candidate formula the corresponding concrete formula found by the CP model.

(B3) Testing the candidate conjectures. This last phase tests the validity of the conjectures against the largest bound table $\mathcal{T}^{\text{o}\leq}_{P,n}$, i.e. against the largest available generated dataset.

3.2 A constraint approach for acquiring symbolic equations

The search for sharp bounds leads to the identification of equations in which the left-hand side is an output or a secondary characteristic, and the right-hand side is a formula involving input and secondary characteristics. As already noted in the introduction of [9] and in the conclusion of [18], the space of candidate formulae constitutes a major challenge for equation discovery methods. Rather than applying a bottom-up approach that generates formulae of increasing complexity, we adopt the following strategy. As we aim at finding simple formulae, we use three complementary classes of formulae that turned out to appear concomitantly in a map: (1) Boolean formulae involving k arithmetic conditions linked by a single commutative logical operator or by a sum, (2) simple conditional formulae, and (3) formulae over polynomials that can share common sub-expressions. A first attempt to use only polynomials without common sub-expressions missed some formulae, e.g. see Ex. 5, and quite often provided too complicated formulae, as illustrated in Ex. 8. Based on the metadata introduced in Sect. 3.1, we will present a CP approach for restricting the space of formulae: for space reasons, we focus on polynomials sharing common sub-expressions.

3.2.1 A parameterised candidate formulae generator for Phase (B1)

Formula syntax. All conjectures we generate have the form *characteristic op formula*, where *op* is one of the comparison operators $\leq$, $=$, $\geq$, and *formula* is a formula involving a set of characteristics. Consequently, formulae are described by the following set of simplified grammar rules, where "Small Capitals" indicates a non-terminal symbol, "Roman" denotes a function or a known constant, "*Italic*" highlights a (digraph) characteristics, "**Bold**" denotes an unknown integer constant. Within these rules, polynomial(Params, degree) denotes a polynomial whose maximum degree is fixed (with degree > 0) on a non-empty subset of parameters of its potential parameters Params, and the functions geq0(x), geq(x, y), sum_consec(x), cmod(x, y), dmod(x, y) resp. stand for 1 if $x \geq 0$ otherwise 0, 1 if $x \geq y$ otherwise 0, $\frac{x\cdot(x+1)}{2}$, $x - (y \bmod x)$, $x - (x \bmod y)$.

$$\begin{array}{lll}
\textsc{Formula} & ::= \ \mathbf{cst} \mid \textsc{Bool} \mid \mathbf{cst} + \textsc{Bool} \mid \textsc{Cond} \mid \textsc{Pol} \mid \textsc{PolBinary} \mid \textsc{PolUnary} & \\
\textsc{Bool} & ::= \textsc{BoolOp}(\textsc{BoolConds}) & \textsc{BoolOp} ::= \ \wedge \mid \vee \mid = \mid + \\
\textsc{BoolConds} & ::= \textsc{BoolCond}, \textsc{BoolConds} \mid \textsc{BoolCond} & \\
\textsc{BoolCond} & ::= \textsc{Param}\ \textsc{Cmp}\ \mathbf{cst} & \textsc{Cmp} ::= \leq \mid = \mid \geq \mid \neq \\
\textsc{Cond} & ::= (\textsc{BoolCond}\ ?\ \textsc{ParamCst} : \textsc{ParamCst}) & \textsc{ParamCst} ::= \textsc{Param} \mid \mathbf{cst}
\end{array}$$

Pol ::= polynomial(Params, degree)

PolBinary ::= Bf(Pol, Pol) Bf ::= min | max | floor | mod | cmod | dmod | prod

PolUnary ::= Uf1(Pol) | Uf2(Pol, **cst**)

Uf1 ::= geq0 | sum_consec Uf2 ::= min | max | floor | mod | power

Params ::= Param* Param ::= Char|BTerm|UTerm Char ::= $v|c|\underline{c}|\overline{c}|s|\underline{s}|\overline{s}$

BTerm ::= Bt(Char, Char)

UTerm ::= sum_consec(Char) | Ut(Char, **cst**) | Char ∈ [**cst**, **cst**]

Bt ::= min | max | floor | ceil | mod | cmod | dmod | prod

Ut ::= min | max | floor | ceil | mod | geq | power

▶ **Example 7** (Examples of generated Boolean, conditional, polynomial formulae).

- $(\overline{s}=1)\wedge(\overline{c}\in[2,3])$ and $(v=\underline{c})=(c=1)$, where the 2nd formula denotes a condition that is satisfied only if both conditions $(v=\underline{c})$ and $(c=1)$ are true, or both false.
- $(\underline{s}=1\ ?\ \lceil\frac{v}{2}\rceil\ :\ v)$ and $((\overline{c}-\underline{c})=0\ ?\ \underline{c}\ :\ \underline{c}+\overline{c})$, where $(cond\,?\,x\ :\ y)$ denotes x if the condition $cond$ holds, y otherwise.
- $(v \bmod \overline{c})^2-\overline{c}\cdot(v \bmod \overline{c})+v\cdot\overline{c}$ where $v \bmod \overline{c}$ is a shared binary term BTerm, $\lfloor\frac{(\overline{s}\geq 2)+\overline{s}+v}{2}\rfloor$ where $(\overline{s}\geq 2)$ is a unary term UTerm of the form geq$(\overline{s},2)$.

▶ **Example 8** (Finding simpler bounds using Boolean and conditional formulae). We illustrate with an example generated by the system on the lower bound of the number of arcs a wrt the size of the smallest and largest connected components $\underline{c}$ and $\overline{c}$, and the size $\overline{s}$ of the largest strongly connected component, how using Boolean and conditional formulae often leads to simpler conjectures. Without using Boolean and conditionals, we get $a\geq s_{>1}-c_{>1}+v$ with $s_{>1}=\min(\overline{s}-1,1)$, $c_{>1}=\min(\min(\underline{c},2),\min(\underline{c},2)+\overline{c}-\underline{c}-1)$, and $v=\min(\overline{c}+\underline{c},\underline{c}\cdot\overline{c}-\underline{c}^2+\overline{c})$; enabling Boolean and conditional formulae, we get the simpler bound: $a\geq s_{>1}-c_{>1}+v$ with $s_{>1}=(\overline{s}\geq 2)$, $c_{>1}=(\underline{c}\geq 2)+((\overline{c}-\underline{c})\geq 1)$, and $v=((\overline{c}-\underline{c})=0\,?\,\underline{c}:\underline{c}+\overline{c})$.

Candidate formulae generator. Since we want to try out a variety of formulae, we create a parameterised candidate formulae generator, which, upon backtracking, proposes a new candidate formula with non-fixed coefficients; these are variables for the constants and for the input characteristics that will be used in a candidate formula. In this generator we specify:

- The structure of the formula, that is whether we use (1) a Boolean formula, (2) a simple conditional formula, or (3) a formula over polynomials; in this later case we also specify how many unary and binary terms occur in each polynomial.
- The arithmetic functions we may use in the terms.
- The complexity of a polynomial, that is its potential maximum degree, its maximum number of non-zero coefficients, the ranges of its coefficients.
- The list of possible combinations of characteristics that the candidate formula can use in its parameters. Such combinations correspond to functional dependencies identified by the metadata generation phase, i.e. Phase (A2).

We use more than one generator to design a formula generation policy where the simplest candidate formulae are tried first.

3.2.2 Constraint model for acquiring a conjecture for formulae over polynomials for Phase (B2)

Given a candidate formula $\mathcal{F}$, (corresponding either to Pol, to PolBinary, or to PolUnary as described in the set of grammar rules in Sect. 3.2.1), for which the set of used parameters is partially determined, and for which the coefficients are not yet fixed, we create a constraint model that relates these unknowns to all rows in a bound table. Our model includes four types of constraints, namely (i) structural constraints on the input and secondary characteristics that will be used in $\mathcal{F}$, (ii) symmetry-breaking constraints, (iii) constraints preventing the generation of formulae in which a term could be simplified, and (iv) equational constraints on each row of a bound table. We describe the model variables, the constraints on the characteristics used in $\mathcal{F}$, the constraints on the unary/binary terms and binary function of $\mathcal{F}$, and the equational constraints on the table entries. The number of variables and constraints of the model is linear wrt the number of table entries as it is dominated by the equational constraints. For reasons of space, concerning the constraints of the type (ii) and (iii), we will only detail the constraints related to the min function.

Variables used in the model. Table 3 introduces the variables used to represent a non-constant formula $\mathcal{F}$ involving at most n_c characteristics (i.e. input and secondary characteristics), n_u unary terms, n_b binary terms, and n_p polynomials, wrt a bound table $\mathcal{T}$ of nrows rows. We use n as a shortcut for $n_c + n_u + n_b$. For the binary term $\mathcal{B}_i$, the variables $\mathit{B\_IND1}_i$, $\mathit{B\_IND2}_i$, $\mathit{B\_O}_i$ designate a term with the arguments $(C_{\mathit{B\_IND1}_i}, C_{\mathit{B\_IND2}_i})$ when $\mathit{B\_O}_i = 0$, and $(C_{\mathit{B\_IND2}_i}, C_{\mathit{B\_IND1}_i})$ otherwise. When the binary term is commutative, e.g. min, the order of the arguments is irrelevant and $\mathit{B\_O}_i$ will be set to 0 (see constraint (4.c) in Table 4), but otherwise, e.g. mod, the order matters.

Table 3 Variables of the model, where n_{cu} is an abbreviation of the term $n_c + n_u$.

Objects	Variables	Comments
Characteristics $\mathcal{C}_j$ $(j \in [1, n_c])$	$C_j \in \{0,1\}$	$C_j = 1$ iff $\mathcal{C}_j$ used by formula $\mathcal{F}$
Unary term $\mathcal{U}_i$ $(i \in [1, n_u])$	$U_{i,j} \in \{0,1\}$ $(j \in [1, n_c])$	$U_{i,j} = 1$ iff $\mathcal{C}_j$ used by $\mathcal{U}_i$
	$\mathit{U\_IND}_i \in [1, n_c]$	index of the used characteristics
	$\mathit{U\_MIN}_i$	minimum value of the used characteristics
	$\mathit{U\_MAX}_i$	maximum value of the used characteristics
	$\mathit{U\_CST}_i$	constant used in $\mathcal{U}_i$
$(r \in [1, \mathit{nrows}])$	$\mathit{U\_VAL}_{i,r}$	value of term $\mathcal{U}_i$ wrt r-th row and the j-th column (with $U_{i,j} = 1$) of table $\mathcal{T}$
Binary term $\mathcal{B}_i$ $(i \in [1, n_b])$	$B_{i,j} \in \{0,1\}$ $(j \in [1, n_c])$	$B_{i,j} = 1$ iff $\mathcal{C}_j$ used by $\mathcal{B}_i$
	$\mathit{B\_IND1}_i \in [1, n_c]$	index of first used characteristic
	$\mathit{B\_IND2}_i \in [1, n_c]$	index of second used characteristic
	$\mathit{B\_O}_i \in \{0,1\}$	order of used characteristics in arguments
$(r \in [1, \mathit{nrows}])$	$\mathit{B\_VAL}_{i,r}$	value of term $\mathcal{B}_i$ wrt r-th row, the $\mathit{B\_IND1}_i$-th, and the $\mathit{B\_IND2}_i$-th columns of table $\mathcal{T}$
Polynomial $\mathcal{P}_i$ of degree d_i	$P_{i,j} \in \{0,1\}$ with $j \in [1, n]$ and $n = n_c + n_u + n_b$	$\begin{cases} P_{i,j} = 1, j \in [1, n_c] \Rightarrow \mathcal{C}_j \text{ used by } \mathcal{P}_i \\ P_{i,j} = 1, j \in [n_c + 1, n_{cu}] \Rightarrow \mathcal{U}_{j-n_c} \text{ used by } \mathcal{P}_i \\ P_{i,j} = 1, j \in [n_{cu} + 1, n] \Rightarrow \mathcal{B}_{j-n_c-n_u} \text{ used by } \mathcal{P}_i \end{cases}$
$(i \in [1, n_p])$	$M_{i,k}$ $(k \in [1, \binom{n+d_i}{d_i}])$	$M_{i,k}$ is the k-th coefficient of $\mathcal{P}_i$, the coefficient with the largest k is the constant
$(r \in [1, \mathit{nrows}])$	$\mathit{P\_VAL}_{i,r}$	value of polynomial $\mathcal{P}_i$ wrt r-th row of table $\mathcal{T}$

Constraints on the structure of the formula. The upper part of Table 4 lists the constraints, (i) specifying which characteristics the formula $\mathcal{F}$ uses, i.e. see (1a), (ii) forcing a unary term, a binary term, and a polynomial to use the appropriate number of characteristics, i.e. see (2a), (3a) and (4a), (iii) connecting the characteristics used by the unary and binary terms with the characteristics used in the polynomials and the formula, i.e. see (5a), (6a), (iv) restricting non-zero coefficients of polynomials, i.e. see (7a), (8a).

Table 4 (**Top**) Constraints on the structure of a formula $\mathcal{F}$; FD_TABLE is the list of characteristics combinations that may be used by $\mathcal{F}$, created by the candidate formulae generator, while *maxz* is the maximum number of non-zero coefficients of a polynomial. (**Mid**) Constraints on a unary term $\mathcal{U}_i$ (with $i \in [1, n_u]$), where u_{f_i} is the function assigned to $\mathcal{U}_i$, min_j (with $j \in [1, n_c]$), is the smallest value of the j-th characteristic. (**Bottom**) Constraints on a binary term $\mathcal{B}_i$ (with $i \in [1, n_b]$), where b_{f_i} is the function assigned to $\mathcal{B}_i$, and TABLE_UNORDERED is the set of pairs of characteristics indices such that the 1st characteristic is not always smaller, or greater, than 2nd characteristic; char. is an abbreviation for characteristic.

Constraints	Comments
(1a) $\text{TABLE}(\langle C_1, \dots, C_c\rangle, \text{FD\_TABLE})$	restrict the char. used in $\mathcal{F}$
(2a) $\forall i \in [1, n_u] : \sum_{j=1}^{j=n_c} U_{i,j} = 1$	$\mathcal{U}_i$ uses 1 char.
(3a) $\forall i \in [1, n_b] : \sum_{j=1}^{j=n_c} B_{i,j} = 2$	$\mathcal{B}_i$ uses 2 char.
(4a) $\forall i \in [1, n_p] : \sum_{j \in [1,n]} P_{i,j} \geq 1$	$\mathcal{P}_i$ uses at least one char., or at least one unary or binary term
(5a) $\forall j \in [1, n_c] :\ C_j = \bigvee_{i \in [1,n_u]} U_{i,j} \vee \bigvee_{i \in [1,n_b]} B_{i,j} \vee \bigvee_{i \in [1,n_p]} P_{i,j}$	link $U_{i,j}$, $B_{i,j}$, and $P_{i,j}$ to C_j
(6a) $\forall j \in [n_c + 1, n] : \sum_{i \in [1,n_p]} P_{i,j} > 0$	force each unary/binary term to be used by at least 1 polynomial
(7a) $\forall i \in [1, n_p] : (\sum_{k=1}^{k<\binom{n+d_i}{d_i}} [M_{i,k} \neq 0]) > 0$	polynomials are not constant
(8a) $\forall i \in [1, n_p] : (\sum_{k=1}^{k\leq\binom{n+d_i}{d_i}} [M_{i,k} \neq 0]) \leq maxz$	each polynomial has a maximum number of non-zeros coefficients
(1b) $\text{ELEMENT}(U\_IND_i, \langle U_{i,1}, \dots, U_{i,n_c}\rangle, 1)$	get index of used char.
(2b) $\text{ELEMENT}(U\_IND_i, \langle min_1, \dots, min_{n_c}\rangle, U\_MIN_i)$	get min. value of used char.
(3b) $\text{ELEMENT}(U\_IND_i, \langle max_1, \dots, max_{n_c}\rangle, U\_MAX_i)$	get max. value of used char.
(4b) $u_{f_i} \in \{\min\} \Rightarrow \left\{ \begin{array}{l} U\_CST_i > U\_MIN_i \\ U\_CST_i < U\_MAX_i \end{array} \right.$	cannot simplify unary term $\mathcal{U}_i$, as otherwise could remove $\mathcal{U}_i$
(1c) $\text{ELEMENT}(B\_IND1_i, \langle B_{i,1}, \dots, B_{i,n_c}\rangle, 1)$	get index of first used char.
(2c) $\text{ELEMENT}(B\_IND2_i, \langle B_{i,1}, \dots, B_{i,n_c}\rangle, 1)$	get index of second used char.
(3c) $B\_IND1_i < B\_IND2_i$	indexes are ordered
(4c) $b_{f_i} \in \{\min\} \Rightarrow B\_O_i = 0$	fix order of the 2 arguments as min is a commutative function
(5c) $b_{f_i} \in \{\min\} \Rightarrow \text{TABLE}\left(\begin{array}{l} \langle B\_IND1_i, B\_IND2_i\rangle, \\ \text{TABLE\_\_UNORDERED} \end{array} \right)$	assign two char.whose values are not ordered

Constraints on unary/binary terms and on a binary function. Within Table 4, constraint (1b) (resp. (1c), (2c)), links the 0-1 variables $U_{i,j}$ (resp. $B_{i,j}$) to the index of the characteristic involved in the term. To avoid generating unary terms of the form $\min(\mathit{Characteristic}, \mathit{Cst})$ which could just be rewritten as *Characteristic* or as *Cst*, constraint (4b) restricts the minimum and maximum values of the constant. When using the min function in a binary term, constraint (4c) avoids generating equivalent binary terms whose arguments are permuted. Constraint (5c) prevents generating a binary term when the min could be simplified, e.g. avoids generating $\min(\underline{c}, \overline{c})$ as the metadata information found in Phase (A2) indicates that $\underline{c}$ is always smaller than or equal to $\overline{c}$. Finally, when the candidate formula $\mathcal{F}$ is a binary function corresponding to min, that uses the polynomials $\mathcal{P}_1$ and $\mathcal{P}_2$ of degree d, we post the lexicographic ordering constraint $\langle M_{1,1}, \dots, M_{1,\binom{n+d}{d}}\rangle <_{\text{LEX}} \langle M_{2,1}, \dots, M_{2,\binom{n+d}{d}}\rangle$ between the monomial coefficients of $\mathcal{P}_1$ and $\mathcal{P}_2$. Note that, for space reason, besides constraints (4b), (4c), and (5c), we omit in Table 4 the symmetry and simplification constraints related to functions that are different from min.

Equational constraints. For each row r of the bound table $\mathcal{T}$ we post some constraints linking the selected characteristics $\mathcal{C}_j$ with (i) the value variable $U\_VAL_{i,r}$ of each unary term $\mathcal{U}_i$, (ii) the value variable $B\_VAL_{i,r}$ of each binary term $\mathcal{B}_i$, and (iii) the value variable $P\_VAL_{i,r}$ of each polynomial $\mathcal{P}_i$. For each row r we also post an equality constraint linking the value of the candidate formula $\mathcal{F}$ on row r with the corresponding bound value on the same row. Finally, for a binary function min between two polynomial $\mathcal{P}_1$ and $\mathcal{P}_2$, we impose that for at least one of the entries of the bound table the value of $\mathcal{P}_1$ is strictly less than the value of $\mathcal{P}_2$ on the same entry, and that the converse applies for another entry of the table. To avoid unnecessarily complex formulae, we minimise the sum of the absolute values of the coefficients of a candidate formula $\mathcal{F}$.

4 Evaluation of the Bound Seeker

We focus on constructing 16 maps on the lower and upper-bounds of the number of vertices, the number of arcs, the number of connected (resp. strongly connected) components, and their minimum and maximum sizes. The components of the system are written in SICStus Prolog and consist of 10000 lines of code for the Data Generation, the Metadata Generation, the Meta Metadata Generation, the Candidate Formulae Generation, the CP Model Generation, and the Test phase. The Data Generation phase generates a total of 1944 bound tables (occupying 2 Gb) for each maximum number of digraph vertices ranging from 2 to 26; each bound table gives the lower or upper-bound of a characteristic wrt different subsets of input characteristics. We evaluate the Bound Seeker from several standpoints:

- The percentage of conjectures that, while acquired from the size selected by the Meta Metadata, still hold for all entries of the largest generated bound tables, i.e. the tables of digraphs containing up to 26 vertices.
- The percentage of bounds from the database of invariants in [2] that was retrieved (resp. not found).
- Besides the conjectures retrieved from the global constraint catalogue database, we manually proved ten new conjectures. Using WolframAlpha, we also checked the consistency of 105 projections of a sharp bound B_1 onto a sharp bound B_0 involving one less input characteristic, by substituting in B_1 the input characteristic to be eliminated, by the expression defined by the corresponding maximality conjecture.

As the complexity of a formula increases with the number of input characteristics, we limit our evaluation to up to 3 input characteristics. All experiments to acquire the conjectures for the 16 maps were done using the same system parameters, i.e. none of the components have been tuned manually to behave differently depending on the considered map. Out of 350 (resp. 202) combinations of input characteristics for which the Bound Seeker tried to find a sharp lower (resp. upper) bound, using only polynomials, it got at least one sharp bound for 279 (resp. 149) combinations of characteristics, as well as 1236 (resp. 975) minimality (resp. maximality) conjectures. Using also Boolean and conditional expressions it found 3 extra lower bounds and 93 new maximality/minimality conjectures. Table 5 provides the results for the 16 maps using SICStus 4.6.0 on a 2015 iMac with a 4 GHz Core i7 and 32Gb of memory: for each map, we give the number of formulae found using only polynomials (see col. #P1), then using Boolean, conditional, and polynomial (see col. #B2, #C2 and #P2). Using Boolean and conditional expressions generates 3.8% new formulae compared to when using polynomials alone; moreover, 31.07% of the formulae that use polynomials are replaced by simpler formulae that use Boolean or conditionals expressions. The time spent is explained by a significant number of candidate formulae tested, as it comes from the

Table 5 Number of minimum/maximum and minimality/maximality conjectures found for each of the 16 maps and time in min. using only polynomials (see only Poly), and using Booleans, conditionals and polynomials (see Bool/Cond/Poly).

Maps	only Poly		Bool/Cond/Poly				Maps	only Poly		Bool/Cond/Poly			
	#P1	Time	#B2	#C2	#P2	Time		#P1	Time	#B2	#C2	#P2	Time
$\mathcal{M}_{\mathcal{P}}^{c\geq}$	259	257	47	25	194	533	$\mathcal{M}_{\mathcal{P}}^{c\leq}$	100	218	9	17	77	439
$\mathcal{M}_{\mathcal{P}}^{\overline{c}\geq}$	129	230	0	13	120	476	$\mathcal{M}_{\mathcal{P}}^{\overline{c}\leq}$	153	193	32	31	94	542
$\mathcal{M}_{\mathcal{P}}^{\overline{s}\geq}$	97	130	0	8	90	306	$\mathcal{M}_{\mathcal{P}}^{\overline{s}\leq}$	102	392	15	31	60	999
$\mathcal{M}_{\mathcal{P}}^{a\geq}$	367	1248	38	102	255	3180	$\mathcal{M}_{\mathcal{P}}^{a\leq}$	384	2505	46	84	264	5939
$\mathcal{M}_{\mathcal{P}}^{\underline{c}\geq}$	63	167	10	27	27	388	$\mathcal{M}_{\mathcal{P}}^{\underline{c}\leq}$	130	223	16	14	102	457
$\mathcal{M}_{\mathcal{P}}^{\underline{s}\geq}$	43	54	0	18	25	226	$\mathcal{M}_{\mathcal{P}}^{\underline{s}\leq}$	48	171	1	8	40	365
$\mathcal{M}_{\mathcal{P}}^{s\geq}$	263	474	37	31	198	813	$\mathcal{M}_{\mathcal{P}}^{s\leq}$	93	100	5	17	73	267
$\mathcal{M}_{\mathcal{P}}^{v\geq}$	294	368	30	75	205	1570	$\mathcal{M}_{\mathcal{P}}^{v\leq}$	14	7	0	2	12	90

Table 6 Comparing the conjectures on the bounds found by the Bound Seeker (BS) with the database of invariants of the global constraint catalogue (GCC).

Number of input characteristics	1	2	3	**Total**	**Percentage**
Number of equivalent sharp bounds retrieved by BS	22	14	4	40	**66,66%**
Number of sharper bounds than the GCC found by BS	1	3	0	4	**6,66%**
Number of generalised sharp bounds found by BS	0	6	0	6	**10%**
Number of erroneous bounds found in the GCC by BS	1	1	1	3	**5%**
Number of bounds in the GCC not retrieved by BS	0	0	7	7	**11,66%**
Total bounds of the GCC per column	24	24	12	60	

combination of minimal functional dependencies and grammar rules. Moreover, arithmetic constraints like div and mod with multiple occurrences of the same variable are handled poorly by CP solvers. The datasets used in the experiments and the sixteen maps found will be available for download in a technical report.

Evaluation of the acquired conjectures wrt the largest data sets. Of the 3625 conjectures acquired when only using polynomials, we found 5 invalid conjectures when tested against all samples of the largest data set, i.e. all digraphs up to 26 vertices. Of the 3264 conjectures acquired when also using Boolean and conditional expressions, we found 16 invalid conjectures. Note that in this setting the Bound Seeker does not try to find polynomial formulae if it already found a Boolean or a conditional formula.

Comparing the conjectures founds with proved bounds of the constraint catalogue. As shown in Table 6, the Bound Seeker retrieves 66.66% of the bounds of the constraint catalogue, even if the resulting formulae have sometimes a different form: e.g., the upper-bound on the number of arcs a wrt the number of vertices v, connected components c, and strongly connected components s in the catalogue is expressed as $a \leq c-1+(v-s+1)\cdot(v-c+1)+\lfloor\frac{(s-c+1)\cdot(s-c)}{2}\rfloor$, while the Bound Seeker finds the equivalent inequality $a \leq \lfloor\frac{r^2+\overline{s}^2+v\cdot\underline{c}+r-\overline{s}+v}{2}\rfloor$, with $\underline{c} = \max(2\cdot v - v\cdot c, 1)$, $\overline{s} = v - s + 1$ and $r = v\cdot[c \geq 2] - c\cdot[c \geq 2]$; r is a secondary characteristic corresponding to $v - c\cdot\underline{c}$. Unlike the bound given by [2], the bound found by the Bound Seeker defines the size $\underline{c}$ of the smallest connected component, and the size $\overline{s}$ of the largest strongly connected component of those extreme digraphs for which the upper-bound is reached.

An example of a generalised bound found by the Bound Seeker is the lower bound $a \geq ((v - \overline{c}) \leq 1\,?\, \max(v-1,1) \,:\, v-2)$, with $v = (\underline{c} = \overline{c}\,?\,\underline{c} \,:\, \underline{c} + \overline{c})$ which extends the catalogue bound $\underline{c} \neq \overline{c} \;\Rightarrow\; a \geq \underline{c} + \overline{c} - 2 + (\underline{c} = 1)$. An example of correct bound found by the Bound Seeker replacing the erroneous bound (i) $a \geq v - \lfloor \frac{s-1}{2} \rfloor$ of the catalogue is (ii) $a \geq v - c_{\in\{2,3\}}$ with $c_{\in\{2,3\}} = (v = s\,?\,\lfloor \frac{v}{2} \rfloor \,:\, \lfloor \frac{s-1}{2} \rfloor)$: for the edge condition $v = s = 2$, (i) returns 2, rather than 1 as (ii) does. Bound (ii) $a \geq v - c_{\in\{2,3\}}$ can be interpreted as follows: to minimise the number of arcs, one has to maximise the number of connected components of the form $\bullet\!\rightarrow\!\bullet$, $\bullet\!\rightarrow\!\bullet\!\rightarrow\!\bullet$, $\bullet\!\rightarrow\!\bullet\!\leftarrow\!\bullet$ or $\bullet\!\leftarrow\!\bullet\!\rightarrow\!\bullet$. The missing bounds of the catalogue are partially explained by the limited complexity of the common subexpressions (see BTerm, UTerm in Sect. 3.2.1) of our polynomials, and by the lack of some secondary characteristics.

5 Related work

While there exist several discovery programs in the context of mathematics devoted to set theory, number theory, finite algebra and knot theory [12, 23, 13], only a few systems focus on finding bounds between characteristics of a combinatorial object. The two most notable systems are S. Fajtlowicz's Graffiti program [14] and P. Hansen's AutoGraphiX system [1, 17]. The first difference is that the Bound Seeker attempts to systematically construct a set of sharp bounds on all possible combinations of a set of input characteristics. The second main difference is that the Bound Seeker introduces secondary characteristics and searches for key properties of extreme combinatorial objects for which the bounds are reached.

In slightly different domains, recent work in CP uses machine learning techniques to estimate the domain boundaries of an objective function [30] of an optimisation problem. Some other work uses CP to extract equations from a spreadsheet [18, 25], and some recent work investigates how to integrate integer programming solvers within neural networks [15, 26].

The specificity of our approach compared to machine learning and constraint acquisition [7] is twofold: (i) we can generate our input data, but we need to ensure that these data contain the correct values of the sharp bounds we consider, as otherwise, we would necessarily obtain wrong maximal conjectures; moreover, maximality conjectures only make sense for sharp bounds; (ii) we have to learn concise conjectures that fit perfectly to all available data, as minimising an error measure would be irrelevant for acquiring conjectures on sharp bounds.

6 Conclusion

We introduce a structure that connects a set of sharp bounds. Based on this structure, we propose a constructive approach to acquire a set of interrelated conjectures on sharp bounds. We show the relevance of using a variety of types of formulae, i.e., Boolean, conditionals, and polynomials with shared sub-expressions, to acquire simpler conjectures. This work opens a new application domain for CP for automated conjectures-making systems. It creates a new line of research to those already reported in a recent survey on machine learning for combinatorial optimisation [5].

References

1 Mustapha Aouchiche, Gilles Caporossi, Pierre Hansen, and M. Laffay. Autographix: a survey. *Electron. Notes Discret. Math.*, 22:515–520, 2005. doi:10.1016/j.endm.2005.06.090.

2 Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global Constraint Catalog, 2nd Edition (revision a). Technical Report T2012-03, Swedish Institute of Computer Science, 2012. Available at http://ri.diva-portal.org/smash/get/diva2:1043063/FULLTEXT01.pdf.

3 Nicolas Beldiceanu, Mats Carlsson, Jean-Xavier Rampon, and Charlotte Truchet. Graph invariants as necessary conditions for global constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2005.

4 Nicolas Beldiceanu and Helmut Simonis. A Model Seeker: Extracting Global Constraint Models from Positive Examples. 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 141–157. Springer, 2012. doi:10.1007/978-3-642-33558-7_13.

5 Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *Eur. J. Oper. Res.*, 290(2):405–421, 2021.

6 Christian Bessière, Emmanuel Hebrard, George Katsirelos, Zeynep Kızıltan, Émilie Picard-Cantin, Claude-Guy Quimper, and Toby Walsh. The balance constraint family. 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 174–189. Springer, 2014.

7 Christian Bessière, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artif. Intell.*, 244:315–342, 2017. doi:10.1016/j.artint.2015.08.001.

8 V. Brankov, P. Hansen, and D. Stevanović. Automated conjectures on upper bounds for the largest laplacian eigenvalue of graphs. *Linear Algebra and its Applications*, 414(2):407–424, 2006.

9 Jure Brence, Ljupčo Todorovski, and Sašo Džeroski. Probabilistic grammars for equation discovery. *Knowledge-Based Systems*, 224:107077, 2021. doi:10.1016/j.knosys.2021.107077.

10 Céline Brouard, Simon de Givry, and Thomas Schiex. Pushing data into CP models using graphical model learning and solving. 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 811–827. Springer, 2020. doi:10.1007/978-3-030-58475-7_47.

11 John William Charnley, Simon Colton, and Ian Miguel. Automatic generation of implied constraints. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 73–77. IOS Press, 2006. URL: http://www.booksonline.iospress.nl/Content/View.aspx?piid=1649.

12 Simon Colton, Andreas Meier, Volker Sorge, and Roy L. McCasland. Automatic generation of classification theorems for finite algebras. In David A. Basin and Michaël Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Computer Science*, pages 400–414. Springer, 2004. doi:10.1007/978-3-540-25984-8_30.

13 Alex Davies, Petar Veličković, Lars Buesing, Sam Blackwell, Daniel Zheng, Nenad Tomašev, Richard Tanburn, Peter Battaglia, Charles Blundell, András Juhász, Marc Lackenby, Geordie Williamson, Demis Hassabis, and Pushmeet Kohli. Advancing mathematics by guiding human intuition with ai. *Nature*, 600(7887):70–74, 2021. doi:10.1038/s41586-021-04086-x.

14 Siemion Fajtlowicz. On conjectures of Graffiti. *Discret. Math.*, 72(1-3):113–118, 1988. doi:10.1016/0012-365X(88)90199-9.

15 Aaron M. Ferber, Bryan Wilder, Bistra Dilkina, and Milind Tambe. Mipaal: Mixed integer program as a layer. 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 1504–1511. AAAI Press, 2020. URL: https://aaai.org/ojs/index.php/AAAI/article/view/5509.

16 Minh Hoàng Hà, Claude-Guy Quimper, and Louis-Martin Rousseau. General bounding mechanism for constraint programs. 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 158–172. Springer, 2015.

17 Pierre Hansen and Gilles Caporossi. Autographix: An automated system for finding conjectures in graph theory. *Electron. Notes Discret. Math.*, 5:158–161, 2000. doi:10.1016/S1571-0653(05)80151-9.

18 Samuel Kolb, Sergey Paramonov, Tias Guns, and Luc De Raedt. Learning constraints in spreadsheets and tabular data. *Mach. Learn.*, 106(9-10):1441–1468, 2017. doi:10.1007/s10994-017-5640-x.

19 Mohit Kumar, Stefano Teso, and Luc De Raedt. Acquiring integer programs from data. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1130–1136. ijcai.org, 2019. doi:10.24963/ijcai.2019/158.

20 Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=S1eZYeHFDS.

21 Craig E. Larson and Nicolas Van Cleemput. Automated conjecturing I: Fajtlowicz's Dalmatian heuristic revisited. *Artif. Intell.*, 231:17–38, 2016. doi:10.1016/j.artint.2015.10.002.

22 Jimmy Ho-Man Lee, Ka Lun Leung, and Yu Wai Shum. Consistency techniques for polytime linear global cost functions in weighted constraint satisfaction. *Constraints*, 19(3):270–308, 2014.

23 Doug Lenat. *AM: An artificial intelligence approach to discovery in mathematics.* PhD thesis, Stanford University, 1976.

24 Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proc. VLDB Endow.*, 8(10):1082–1093, 2015. doi:10.14778/2794367.2794377.

25 Sergey Paramonov, Samuel Kolb, Tias Guns, and Luc De Raedt. Tacle: Learning constraints in tabular data. In Ee-Peng Lim, Marianne Winslett, Mark Sanderson, Ada Wai-Chee Fu, Jimeng Sun, J. Shane Culpepper, Eric Lo, Joyce C. Ho, Debora Donato, Rakesh Agrawal, Yu Zheng, Carlos Castillo, Aixin Sun, Vincent S. Tseng, and Chenliang Li, editors, *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, pages 2511–2514. ACM, 2017. doi:10.1145/3132847.3133193.

26 Anselm Paulus, Michal Rolínek, Vít Musil, Brandon Amos, and Georg Martius. Comboptnet: Fit the Right NP-Hard Problem by Learning Integer Programming Constraints, 2021. arXiv:2105.02343.

27 Émilie Picard-Cantin, Mathieu Bouchard, Claude-Guy Quimper, and Jason Sweeney. Learning the parameters of global constraints using branch-and-bound. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 512–528. Springer, 2017. doi:10.1007/978-3-319-66158-2_33.

28 Steve Prestwich. Robust constraint acquisition by sequential analysis. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 355–362. IOS Press, 2020. doi:10.3233/FAIA200113.

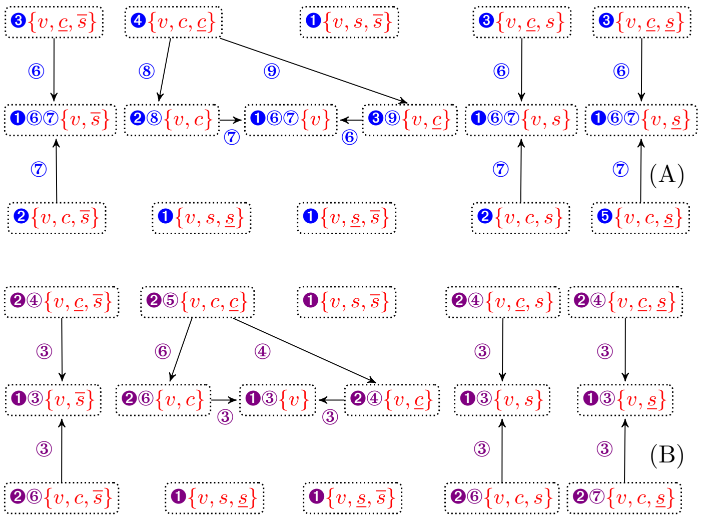
29 Gal Raayoni, Shahar Gottlieb, Yahel Manor, George Pisha, Yoav Harris, Uri Mendlovic, Doron Haviv, Yaron Hadad, and Ido Kaminer. Generating conjectures on fundamental constants with the Ramanujan Machine. *Nature*, 590:67–73, 2021. doi:10.1038/s41586-021-03229-4.

30 Helge Spieker and Arnaud Gotlieb. Learning objective boundaries for constraint optimization problems. In Giuseppe Nicosia, Varun Kumar Ojha, Emanuele La Malfa, Giorgio Jansen, Vincenzo Sciacca, Panos M. Pardalos, Giovanni Giuffrida, and Renato Umeton, editors, *Machine Learning, Optimization, and Data Science - 6th International Conference, LOD 2020, Siena, Italy, July 19-23, 2020, Revised Selected Papers, Part II*, volume 12566 of *Lecture Notes in Computer Science*, pages 394–408. Springer, 2020. doi:10.1007/978-3-030-64580-9_33.

31 Ljupco Todorovski. Equation discovery. In Claude Sammut and Geoffrey I. Webb, editors, *Encyclopedia of Machine Learning*, pages 327–330. Springer, 2010. doi:10.1007/978-0-387-30164-8_258.

32 Hao Wang. Toward mechanical mathematics. In Jörg Siekmann and Graham Wrightson, editors, *Automation of Reasoning: Classical Papers on Computational Logic 1957–1966*, pages 244–264. Springer-Verlag, Berlin, 1983.

A Map example



❶ $\bar{c} \le v$

❷ $\bar{c} \le v - c + 1$

❸ $\bar{c} \le (v = \underline{c}\, ?\, v : v - \underline{c})$

❹ $\bar{c} \le \underline{c} - c \cdot \underline{c} + v$

❺ $\bar{c} \le \underline{s} - c \cdot \underline{s} + v$

⑥ $\underline{c} = v$

⑦ $c = 1$

⑧ $\underline{c} = (c = 1\, ?\, v : 1)$

⑨ $c = 1 + (v \neq \underline{c})$

❶ $\bar{c} \le v$ ❷ $\bar{c} \le r + \underline{c}$

③ $r = 0, \underline{c} = v$

④ $r = \max(v - 2 \cdot \underline{c}, 0)$

⑤ $r = v - c \cdot \underline{c}$

⑥ $\begin{cases} r = v \cdot (c \geq 2) - c \cdot (c \geq 2) \\ \underline{c} = \max(2 \cdot v - v \cdot c, 1) \end{cases}$

⑦ $\begin{cases} r = v + \underline{s} \cdot (c = 1) - c \cdot \underline{s} \\ \quad -v \cdot (c = 1) \\ \underline{c} = \max(2 \cdot v - v \cdot c, \underline{s}) \end{cases}$

Figure 4 Map $\mathcal{M}^{\bar{c}\le}_{\{v,c,\underline{c},s,\underline{s},\bar{s}\}}$ of upper-bounds of the output characteristic $\bar{c}$ found by the Bound Seeker, where each dotted node contains, from left to right, a reference to the maximum conjecture ❶,...,❺, ❶,❷, possibly a set of maximality conjectures ⑥,...,⑨,③,...,⑦, and the set of input characteristics in red; Part (A) corresponds to the bounds found while only using the input characteristics, and Part (B) refers to the bounds found using also the secondary characteristics.

Parallel Hybrid Best-First Search

Abdelkader Beldjilali
Université Fédérale de Toulouse, INRAE, UR 875, 31326 Toulouse, France

Pierre Montalbano
Université Fédérale de Toulouse, ANITI, INRAE, UR 875, 31326 Toulouse, France

David Allouche
Université Fédérale de Toulouse, ANITI, INRAE, UR 875, 31326 Toulouse, France

George Katsirelos
Université Fédérale de Toulouse, ANITI, INRAE, MIA Paris, AgroParisTech, 75231 Paris, France

Simon de Givry
Université Fédérale de Toulouse, ANITI, INRAE, UR 875, 31326 Toulouse, France

Abstract

While processor frequency has stagnated over the past two decades, the number of available cores in servers or clusters is still growing, offering the opportunity for significant speed-up in combinatorial optimization. Parallelization of exact methods remains a difficult challenge. We revisit the concept of parallel Branch-and-Bound in the framework of Cost Function Networks. We show how to adapt the anytime Hybrid Best-First Search algorithm in a Master-Worker protocol. The resulting parallel algorithm achieves good load-balancing without introducing new parameters to be tuned as is the case, for example, in Embarrassingly Parallel Search (EPS). It has also a small overhead due to its light communication messages. We performed an experimental evaluation on several benchmarks, comparing our parallel algorithm to its sequential version. We observed linear speed-up in some cases. Our approach compared favourably to the EPS approach and also to a state-of-the-art parallel exact integer programming solver.

2012 ACM Subject Classification Computing methodologies → Parallel algorithms

Keywords and phrases Combinatorial Optimization, Parallel Branch-and-Bound, CFN

Digital Object Identifier 10.4230/LIPIcs.CP.2022.7

Supplementary Material *Software (Source Code)*: `https://github.com/toulbar2/toulbar2`
archived at `swh:1:dir:93fd0c2246746901b0391ac4c4c04ca57980b3bb`
Other (Results): `https://miat.inrae.fr/degivry/Beldjilali22Supp.pdf`

Funding This work has been partially funded by the French ”Agence Nationale de la Recherche”, through grant ANR-19-P3IA-0004. It was performed using HPC resources from CALMIP (Grant 2022-P21010).

Carbon footprint The experiments in this paper took approximately $17,000$ hours and emitted 68kg of CO_2, with an estimate of $4g/h$ per core.

1 Introduction

Cost Function Networks (CFNs), also known as Weighted Constraint Satisfaction Problems (WCSPs) [17] is a mathematical framework which has been derived from Constraint Satisfaction Problems by replacing constraints with cost functions. In a CFN, we are given a set of variables with an associated finite domain and a set of local cost functions. Each cost function involves some variables and associates a non-negative integer cost to each of the possible combinations of values they may take. The usual WCSP problem considered is to assign all variables in a way that minimizes the sum of all costs. This minimization problem is NP-hard, and exact methods usually rely on Branch and Bound (B&B) algorithms exploring a binary search tree with *soft local consistency* maintained at each node in order to improve the problem lower bound (represented by $c_\emptyset$) and prune domain values with a forbidden cost (represented by a maximum cost k) [5].

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 7; pp. 7:1–7:10
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Contraint Programming (CP) exact approaches usually rely on Depth-First Search (DFS) methods while Integer Linear Programming (ILP) approaches explore the tree in a best-first manner by exploiting strong bounds. We are interested in hybrid methods combining depth-first and best-first with possibly weaker bounds but faster to compute. This is the case of the Hybrid Best-First Search (HBFS) method [2]. HBFS is a B&B algorithm for solving WCSPs.

Dealing with parallel computers or grids to speed-up solving time of exact methods has been explored in many different ways. For grids, with slow network interconnection, MapReduce is a general approach exploiting problem decomposition into independent subproblems solved in parallel (*map* on the grid processors) and then sequentially *reduced* at the end of the resolution. In CP, this decomposition approach is called Embarrassingly Parallel Search (EPS) [15]. MapReduce has been applied also in the context of non serial dynamic programming in Graphical Models [19] and CFNs [3]. Message-passing approaches, on the other hand, take advantage of the low-latency communication of supercomputers, consisting of a large number of multiprocessor servers interconnected at high speed and low latency. This allows for finer granularity in B&B parallelization. According to a recent survey [9], parallelizing the search based on message-passing and parallel B&B in CP are difficult problems and still poorly explored. In CP, for example, COMET [18] uses *work-stealing* where workers which have run out of work take unexpanded nodes from other workers, leaving them less work to do and keeping all workers busy. In ILP, a recent review on parallel B&B was proposed in [21]. We selected the Master-Worker protocol as the basis for our approach. Other approaches rely on portfolios.

In this work we describe a parallel version of HBFS. We give an empirical evaluation on combinatorial optimization academic problems from Operations Research and real-life Graphical Model problems occuring in genetics and biology. Our experimental study analyses solving time and speed-ups of the parallel version compared to the original sequential HBFS. We also compare our approach with a parallel ILP solver (IBM Ilog `cplex`). Moreover, we performed experiments on a high-performance cluster to study the scalability of our algorithm and compare with EPS.

2 Hybrid Best-First Search

The sequential version of HBFS [2] is a B&B method for CFNs that combines Best-First Search (BFS) and Depth-First Search (DFS). Like BFS, HBFS provides an anytime global lower bound on the optimum, while also providing anytime upper bounds, like DFS. Hence, it provides feedback on the progress of search and solution quality in the form of an optimality gap. Besides, it exhibits highly dynamic behavior that allows it to perform on par with methods like Limited Discrepancy Search [11] and frequent restarting [10, 7] in terms of quickly finding good solutions. As in BFS, HBFS maintains a frontier of open search nodes. It expands each open node using DFS with a limit on its number of backtracks. Each bounded DFS returns a new list of open nodes to be inserted in the BFS frontier.

The pseudo-code of HBFS is given in Algorithm 1. The main procedure is in charge of the BFS frontier of open nodes. Here a node ν corresponds to a sequence of decisions $\nu.\delta$. The root node has an empty decision sequence (line 1). When a node is explored by DFS (line 5), an unassigned variable is chosen and a branching decision to either assign the variable to a chosen value (left branch, positive decision) or remove the value from the domain (right branch, negative decision) is taken. The number of decisions taken to reach a given node ν is the depth of the node, $\nu.depth$. HBFS always chooses the next open node to explore with minimum lower bound $\nu.lb$ (best-first principle) and, in case of ties, maximum depth

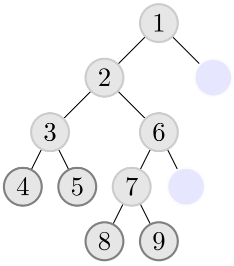


Figure 1 A tree that is partially explored by DFS with a backtrack limit $Z = 4$. Nodes with a bold border are leaves, nodes with no border are placed in the open list after the backtrack bound is exceeded. Nodes are numbered in the order they are visited.

$\nu.depth$ (depth-first principle) in the frontier. The minimum of all open node lower bounds, denoted $lb(open)$, is a valid global lower bound (kept in clb at line 6) for the problem. HBFS also maintains the current upper bound (cub) as the cost of the best solution found so far by DFS (line 5). The search ends when the open list is empty or contains nodes with a lower bound greater than or equal to cub (line 2).

Algorithm 1 Hybrid Best-First Search. Initial call: HBFS($c_\emptyset$,k) with $Z = 1$.

```
Function HBFS($clb$,$cub$): integer ;                    /* Returns the optimum value */
1   $open := \{\nu(\delta = \emptyset, lb = clb)\}$ ;          /* Initializes the open list with a root node */
2   while ($open \neq \emptyset$ and $clb < cub$) do
        $\nu$ :=pop($open$) ;      /* Chooses a node with minimum lower bound and maximum depth */
3       Restores state $\nu.\delta$, leading to assignment $A_\nu$, maintaining soft local consistency ;
4       $NodesRecompute := NodesRecompute + \nu.depth$ ;
5       $cub$ :=DFS($A_\nu$,$cub$,$Z$) ; /* Increase Nodes and put all right open branches in open */
6       $clb := \max(clb, lb(open))$ ;
        if ($NodesRecompute > 0$) then
7           if ($NodesRecompute/Nodes > \beta$ and $Z \leq N$) then $Z := 2 \times Z$;
8           else if ($NodesRecompute/Nodes < \alpha$ and $Z \geq 2$) then $Z := Z/2$;
    return $cub$;
```

DFS increases a counter *Nodes* at each branching decision. It can backtrack (taking right branches) up to a limit of Z backtracks. When this limit is reached, all the unexplored right branches are placed in *open*. HBFS controls the balance between best-first search (partially exploring more open nodes) and depth-first search (complete exploration from a given starting node). Best-first search requires recomputing the state $\nu.\delta$ of a node (line 3) which can be costly in practice. HBFS uses a simple rule to limit this recomputation effort (measured by *NodesRecompute* at line 4). It tries to keep the ratio $\frac{NodesRecompute}{Nodes}$ in the interval $[\alpha, \beta]$ by increasing (by a power of two) the backtrack limit Z if the ratio value is above β or decreasing Z if it is below *alpha* (lines 7–8). Initially, Z is set to 1. In order to avoid exponential DFS behavior, HBFS limits the maximum value taken by Z to N. We kept the same value $\alpha = 5\%, \beta = 10\%, N = 2^{14}$ in our experiments as in the original paper [2].

3 Parallel HBFS

The parallel version of HBFS is based on the Master-Worker parallel paradigm [21] where the *Master* is in charge of the open node frontier and dispatches the current best (with minimum lower bound) open node plus the current best solution found so far to the next available *Worker*. The Worker performs a bounded DFS starting from the received node and returns to the Master the resulting list of open nodes (see Fig. 1, with a DFS limit here of 4 backtracks). The Worker also returns the best solution found during its restricted search if any. Only the Master has a global view of the whole search and reports optimality gaps ($\frac{cub-clb}{cub}$) until the proof of optimality is reached: when the current best lower bound in the frontier of open nodes, including active worker starting nodes, is equal or greater than the cost of the best solution found so far or the frontier is empty and there are no active workers. When the problem is solved, the Master kills all the workers and returns the optimum value.

According to a round robin schema, the Master sends open nodes to every idle worker in a balanced way, ensuring a natural load balancing between the workers as soon as the number of open nodes in the frontier is larger than the number of workers. Moreover, an initial backtrack limit of $Z_i = 1$ associated to each Worker i favors the production of open nodes at the beginning of the search. Each Z_i is bounded by N as in sequential HBFS so that no worker takes too long.

The pseudo-code of the Master (resp. Worker) is given in Algorithm 2 (resp. Alg. 3). In the implementation, we avoid to send the same solution twice to a Worker. Moreover, workers send their solution only if it improves compared to the last solution sent by the Master. This strategy allows to shorten messages in the Master-Worker protocol.

Algorithm 2 Parallel HBFS-Master. Initial call for p workers: HBFS-Master($c_\emptyset$,k,$(1,\ldots,p)$).

```
Function HBFS-Master(clb, cub, S): integer ; /* S queue of workers, return the optimum */
  $open := \{\nu(\delta = \emptyset, lb = clb)\}$ ;           /* Initializes the open list with a root node */
  $I := S$ ;                                                  /* Queue of idle workers */
  $A := \emptyset$ ;             /* Maps active workers to open nodes currently being processed */
  while (($open \neq \emptyset$ or $A \neq \emptyset$) and $clb < cub$) do
    while ($open \neq \emptyset$ and $I \neq \emptyset$) do
      $\nu$ :=pop($open$) ;   /* Chooses a node with minimum lower bound and maximum depth */
      $i$ :=popFront($I$) ;                               /* Unqueue the first idle worker */
      $A := A \cup \{(i, \nu)\}$ ;
      Send $\nu$ and best solution $cub$ to Worker $i$ ;
    Receive a list of open nodes $\mathcal{V}$ and solution $cub'$ by worker $j$ ;     /* Wait for message */
    push($open$, $\mathcal{V}$) ;                 /* Adds worker open nodes to the Master open list */
    $cub := \min(cub, cub')$ ;                        /* Checks if a better solution as been found */
    pushBack($I$, $j$) ;              /* Pushes Worker j at the end of the idle worker queue I */
    $A := A \setminus \{(j, A[j])\}$ ;                          /* Removes Worker j from active workers */
    $clb := \max(clb, \min(lb(open), \min\{lb(\nu)$ for $(i,\nu) \in A\}))$ ;        /* Global lower bound */
  return $cub$;
```

3.1 Improving the ramp-up phase

We observed that at the beginning of the search the first active worker may take a long time to build its list of open nodes when it reaches the initial backtrack limit (equal to one). It can be explained by the fact that if it found a new solution then this improved upper bound will possibly imply more work in subsequent propagation made later when assessing the lower bound of each open node. This has the effect to slow-down the construction of the list of open nodes when HBFS stops backtracking. During this period, called the *ramp-up phase* (where some workers have not been assigned at least one task), no parallelism is exploited. We modified our communication protocol to send a message to the master as soon as an open-node has been collected or a new solution has been found by a worker inside its DFS

Algorithm 3 Parallel HBFS-Worker. Initial call for Worker i: HBFS-Worker(k,i) with $Z_i = 1$.

```
Procedure HBFS-Worker($cub$,$rank$) ;                          /* rank: Worker ID */
    while (true) do
        $open_i := \varnothing$ ;                                  /* local open list of Worker i */
        Receive an open node $\nu$ and solution $cub'$ by Master ;   /* Wait for message */
        $cub := \min(cub, cub')$ ;                  /* Updates cub and best solution if any */
        Restores state $\nu.\delta$, leading to assignment $A_\nu$, maintaining soft local consistency ;
        $NodesRecompute := NodesRecompute + \nu.depth$ ;
12      $cub :=$DFS($A_\nu$,$cub$,$Z_i$) ;  /* Increase Nodes ; put all right open branches in $open_i$ */
        if ($NodesRecompute > 0$) then
13          if ($NodesRecompute/Nodes > \beta$ and $Z_i \leq N$) then $Z_i := 2 \times Z_i$;
14          else if ($NodesRecompute/Nodes < \alpha$ and $Z_i \geq 2$) then $Z_i := Z_i/2$;
15      Send $open_i$ and best solution $cub$ to the Master ;     /* or closing-node mes. in burst
         mode */
```

subroutine (line 12). Such messages are received by the Master (line 9) which does not change the Worker state to idle (lines 10 and 11) until it receives a closing-node message by the Worker (sent at line 15). By doing so, it allows the Master to distribute open nodes to idle workers earlier before the first active worker has finished its initial DFS. We call this modified Master-Worker protocol the *burst mode*. However, the Worker can potentially send $O(nd)$ more messages and it disallows data compression of the open list messages.[1]

4 Experimental Results

We implemented in C++ our parallel HBFS in the CFN solver toulbar2.[2] We used the boost MPI library for the Master-Worker communication protocol. We kept default parameters of toulbar2 except no dichotomic branching in order to explore a binary search tree with DFS (option `-d:`). The variable ordering heuristic is *dom/wdeg* [4] combined with *last conflict* [14]. The value ordering heuristic exploits the last solution found if any [7] or else EDAC existential value [6]. EDAC is also used as soft local consistency during search. Instances were preprocessed by VAC [5] and the resulting CFNs saved to files before the experiments to reduce the setup sequential time of paralllel HBFS. We compared both the sequential and parallel version of HBFS and also with the integer programming solver cplex (version 20.1 with non-premature stop parameters `EPAGAP=EPGAP=EPINT=0`). We set the number of threads used by cplex to the desired number of cores.

Experiments were performed either on medium-scale computers (24-core Intel Xeon E5-2687W v4 at 3 GHz and 256 GB) with 1-hour timeout or on a large-scale cluster with more than $10,000$ cores (36-core per node of Intel Skylake 6140 at 2.3 GHz and 192 GB) with a longer 10-hour timeout for the sequential version only. Solving times are reported in seconds and correspond to CPU (resp. wall-clock) time for the sequential (resp. parallel) methods. No initial upper bounds were provided.

We tested the methods on four benchmarks selected from [12] with a total of 134 instances: two academic benchmarks taken in Operations Research, uncapacitated warehouse location problem (Warehouses) with 15 instances [13] and DIMACS maximum clique problem with

[1] In non-burst mode, all right branches share a common prefix in their $\nu.\delta$ and only the deepest δ information need to be sent to the Master.

[2] `https://toulbar2.github.io/toulbar2` version 1.2.

62 instances (MaxClique)[3] and two real-life Graphical Model benchmarks, linkage analysis problem occuring in genetics (Linkage) with 22 instances coming from UAI Evaluation 2008 [4] and computational protein design problem in biology (CPD) with 35 instances [1]. We applied the *tuple encoding* to convert Linkage and CPD to integer linear programs [12]. For a comparison on MaxClique with another parallel branch and bound implementation, see [16].

4.1 Comparison of parallel HBFS with its sequential version

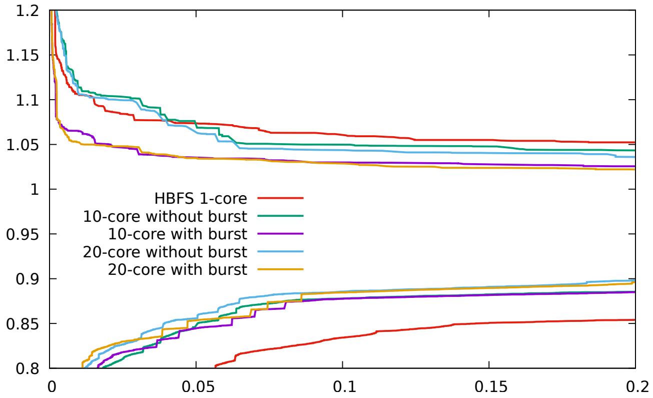


Figure 2 Comparison on a medium-scale computer between sequential versus parallel HBFS with or without burst mode. The x-axis represents normalized time (with 0.2 corresponding to 720 seconds). The y-axis corresponds to normalized lower and upper bounds on 134 instances (with 1 corresponding to the optimum or best known cost, see the text description).

We compared the anytime behavior of sequential (HBFS-1) and parallel HBFS (with 10 or 20 cores) with or without burst mode (see Sec. 3.1) on a medium-scale computer. We summarize the evolution of lower (*clb* in Alg. 1 and 2) and upper bounds (*cub*) for each method over all instances in Fig. 2. Specifically, for each instance we normalize all costs as follows: the initial lower bound $c_\emptyset$ produced by EDAC is 0; the best but potentially suboptimal solution found by any method is 1; the worst solution is 2. This normalization is invariant to translation and scaling. Additionally, we simply normalize time from 0 to 1, corresponding to 1 hour. A point x, y on the lower bound line for method M in Fig. 2 means that after normalized runtime x, method M has proved on average over all instances a normalized lower bound of y and similarly for the upper bound.

First, we observed that all parallel versions significantly outperformed the sequential HBFS lower bound curve. Concerning upper bound curves, the burst mode gave a clear advantage to parallel HBFS especially at the beginning of the search. In the sequel of the paper, we always report results of parallel HBFS with burst mode. As shown in the figure, increasing the number of cores from 10 to 20 slightly improved the bounds.

In Table 1 we report the number of instances solved by sequential and parallel HBFS for each benchmark. Parallel HBFS solved 1 more instance than the single core version in Linkage and 1 (resp. 3) in MaxClique using 10 (resp. 20) cores. We made local comparisons of solving times (shown in parentheses) by averaging on the subset of instances solved by the

[3] We removed the largest instances keller6 and p_hat1500-1,2,3 from the original 66 DIMACS instances.
[4] Linkage instances were further preprocessed by variable elimination limited to at most 8 neighbors [8].

three methods (HBFS-1, HBFS-10, HBFS-20). It allows us to display overall speed-up of parallel approaches by giving the ratio of total sequential over parallel time. Parallel HBFS obtained near linear speed-up on MaxClique. Recall that 1 core is used by the master and the rest by the workers in the Master-Worker approach preventing us from full linear speed-up. On CPD and Linkage the speed-up was halved. For Warehouses, only 50% of reduction in overall time was observed. This can be explained partly by the limited number of search nodes (Table 4 in Supplementary Material).We also observed that the evaluation of right branches made by the first active worker starting from the root node took most of the time. This is due to the fact that a first solution has been found by the worker resulting in more propagation on the right branches especially near the root. This pathological phenomenon did not appear on the other benchmarks.

Table 1 Number of solved instances within 1 hour (except for sequential HBFS-1 run on the cluster with a larger timeout of 10 hours) and average time in seconds in parentheses. To compute the mean we only consider for a given method (toulbar2 HBFS or cplex) the instances solved with any number of cores on the same computer (server with 3 GHz cores or cluster with 2.3 GHz cores).

Method	CPD (35)		Warehouses (15)		Linkage (22)		MaxClique (62)	
		Speed-up		*Speed-up*		*Speed-up*		*Speed-up*
HBFS-1	30 (43.44s)		15 (128.96s)		20 (23.24s)		37 (364.25s)	
HBFS-10	30 (8s)	*5.43*	15 (80.174s)	*1.61*	21 (3.5s)	*6.64*	38 (40.24s)	*9.05*
HBFS-20	30 (4.43s)	*9.81*	15 (85.39s)	*1.51*	21 (2s)	*11.62*	40 (19.9s)	*18.3*
cplex-1	24 (331.2s)		15 (123.83s)		22 (8.04s)		42 (282.16s)	
cplex-10	24 (226.51s)	*1.46*	**15 (68.82s)**	*1.8*	22 (2.56s)	*3.14*	45 (**55.48s**)	*5.08*
cplex-20	24 (198.49s)	*1.67*	15 (72.06s)	*1.72*	**22 (2.29s)**	*3.51*	**46** (71.47s)	*3.95*
HBFS-1 (cluster)	30 (66.46s)		15 (392.30s)		21 (427.21s)		37 (504s)	
HBFS-180 (cluster)	**30 (3.7s)**	***17.96***	15 (126s)	***3.11***	22 (4.15s)	***102.94***	45 (6.44s)	***78.26***

4.2 Comparison of parallel HBFS with integer programming

In Table 1 we also report the number of instances solved and their average solving time (as explained above) by cplex using multithreading. It clearly dominates HBFS on Linkage (Supp. Fig. 5).For Warehouses, the differences are less important still in favor of cplex. For MaxClique, although the global picture shows that it solved six more instances than HBFS with 20 cores, both methods performed well on different subsets of instances (e.g., HBFS-20 solved two instances – brock400_4 and sanr400_0.7 – unsolved by cplex-20 whereas cplex-20 solved eight instances unsolved by HBFS-20). For CPD, the CFN approach largely dominates the integer programming approach for all the instances. Concerning anytime curves shown in Fig. 3 (see also Supp. Fig. 4 and 5), the CFN approach is also significantly superior to cplex on average in producing good upper bounds faster, HBFS-20 being the best method. Concerning overall speed-up, cplex had difficulties to benefit from parallelism on CPD, Linkage, and Warehouses where it usually develops a small amount of search nodes (less than $7,059$ nodes except on Linkage/pedigree19 and pedigree40), resulting in poor speed-up except in a few cases. The speed-up is better on MaxClique but seems to stagnate when going from 10 to 20 cores (it was even slower on four instances).

4.3 Comparison of parallel HBFS with EPS on a cluster

The EPS approach is a two-phase procedure. First, the problem to be solved is decomposed into a list of l independent subproblems. Next, all the subproblems are solved in parallel (with at most p workers running at the same time) based on a particular scheduling strategy

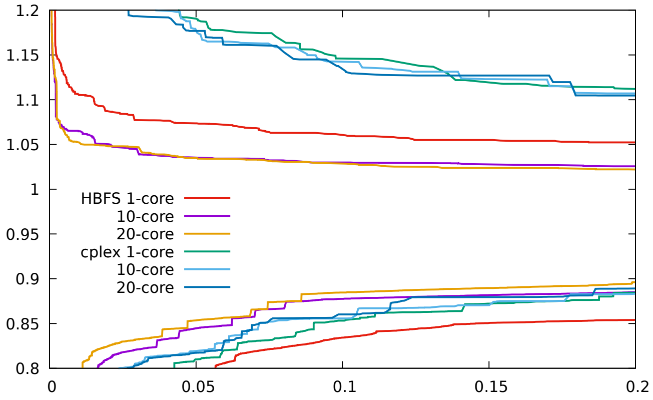


Figure 3 Comparison on a medium-scale computer between toulbar2 using parallel HBFS (with burst mode) and cplex using multiple threads. The x-axis represents normalized time (with 0.2 corresponding to 720 seconds). The y-axis corresponds to normalized lower and upper bounds on 134 instances (with 1 corresponding to the optimum or best known cost, see the text description).

with no communication between the workers. For optimization problems, we need to provide a good initial upper bound. Otherwise the search tree can be much larger than needed. In the first phase, we used the original HBFS method to collect l subproblems. As soon as HBFS has more than l open nodes in its frontier it stops and returns the current upper bound (cub) and the list of open nodes (without those having a lower bound $lb(\nu) \geq cub$). Each open node ν defines an independent subproblem with partial assignment $\nu.\delta$. In order to collect open nodes more rapidly we fix the (maximum) backtrack limit $Z = N = 1$. Ideally l should be $30 \times p$ with p the number of available cores [15]. In the second phase, we schedule on the cluster the subproblems that are solved by the original HBFS method using a simple scheduling heuristic based on increasing $|\nu.\delta|$.

In Table 2 we report for nine difficult instances their optimum value, the upper bound found at the end of EPS Phase-1, the actual number of generated subproblems, the average solving time of all subproblems, the maximum solving time, the number of failed subproblems (timeout of 1 hour) and the overall solving time of EPS Phase-2 using 180 cores on the cluster. We compare with HBFS using the same number of cores. Our EPS strategy failed on 4/9 instances. In parentheses, we indicate the maximum depth $\nu.depth$ of failed subproblems. Clearly, finding the right number l of not-too-difficult subproblems corresponding to partial assigments greater than a given depth is a challenging task. In our experiments, we tried with different values for $l \in [50, 6000]$, selecting the largest threshold value with a Phase-1 duration being less than 1 second for Linkage ($l = 6000$), 6 seconds for MaxClique ($l = 6000$) and 44 seconds for CPD ($l = 1000$). On the opposite, we did not tune any specific parameter for our parallel HBFS method.

In Table 1 we also report the overall speed-up of HBFS-180 compared to HBFS-1 on the cluster. HBFS-180 got a two-order-of-magnitude speed-up on Linkage.

5 Conclusion

Although the speed-up offered by the parallel version of HBFS was very instance dependent, we observed significant gain on several instances, outperforming in some cases state-of-the-art solvers like `cplex`. Even if the scalability of our approach must be subject of deeper investigation, due to the minimal size of the information shared between the Master and the Workers, our approach is very likely compliant with a larger number of cores.

Table 2 EPS and HBFS-180 results on hard instances (with n variables and maximum domain size d). A '-' indicates that some (see #failed) subproblems could not be solved in less than 3, 600sec.

instance	n	d	opt.	cub	l	av. time	max. t.	#fail(depth)	EPS-180	HBFS-180
linkage/pedigree19	259	5	4625	5684	5114	20.57	-	1 (4)	-	**69.1**
linkage/pedigree40	274	6	7300	8838	5641	101.99	-	49 (21)	-	**1680**
linkage/pedigree51	295	5	6406	6802	5798	0.61	497.38	0	499	**5.7**
cpd/1BRS	38	178	4007610	4007679	956	2.94	38.90	0	44	**37.5**
cpd/1CDL	38	170	3590514	3590825	1001	6.66	79.04	0	79	**18.3**
cpd/1GVP	52	170	5196719	5196841	979	14.59	170.66	0	171	**17.0**
maxcl./brock400_1	400	2	373	379	6010	63.95	-	12 (149)	-	**1812**
maxcl./brock400_2	400	2	371	379	5975	65.27	-	18 (149)	-	**880**
maxcl./san400_0.5_1	400	2	387	392	6073	5.07	414.96	0	3652	**1220**

A more challenging task which remains as future work is to exploit the structure of CFNs by parallelizing Backtrack with Tree Decomposition (BTD-HBFS) [2]. Shared memory protocols may be more suitable for this task to make learnt nogoods available to all Workers.

On the practical side, our parallel HBFS could ran in conjunction with a parallel large neighborhood search strategy [20] offering even better anytime lower and upper bounds.

References

1 D Allouche, J Davies, S de Givry, G Katsirelos, T Schiex, S Traoré, I André, S Barbe, S Prestwich, and B O'Sullivan. Computational protein design as an optimization problem. *Artificial Intelligence*, 212:59–79, 2014.

2 D Allouche, S de Givry, G Katsirelos, T Schiex, and M Zytnicki. Anytime Hybrid Best-First Search with Tree Decomposition for Weighted CSP. In *Proc. of CP-15*, pages 12–28, Cork, Ireland, 2015.

3 D. Allouche, S. de Givry, and T. Schiex. Towards parallel non serial dynamic programming for solving hard weighted csp. In *Proc. of CP-10*, St Andrews, Scotland, 2010.

4 F Boussemart, F Hemery, C Lecoutre, and L Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.

5 M. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7–8):449–478, 2010.

6 S. de Givry, M. Zytnicki, F. Heras, and J. Larrosa. Existential arc consistency: Getting closer to full arc consistency in weighted csps. In *Proc. of IJCAI'05*, pages 84–89, Edinburgh, Scotland, 2005.

7 E Demirovic, G Chu, and P J. Stuckey. Solution-based phase saving for CP: A value-selection heuristic to simulate local search behavior in complete solvers. In *Proc. of CP-18*, pages 99–108, Lille, France, 2018.

8 A Favier, S de Givry, A Legarra, and T Schiex. Pairwise decomposition for combinatorial optimization in graphical models. In *Proc. of IJCAI-11*, Barcelona, Spain, 2011.

9 I Gent, I Miguel, P Nightingale, C McCreesh, P Prosser, N Moore, and C Unsworth. A review of literature on parallel constraint solving. *Theory and Practice of Logic Programming*, 18(5-6):725–758, 2018.

10 C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. of AAAI'98*, Madison, WI, 1998.

11 W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proc. of IJCAI'95*, Montréal, Canada, 1995.

12 B Hurley, B O'Sullivan, D Allouche, G Katsirelos, T Schiex, M Zytnicki, and S de Givry. Multi-Language Evaluation of Exact Solvers in Graphical Model Discrete Optimization. *Constraints*, 21(3):413–434, 2016.

13 J Kratica, D Tošic, V Filipović, and I Ljubić. Solving the simple plant location problem by genetic alg. *RAIRO*, 35(1):127–142, 2001.

14 C. Lecoutre, L Saïs, S. Tabary, and V. Vidal. Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173:1592,1614, 2009.

15 A Malapert, J-C Régin, and M Rezgui. Embarrassingly parallel search in constraint programming. *Journal of Artificial Intelligence Research*, 57:421–464, 2016.

16 C McCreesh and P Prosser. The shape of the search tree for the maximum clique problem and the implications for parallel branch and bound. *ACM Trans. Parallel Comput.*, 2(1), 2015.

17 P. Meseguer, F. Rossi, and T. Schiex. Soft constraints processing. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 9. Elsevier, 2006.

18 L Michel, A See, and P Van Hentenryck. Parallelizing constraint programs transparently. In C Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 514–528, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

19 L Otten and R Dechter. And/or branch-and-bound on a computational grid. *JAIR*, 59:351–435, 2017.

20 Abdelkader Ouali, David Allouche, Simon de Givry, Samir Loudni, Yahia Lebbah, Francisco Eckhardt, and Lakhdar Loukil. Iterative Decomposition Guided Variable Neighborhood Search for Graphical Model Energy Minimization. In *Proc. of UAI-17*, pages 550–559, Sydney, Australia, 2017.

21 T Ralphs, Y Shinano, T Berthold, and T Koch. Parallel solvers for mixed integer linear optimization. In *Handbook of parallel constraint reasoning*, pages 283–336. Springer, 2018.

Learning MAX-SAT Models from Examples Using Genetic Algorithms and Knowledge Compilation

Senne Berden ✉

Declarative Languages and Artificial Intelligence, KU Leuven, Belgium

Mohit Kumar ✉

Declarative Languages and Artificial Intelligence, KU Leuven, Belgium

Samuel Kolb ✉

Declarative Languages and Artificial Intelligence, KU Leuven, Belgium

Tias Guns ✉ 🏠

Declarative Languages and Artificial Intelligence, KU Leuven, Belgium

Abstract

Many real-world problems can be effectively solved by means of combinatorial optimization. However, appropriate models to give to a solver are not always available, and sometimes must be learned from historical data. Although some research has been done in this area, the task of learning (weighted partial) MAX-SAT models has not received much attention thus far, even though such models can be used in many real-world applications. Furthermore, most existing work is limited to learning models from non-contextual data, where instances are labeled as solutions and non-solutions, but without any specification of the contexts in which those labels apply. A recent approach named HASSLE-SLS has addressed these limitations: it can jointly learn hard constraints and weighted soft constraints from labeled contextual examples. However, it is hindered by long runtimes, as evaluating even a single candidate MAX-SAT model requires solving as many models as there are contexts in the training data, which quickly becomes highly expensive when the size of the model increases. In this work, we address these runtime issues. To this end, we make two contributions. First, we propose a faster model evaluation procedure that makes use of knowledge compilation. Second, we propose a genetic algorithm named HASSLE-GEN that decreases the number of evaluations needed to find good models. We experimentally show that both contributions improve on the state of the art by speeding up learning, which in turn allows higher-quality MAX-SAT models to be found within a given learning time budget.

2012 ACM Subject Classification Computing methodologies → Machine learning

Keywords and phrases Machine learning, constraint learning, MAX-SAT

Digital Object Identifier 10.4230/LIPIcs.CP.2022.8

Supplementary Material *Software (Source Code)*: `https://github.com/ML-KULeuven/HASSLE-GEN` archived at `swh:1:dir:6769420692514ea592885add6759177a0c221e8b`

Funding This project was partially funded by the Research Foundation - Flanders (FWO) projects G0G3220N and FWO-S007318N, as well as the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant Nos. 101002802, CHAT-Opt and 694980, SYNTH) and the "Agentschap Innoveren & Ondernemen" (VLAIO) as part of the innovation mandate HBC.2021.0246. (S. Kolb).

Acknowledgements We would like to thank Luc De Raedt for his helpful comments.

1 Introduction

Many real-world problems can be effectively solved by means of combinatorial optimization. With appropriate mathematical models, problems like planning [25], routing [20] and scheduling [9] can be delegated to a highly-optimized solver that can quickly and automatically yield one or more high-quality solutions. However, automatic solvers do not remove the need for human effort altogether; they merely move it from the solving phase to the modeling phase. Unfortunately, the manual construction of an appropriate model can be difficult for several reasons. First, it requires both modeling and expert domain knowledge. Finding people that possess both, or setting up a collaboration to marry both, is not a trivial issue. Second, some optimization criteria and constraints might be hard to express explicitly for a human modeler. Third, even with the criteria and constraints successfully modeled, the potentially difficult task of weighting them by importance relative to each other remains, at least for problems with multiple optimization criteria and problems that contain weighted soft constraints. Finally, manual modeling can be a laborious process. It might require many iterations of gradual refinement of the model, as well as a survey of the various stakeholders to discover what is considered important and what constraints might apply. This arduous model construction phase motivates another approach altogether: *learning* the model from data, as opposed to manually constructing it.

This is the machine learning task we tackle in this work. Specifically, we focus on learning *MAX-SAT models*, a type of mathematical model that can be used to represent many interesting real-world problems [1, 9, 25, 26, 27]. The task involves jointly learning hard and soft constraints – as well as appropriate weights for the latter – that best explain a set of labeled contextual examples [16]. These examples are known solutions and non-solutions to the considered problem. For instance, when learning a model for rostering nurses in a hospital, historical rosters can be used as examples. The rosters would be labeled as positive or negative depending on whether they were deemed suitable.

As alluded to above, the type of examples we consider are *contextual*; they come with associated *contexts*. These contexts represent the states of affairs that the examples occurred in. Let us examine their relevance. In the nurse rostering scenario, a roster in which some nurses have to work several consecutive long shifts is likely not optimal in normal circumstances, and thus not a solution. However, in the context of a pandemic taking place, or many nurses being unavailable due to illness or other circumstances, such a roster might be optimal and an appropriate solution to the rostering problem. Contexts are a useful way of including this type of situational information in the example set, with the goal of learning a model that can generalize across contexts. For any given context, such a model can then be used to construct new solutions that are appropriate in that context. As will be discussed in Section 2, learning from examples that are annotated with contexts is one of the things that sets this work apart from most existing related work.

The problem of learning MAX-SAT models from labeled contextual examples has recently been addressed in [16]. This work laid the theoretical groundwork, showing that, given enough training data and time, high-quality models can be learned. Subsequent work introduced HASSLE-SLS, a more efficient stochastic local search approach [17]. However, HASSLE-SLS still suffers from long runtimes, which stem from the particularly expensive candidate model evaluation involved in learning. Faster learning is desirable, as it would allow for better models to be learned in a given training time budget. To this end, we make two contributions:

1. We develop a novel knowledge-compilation-based evaluation procedure that significantly speeds up MAX-SAT model evaluation when learning small to medium-sized models, or when learning from a large set of examples. Because evaluation is the major bottleneck in the search for good models, improving its efficiency speeds up learning considerably.
2. We develop a novel genetic algorithm named HASSLE-GEN, whose crossover operator takes much larger steps in the search space than HASSLE-SLS, reducing the number of model evaluations needed to find a good model. Because of this, HASSLE-GEN learns high-quality MAX-SAT models significantly faster than the state of the art.

2 Related Work

We focus specifically on learning weighted partial MAX-SAT models. This work is positioned in the research field of constraint learning, because the unweighted and weighted clauses in these models can respectively be seen as hard and weighted soft constraints. Here, we provide an overview of some of the most relevant works in constraint learning. For a more thorough overview, we refer the reader to [8].

Learning hard constraints. We start by discussing a few approaches that are aimed at learning constraint satisfaction problems (CSPs). The first of these, named *ConAcq.1*, is a version space algorithm that learns CSPs from examples labeled as solutions and non-solutions [5]. The version space is represented implicitly as a propositional CNF formula over variables that denote binary constraints. Complete variable assignments that satisfy the formula then correspond to CSPs in the version space, i.e., CSPs that are consistent with respect to the given labeled examples. This approach was followed by *ConAcq.2* [6] and *QuAcq* [4], two active learning approaches that learn more efficiently by respectively asking the user complete and partial membership queries.

Some other approaches are instead focused on learning global constraints. For example, [23] and [24] focus on learning the parameters of global constraints from a small pool of positive examples. Similarly, *Model Seeker* is effective at learning global constraints from positive examples provided in matrix-form [3]. In *Model Seeker*, fitting constraints are taken from a catalogue of constraints and related metadata (e.g., information about implication relations between constraints), and then subjected to several types of redundancy checks and simplifications. This method has proven to be very effective, even in the presence of many variables or only few examples.

Finally, *GenetiCS* is a method that learns constraints for mathematical programming models from known feasible and infeasible examples [22]. The approach is related to the one developed in this work in that it involves an evolutionary algorithm, but differs in that it learns linear and polynomial (in)equalities represented as abstract syntax trees (ASTs), rather than weighted partial MAX-SAT models. Additionally, *GenetiCS* does not learn from contextual examples.

Learning soft constraints. Several methods that focus on learning an objective to be optimized, in the form of weighted soft constraints, also exist. They can be categorized into parameter learning and structure learning approaches. The former is merely aimed at learning the weights of a given set of constraints. The latter also learns the constraints themselves.

One relevant example is *CLEO*, an interactive method whose purpose is to learn the weights of clauses occurring in MAX-SAT or MAX-SMT models [7]. Although the focus lies on learning weights, the algorithm is asked to weight many more constraints than are desired in the final model. However, a sparsity assumption is applied by including the minimization of the weight vector's 1-norm as one of the method's objectives, leading to a lot of constraints getting assigned a weight of zero. In this sense, the approach can be seen as a form of structure learning.

Another approach with the purpose of learning soft constraints and their weights is presented in [13]. In this work, a type of weighted MAX-SAT model is learned. These MAX-SAT models are an extension to the ones we consider, in that the constraints are function-free disjunctive first-order logic clauses, and thus are not limited to propositional logic. Another difference with the work presented here is the input given: the method requires examples of possible worlds as well as preference relations between these worlds to be provided. It then uses inductive logic programming to learn a set of appropriate constraints, which are subsequently weighted using preference learning techniques.

HASSLE. The approaches discussed above are focused on learning either hard constraints or soft constraints. They differ significantly with our approach, in which both types of constraints are being learned *jointly*. This joint learning is desirable in the setting we consider, as will be discussed in Section 4. Another large difference is that none of the above works are aimed at learning from *contextual* examples. However, it is a realistic assumption that what one considers optimal depends on the context one is positioned in.

Although none of the methods discussed above is applicable to the presently considered problem setting, two approaches that do try to solve this task already exist. The first approach, called HASSLE-MILP, was introduced in [16]. This work also proved that MAX-SAT models are PAC-learnable, justifying empirical risk minimization. HASSLE-MILP formulates the learning task as a mixed-integer linear program (MILP), which can be solved by an off-the-shelf solver. The main drawbacks of this method are that it is not optimized for efficiency and that, unless a solver that offers anytime functionality is used, it either produces a solution that has zero empirical error, or no solution at all.

The second approach, called HASSLE-SLS, is a stochastic local search algorithm that improves on HASSLE-MILP by offering anytime functionality and by increasing efficiency [17]. It keeps track of a current model and iteratively constructs and evaluates a neighborhood of minimally altered models, of which the best neighbor then replaces the current model. To make the approach tractable, the neighborhood is constructed heuristically, keeping only those models that show some promise of improving on the current model according to the employed heuristic.

One large drawback of HASSLE-SLS is its long runtime, which is exacerbated when the size of the learning problem increases. The algorithm slows down significantly when the model one aims to learn becomes larger, or when the number of distinct contexts occurring in the training data increases. At the heart of this issue lies the high cost of model evaluation. HASSLE-SLS is particularly affected by this, because in every iteration, it evaluates an entire neighborhood of MAX-SAT models before making only a minimal alteration to the current model. In this paper, we alleviate the runtime issues in two ways. First, we make model evaluation less expensive by making use of knowledge compilation. Second, we reduce the number of evaluations needed to find a good model, by developing a novel search strategy called HASSLE-GEN.

3 Preliminaries

Weighted partial MAX-SAT. Let $X = \{X_1, \ldots, X_n\}$ be a set of Boolean variables. An *assignment* x – also referred to as an *instance* – is a mapping of each variable $X_i \in X$ to either *true* or *false*.

Let $L = \{X_i, \overline{X_i} | X_i \in X\}$ be the set of literals defined over X, where $\overline{X_i}$ is the negation of X_i. A *disjunctive clause* C is a disjunction of literals from L. An assignment satisfies C when it satisfies at least one of the literals occurring in C. A formula in *conjunctive normal form* (CNF) F is a conjunction of clauses. An assignment *satisfies* F when it satisfies all of the clauses occurring in F. A *weighted clause* (C, w) consists of a clause C and a weight w, which we assume lies in $[0, 1]$ without loss of generality. A weighted CNF formula is a conjunction of weighted clauses.

A *MAX-SAT problem* is defined by a CNF formula F. An assignment is a solution to the MAX-SAT problem when it satisfies the maximum number of clauses that can simultaneously be satisfied in F. A *partial* MAX-SAT problem is defined by two CNF formulas, F_h and F_s. An assignment is a solution when it satisfies F_h and satisfies the maximum number of clauses that can simultaneously be satisfied in F_s while satisfying F_h. Finally, a *weighted* partial MAX-SAT problem is defined by a CNF formula F_h and a weighted CNF formula F_s. An assignment is a solution when it satisfies F_h and accumulates the maximum total weight in satisfied weighted clauses that can be accumulated in F_s while satisfying F_h. The clauses in F_h and the weighted clauses in F_s can respectively be seen hard and weighted soft constraints. When an assignment is not a solution because it does not satisfy all the clauses in F_h, it is called *infeasible*. When it *does* satisfy all clauses in F_h, but does not accumulate the maximum attainable weight in satisfied soft constraints of F_s, it is called *suboptimal*.

For the sake of brevity, we from now on refer to weighted partial MAX-SAT simply as MAX-SAT. Because this work aims to *learn* MAX-SAT problems from known solutions and non-solutions, we will generally refer to a MAX-SAT problem as a MAX-SAT *model*. When we speak of the MAX-SAT learning problem, we refer to the problem of learning MAX-SAT models.

Contexts. As explained in Section 1, there is a strong motivation to think of a historical labeled example as an instance that is known to be a solution or non-solution *in a specific context*. As a context might represent a state of affairs, it often makes sense for it to be a conjunction. Most generally, however, a context ϕ is simply a propositional formula over Boolean variables.

An assignment is a solution to a MAX-SAT model consisting of hard constraints F_h and weighted soft constraints F_s in a context ϕ if it satisfies F_h and ϕ, and accumulates the maximum total weight in satisfied weighted clauses that can be accumulated in F_s while satisfying F_h and ϕ. Thus, it is possible for an instance to be optimal in a particular context, but suboptimal outside of that context.

4 Problem Statement

In this work, the goal is not to solve MAX-SAT problems, but to *learn* them from a set of labeled, contextual examples. Specifically, such an example consists of:

1. A context ϕ
2. An assignment x that satisfies context ϕ
3. A Boolean label l, denoting whether assignment x is an optimal solution to the target model in context ϕ or not

Note that the label "non-solution" does not specify whether the example is a non-solution because it is infeasible or because it is suboptimal. We focus on this type of supervision, because in real-world settings, the reason for the negative label is typically not available. While this assumption on the input makes supervision easier to provide, it also gives rise to a credit assignment problem: when a candidate model wrongly labels an example as a solution, it is unclear whether the hard constraints or the soft constraints should be altered. For this reason, both types of constraints should be learned jointly, rather than separately. The learning task becomes:

▶ **Definition 1** (MAX-SAT learning). *Given Boolean variables $X = \{X_1, \ldots, X_n\}$ and a set of labeled contextual examples $S = \{(\phi_i, x_i, l_i)|i = 1, \ldots, m\}$, find hard constraints F_h and weighted soft constraints F_s that define a MAX-SAT model which can be used to obtain high-quality instances in any context ϕ.*

Something not yet specified is what constitutes a high-quality instance. Intuitively, an instance is good when it is feasible and close to optimal with respect to the ground-truth model. The learned model's ability to generate high-quality instances is reflected in its *infeasibility* and *average regret*, which will be the primary performance measures in the experimental evaluation. The infeasibility expresses what proportion of solutions to the learned model are actually infeasible with respect to the ground-truth model. The regret captures how good the learned model's solutions are with respect to the ground-truth model's soft constraints. In section 7, we will discuss how exactly these measures are computed.

As the ground-truth model is not available, infeasibility and regret cannot be used during learning. Instead, we aim to maximize the model's training set accuracy, which is the proportion of examples whose label correctly denotes how the example relates to the model.

5 Knowledge Compilation

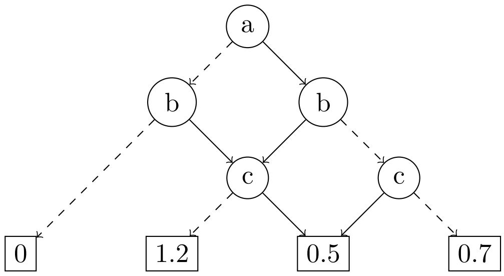
When learning MAX-SAT models, the vast majority of the runtime is spent evaluating candidate models. Let us consider in more detail why this is the case.

To fully evaluate a model's training set accuracy, it has to be evaluated on all contextual examples included in the training data. Checking whether an instance is feasible or infeasible is relatively straightforward: one merely has to loop over at most all hard constraints of the model, and check for each whether it is satisfied by the instance. Checking optimality, however, requires knowing the maximum total weight that can be accumulated while satisfying the hard constraints. Attaining this information requires *solving* the model, which is NP-hard [18]. An additional complication is the fact that examples are accompanied by contexts, which might affect the optimal total weight that can be accumulated. So, in order to correctly evaluate a single candidate model, HASSLE-SLS solves a separate MAX-SAT problem from scratch for every distinct context occurring in the data (which can then be cached and reused for all examples sharing that context). This causes model evaluation to be particularly slow, which in turn significantly slows down learning.

The solution we present is based on a novel representation of a MAX-SAT model as an algebraic decision diagram (ADD). An ADD is an extension of a binary decision diagram (BDD), in which the terminal nodes can be assigned real values, rather than just *true* or *false* [2]. It is a rooted, directed, acyclic graph which consists of two types of nodes: decision nodes and terminal nodes. A decision node is associated with a Boolean variable and branches into two child nodes; one for each truth assignment to the variable. A terminal node is associated with a real value.

Our ADD representation of a MAX-SAT model maps every infeasible instance onto a terminal node with value 0 and every feasible instance onto a terminal node with as value the sum of the weights of all satisfied soft constraints. A useful aspect of this representation is that the optimal value (in the absence of a context) can very easily be discovered: one simply has to loop through all terminals in order to find the highest one. This information is required to determine whether an instance is a solution. An example of a MAX-SAT model and its corresponding ADD representation is shown in Example 2.

▶ **Example 2.**
Consider the MAX-SAT model with hard constraints $F_h = (a \vee b)$ and soft constraints $F_s = (0.5 : b \vee c) \wedge (0.7 : \neg c)$. This problem corresponds to the following ADD, in which negative edges are represented as dashed lines and positive edges as solid lines:



Although the construction of the ADD is necessarily an expensive operation – as it practically solves the NP-hard MAX-SAT problem – there are potential time benefits in reusing the resulting ADD for context-specific inference. The main idea is that each candidate model can be converted to an ADD once, after which the diagram can be reused to compute the optimal attainable value for all separate contexts occurring in the training data. Contrast this with the original approach, in which a brand new MAX-SAT problem has to be solved for every model-context combination.

The evaluation procedure. The full knowledge-compilation-based evaluation procedure is shown at a high level in Algorithm 1. It takes as input a MAX-SAT model and a set of labeled contextual examples, and it produces as output the model's accuracy on the set of examples. To do this, it performs three steps. First, it constructs an ADD that represents the model in the absence of any context. Next, it runs through all the contexts present in the example set and, for each, uses the ADD constructed in step 1 to compute the optimal value in that context. Finally, it runs through all examples and, for each, uses the ADD constructed in step 1 to compute the value the example achieves. Each example's value, along with its associated context's optimal value computed in step 2, can then be used to label the example. Finally, the assigned label is compared with the example's label in the training data.

This description naturally raises the following two questions, which we answer in turn:

1. How to convert a MAX-SAT model to an ADD?
2. How to incorporate contexts in the inference done on the ADD?

Constructing the ADD. To transform a MAX-SAT model defined by hard constraints F_h and soft constraints F_s to an ADD, we make use of the basic operators defined on BDDs and ADDs. We do not go over the details of these operations here. For more information, we refer the reader to chapter 6 of the book *Logic in Computer Science* [12].

The transformation procedure starts by constructing a separate BDD for each disjunctive clause, irrespective of whether it is weighted or not. In each BDD that corresponds to a soft constraint from F_s, the terminal that denotes *true* is then given the constraint's associated weight as value. This effectively casts the BDD to an ADD representing the weighted soft constraint.

All the BDDs that represent hard constraints are then combined using the multiplication operator. This results in a single BDD that represents the conjunction of all hard constraints. Similarly, the ADDs representing soft constraints are combined using the addition operator. This results in a single ADD in which every instance leads to a terminal that denotes the total weight in satisfied soft constraints accumulated by that instance. However, this ADD does not yet consider whether the instances are feasible or not.

Algorithm 1 The evaluation procedure using knowledge compilation.

```
procedure Evaluate($model$: a MAX-SAT model, $examples$: a set of labeled contextual examples)
    $score \leftarrow 0$
    $ADD_{model} \leftarrow$ convert $model$ to ADD
    $contexts \leftarrow$ set of all separate contexts occuring in $examples$
    for each $context$ in $contexts$ do
        Compute best value of $ADD_{model}$ in $context$
        Cache this best value
    for each $example$ in $examples$ do
        $instance$, $context$, $label \leftarrow$ the instance, context and label of $example$
        $best\text{-}value \leftarrow$ retrieve optimal value in $context$ from cache
        $value \leftarrow$ compute value of $instance$ in $ADD_{model}$
        if $value = best\text{-}value$ then
            $assigned\text{-}label \leftarrow$ solution
        else
            $assigned\text{-}label \leftarrow$ non-solution
        if $label = assigned\text{-}label$ then
            $score \leftarrow score + 1$
    $score \leftarrow \frac{score}{\textsc{Length}(examples)}$
    return $score$
```

Finally, the BDD representing the hard constraints and the ADD representing the soft constraints are multiplied. The effect of this is that the terminals of all feasible paths of the ADD representing the soft constraints are multiplied by 1, while the terminals of infeasible paths are multiplied by 0. In turn, all feasible paths still end up in the same terminal node as before, while infeasible paths are redirected to a terminal with value 0. For compactness, the resulting diagram is then reduced to a reduced ordered BDD [12]. The resulting diagram represents the entire MAX-SAT model.

Context-specific inference. Once a candidate model has been transformed into an ADD, we want to be able to quickly infer the optimal value attainable in a specific context ϕ.

A straightforward way of doing this involves multiplying the ADD that represents the MAX-SAT model with a BDD that represents ϕ, and doing inference on the resulting diagram. However, we have found that this generally takes too long and does not shorten overall model evaluation time. Similarly, performing restrict operations on the model's ADD representation in accordance with ϕ is not fast enough.

Instead, we go through the entire ADD, starting from the root node, and ignoring branches that violate ϕ. The maximum terminal value reached this way is the best attainable value achievable in ϕ.

This process is straightforward when ϕ is a conjunction of literals, as one simply has to ignore branches that violate one of the literals in ϕ. Luckily, this is arguably the most common scenario, as a context typically represents a state of affairs, which is most naturally represented using a conjunction.

When ϕ is a disjunction, or more generally a DNF formula, one can repeat the process above for each conjunctive clause in ϕ. The maximum terminal value reached for any of the conjunctive clauses is then the maximum attainable value in ϕ as a whole.

As an optimization, we perform a precomputation before any example or context is considered, which involves finding and storing all paths leading to the terminal node with the maximum value in the ADD representing the MAX-SAT model. A quick computation can then be made for every context occurring in the training data to determine whether it violates all of the paths leading to said terminal node. If any of these paths is not violated, one knows immediately that the optimal value in the context is the same as the one in the absence of a context, which has been precomputed. In the other case, the procedure detailed above is used.

6 The Genetic Algorithm

As discussed above, the bottleneck in HASSLE-SLS is the evaluation of MAX-SAT models, because it involves *solving* MAX-SAT problems, which is NP-hard. In every iteration of HASSLE-SLS, an entire neighborhood – which frequently consists of several dozen models – is evaluated, before the best neighbor is identified and a *minimal* alteration to the current model is made. This is the crux of the problem: many expensive candidate model evaluations lead only to a minimal step in the search space.

For this reason, we develop an alternative search strategy in the form of a genetic algorithm, which we call HASSLE-GEN. Genetic algorithms form a class of population-based metaheuristic optimization approaches that are loosely inspired by biological evolution [10, 11, 21]. They aim to solve optimization problems by evolving a *population* of candidate solutions, also referred to as *individuals*. They generally consist primarily of genetic selection, mutation and crossover operators. An overview of HASSLE-GEN is given in Algorithm 2. In what follows, we discuss its components in detail.

Algorithm 2 HASSLE-GEN, a genetic algorithm for learning MAX-SAT models from examples.

```
procedure HASSLE-GEN($examples$: a set of labeled contextual examples, $k$: the
    total number of constraints to learn, $q$: the population size, $p_c$: the crossover probability,
    $[p_{m1}, p_{m2}, p_{m3}]$: the mutation probabilities, $g$: the maximum number of generations, $t$:
    the cutoff time)
    Randomly initialize a $population$ of models containing $k$ constraints
    Evaluate $population$
    while generation $< g \wedge$ runtime $< t$ do
        $new\text{-}population \leftarrow$ empty population
        while $new\text{-}population$ not of size $q$ do
            $parent_1$, $parent_2 \leftarrow$ CROWDING-PARENT-SELECTION($population$)
            if RANDOM() $< p_c$ then
                $ind \leftarrow$ CLAUSE-CROSSOVER($parent_1$, $parent_2$, $examples$)
            else
                $ind \leftarrow$ either $parent_1$ or $parent_2$, selected at random
            $ind \leftarrow$ HARDNESS-MUTATION($ind$, $p_{m1}$)
            $ind \leftarrow$ WEIGHT-MUTATION($ind$, $p_{m2}$)
            $ind \leftarrow$ LITERAL-MUTATION($ind$, $p_{m3}$, $examples$)
            $surv_1$, $surv_2 \leftarrow$ CROWDING-SURVIVOR-SELECTION($ind$, $parent_1$, $parent2$)
                ▷ Includes evaluation of $ind$
            Add $surv_1$ and $surv_2$ to $new\text{-}population$
        $population \leftarrow new\text{-}population$
    return best individual in $population$
```

Selection. Selection consists of two components: parent selection and survivor selection. The former is concerned with determining which individuals of the current population to subject to the mutation and crossover operations. The latter is concerned with determining which individuals to keep in the next generation's population and which to discard. Good selection strikes an appropriate balance between exploitation of useful information present in the current population and exploration of new regions of the search space.

Both HASSLE-GEN's parent and survivor selection are determined by its use of a variation of the deterministic crowding scheme [19], which we employ because it is effective at maintaining population diversity, which in turn benefits the search.

Parent selection is straightforward in deterministic crowding: in every generation, each individual is selected to be a parent exactly once. Because HASSLE-GEN's crossover operator only produces a single offspring, this means that in every generation, every individual gives rise to exactly one offspring, together with another parent individual.

Deterministic crowding's survivor selection requires a distance metric d between individuals to be defined. It uses this metric in the following way. Say parents p_1 and p_2 gave rise to offspring o. Then, by the time survivors have to be selected, a matching is made. Parent p_1 is matched to o only if $d(p_1, o) < d(p_2, o)$; otherwise, parent p_2 is matched to o. The offspring and matched closest parent then compete, wherein only the individual with the highest fitness makes it to the next generation. The other parent automatically survives. Here, the fitness of a MAX-SAT model is simply its training set accuracy.

A first consequence of this scheme is that the average fitness of the population never decreases, because a parent never gets replaced by a worse individual. Second, the best individual is automatically kept in the next generation, unless this individual has produced an even better offspring. Finally, and most importantly, there is a strong emphasis on maintaining the initial population's diversity. This is true because an offspring is typically quite similar to its parents, with one of which it has to compete for survival, and because the matching procedure sets up offspring-parent competitions in such a way that the distance between the competitors is minimized. Having new individuals compete for survival with similar old individuals is how diversity is maintained, because this prevents any specific genetic information from quickly taking over the entire population.

What remains to be answered is how the distance metric d is instantiated. We opt for a metric that captures the semantic distance between MAX-SAT models, rather than a syntactic distance. The metric makes use of the notion of an accuracy bit vector.

▶ **Definition 3** (Accuracy bit vector). *Given a list of examples S, a MAX-SAT model M has an associated accuracy bit vector v, which has as many entries as S has examples. For each index i, the entry at index i in v is 1 iff M accurately labels the example at index i in S, and 0 otherwise.*

▶ **Definition 4** (Semantic distance). *Let S be a list of m examples and let M_1 and M_2 be two MAX-SAT models with respective accuracy bit vectors v_1 and v_2. Let their number of correctly labeled examples be s_1 and s_2, respectively. Finally, let d_H be the Hamming distance between v_1 and v_2. Then, the semantic distance d_{Sem} between M_1 and M_2 is*

$$d_{Sem} = \frac{d_H - L}{U - L}, \text{ where } L = |s_1 - s_2| \text{ and } U = m - |s_1 + s_2 - m|$$

This semantic distance metric considers the Hamming distance of the accuracy bit vectors, *relative to a lower and upper bound.* If we were to simply use the Hamming distance itself, we would give inherent preference to matching highly accurate or highly inaccurate models.

Mutation. A mutation operator takes a single individual as input. Usually, it only makes a small modification to the individual, in order to slightly redirect the search or to introduce new genetic information that can then be recombined by crossover operators.

HASSLE-GEN contains three mutation operators, which are all applied to any selected parent. The first operator, named *hardness mutation*, loops over all of the individual's constraints and, independently for each constraint, with probability p_{m1}, changes the constraint from a hard to a soft constraint or vice versa.

The second operator, named *weight mutation*, loops over all of the individual's weighted soft constraints, and independently for each constraint, with probability p_{m2}, replaces the constraint's weight by a weight sampled uniformly at random from $(0, 1]$.

Finally, *literal mutation* is responsible for altering the occurrences of variables in clauses. This operator differs in nature from the first two in that it is more informed; it considers the training data. It takes effect with probability p_{m3}. It starts by randomly selecting one constraint of the individual being mutated. This constraint is the only one to be affected by the mutation. The operator then constructs a neighborhood around this constraint, consisting of all other constraints that differ in only a single variable occurrence (i.e., a single literal appears, disappears or changes its sign), excluding all constraints that already occur in the model. Finally, the operator uses a heuristic way of evaluating all neighboring constraints, and mutates the selected constraint into the best neighboring one. The heuristic evaluation of constraints is based on the notion of *coverage*, where constraints with higher coverage are better according to the heuristic.

▶ **Definition 5** (Covering). *A constraint c is said to cover an example e consisting of instance i and label l if and only if:*
- *l is "solution" and i satisfies c*
- *l is "non-solution" and i does not satisfy c.*

▶ **Definition 6** (Coverage bit vector). *Given a list of examples S, a constraint c has an associated coverage bit vector v, which has as many entries as S has examples. For each index i, the entry at index i in v is 1 iff c covers the example at index i in S, and 0 otherwise. The sum of all of v's entries is called c's coverage.*

Note that a constraint's coverage is only a heuristic measure of its usefulness, as it disregards the weights of soft constraints and the existence of contexts. Still, in preliminary experiments, we found that the use of this heuristic leads to a much more effective operator than one that simply alters literals at random.

Crossover. A crossover operator recombines information from multiple individuals, generally two. The motivation is that by combining individuals that are fit for different reasons, new individuals can be obtained that combine the strengths of both parents.

HASSLE-GEN contains one crossover operator, which we call *constraint crossover*. It takes two parents as input and produces a single offspring. Unlike the mutation operators, constraint crossover keeps the constraints themselves intact, but rearranges which constraints co-occur. It is through this mechanism that HASSLE-GEN is able to take larger steps in the search space than HASSLE-SLS. Like literal mutation, constraint crossover is an informed operator and considers the training data. This allows it to bias its steps towards directions that are likely to improve the model's training set accuracy.

For any pair of parents, constraint crossover takes effect with probability p_c. Given two parents, each containing k constraints, it selects k constraints that are combined in the single offspring. It does not do so blindly, but considers combinations of the constraints' coverage bit vectors.

▶ **Definition 7** (Combining coverage bit vectors). *Let S be a list of examples, and let c_1 and c_2 be constraints with respective associated coverage bit vectors v_1 and v_2. The coverage bit vector v of conjunction $c_1 \wedge c_2$ is attained by performing:*

- *a pairwise AND operation on v_1 and v_2 for all examples in S labeled "solution"*
- *a pairwise OR operation on v_1 and v_2 for all examples in S labeled "non-solution"*

The sum of all of v's entries is called the coverage of $c_1 \wedge c_2$.

One option would be to choose the k constraints taken from the two parents which lead to the largest coverage when combined. However, identifying these constraints involves computing the coverage of $\binom{2k}{k}$ combinations.

Instead, we opt for a sequential selection of constraints that works as follows. First, the constraint with the highest coverage is selected and copied into the offspring. Then follows a repeated selection of the constraint that leads to the highest coverage when combined with the already selected constraints, until k constraints have been selected.

7 Experimental Evaluation

In this section, we thoroughly investigate the following research questions.

Q1 Does the knowledge-compilation-based model evaluation procedure speed up the evaluation of candidate MAX-SAT models?

Q2 When given the same amount of time, is HASSLE-GEN able to learn higher-quality MAX-SAT models than HASSLE-SLS?

Datasets. Each synthetic ground-truth model with k_h hard constraints and k_s soft constraints over n variables is generated such that none of its clauses is entailed by any combination of the other clauses in the model. For each disjunctive clause, the number of literals is chosen uniformly at random from $[1, 5]$. The literals themselves are also selected randomly. For k_s of the generated clauses, a weight is sampled uniformly from $(0, 1]$.

For each generated ground-truth model, a dataset is constructed in two phases. First, a set of conjunctive contexts of $n/2$ literals is generated such that each included context actually affects the maximum attainable value in the ground-truth model. Then, for each context, a specified number of infeasible, suboptimal and solution instances are generated. Infeasible and suboptimal instances are acquired simply by generating random instances that satisfy the context until the desired number of infeasible and suboptimal ones are found. To generate solution instances, a solver is used. To prevent overly similar solution instances, 10 times as many solutions as required are generated for every context, after which the desired amount are sampled at random.

Performance measures. To answer **Q1**, we consider the relative increase in the number of evaluations HASSLE-SLS makes per second when using the novel evaluation procedure compared to when using the original procedure. For example, a speed-up factor of 3.4 would mean that HASSLE-SLS was able to perform 3.4 times as many evaluations with the new evaluation procedure than with the original procedure in the given cutoff time.

To answer **Q2**, we compute the learned model's score, infeasibility and average regret. A model's score is simply its training set accuracy, which was used as training objective. The other two metrics assess the quality of the learned model's solutions with respect to the ground-truth model that was used to generate the training data.

A learned model M's *infeasibility* expresses what proportion of solutions of M are actually infeasible with respect to the ground-truth model M^*. It can be measured exactly by use of *model counting* (MC), i.e., counting the number of solutions to a propositional formula. However, the MAX-SAT models we consider are not just propositional formulas; they also contain weighted soft constraints. For this reason, we require a propositional formula expressing the solutions to the MAX-SAT model. Let M be a MAX-SAT model with hard constraints F_h. Say that $\hat{x}$ is a solution to the model which realizes a value of $\hat{v}$ in satisfied soft constraints. We can then find each subset of soft constraints S_i for which the associated weights sum up to $\hat{v}$. Using this, a propositional formula θ_M expressing the solutions of M can be attained as:

$$\theta_M = F_h \wedge (\bigvee_{S_i} \bigwedge_{\theta_s \in S_i} \theta_s)$$

This formula expresses that an instance is a solution to the model if it satisfies the hard constraints and satisfies one of the subsets of soft constraints that realizes the optimal value. With F_h^* denoting the hard constraints of ground-truth model M^*, the infeasibility of M can be computed as:

$$\inf_{M^*}(M) = \frac{\text{MC}(\theta_M \wedge \neg F_h^*)}{\text{MC}(\theta_M)}$$

The *average regret* of a learned model M with respect to a ground-truth model M^* expresses how good the solutions of M are with respect to the soft constraints of M^*. We compute average regret using only solutions to M that are feasible in M^*. Hence, infeasibility and regret should be considered together to get a complete picture of a model's quality. We generate up to 1000 such feasible instances. For each such instance x, let v^* be the value it realizes with respect to M^*'s soft constraints, and let $\hat{v}^*$ be M^*'s optimal value. The regret of x is then simply $(\hat{v}^* - v^*)/\hat{v}^*$. The average regret of M with respect to M^* is then the average regret over the considered instances. The regret is computed in the *global context*, i.e., outside of any particular context. Thus, achieving low regret requires good generalization across contexts, as the model is evaluated in the global context, but is trained using only contextual data in which the contexts actually matter (i.e., affect the optimal attainable value of the ground-truth model).

Results. To answer **Q1**, we run HASSLE-SLS with a cutoff time of 60 seconds on learning problems of various sizes. We consider sizes similar to the ones considered in [16] and [17]. By default we use learning problems with 10 variables, 8 hard constraints and 8 soft constraints in the ground-truth model and 100 contexts with 1 infeasible, 1 suboptimal and 2 solution examples per context in the dataset. We change each of these properties in turn, while keeping the others at their default values. When increasing the number of total constraints of the ground-truth model, half are hard constraints and half are soft constraints. When increasing the number of examples per context, half are solutions, a quarter are infeasible and a quarter are suboptimal. For each learning problem size, we vary the randomization seed to create 15 different learning problems, over which we average the results.

As shown in Figure 1, the novel ADD-based model evaluation procedure gives rise to a significant speed-up for all learning problem sizes considered. It should be noted that, as the number of variables or the number of constraints increases, the relative speed-up decreases. This is caused by the ADD representation growing in size as the size of the MAX-SAT model increases. On the other hand, when larger training sets are considered, the ADD

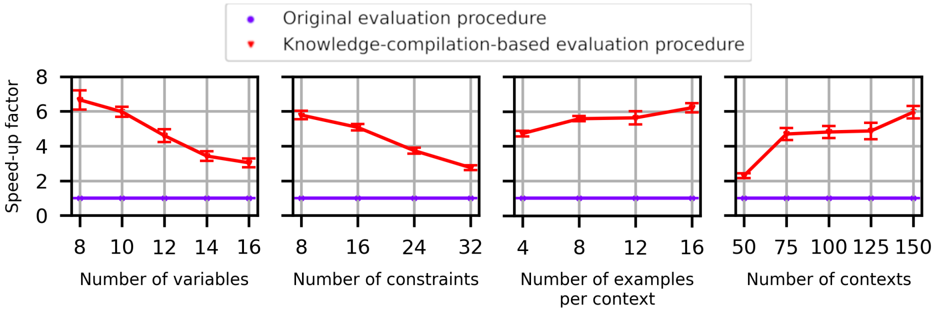


Figure 1 The speed-up in model evaluation realized by using the knowledge-compilation-based model evaluation procedure decreases as the number of variables or the number of constraints in the model increases. On the other hand, the speed-up increases as the example set grows larger.

representation can be reused to a higher degree, increasing the relative speed-up. We can conclude that the knowledge-compilation-based evaluation procedure is most useful when learning relatively small MAX-SAT models, or when learning from large training datasets. This answers **Q1**.

To answer **Q2**, we again vary several aspects of the learning problem in turn, but consider slightly larger learning problems. This time, by default we use 16 variables, 16 hard constraints and 16 soft constraints in the ground-truth model and 100 contexts with 1 infeasible, 1 suboptimal and 2 solution instances per context in the dataset. Again, 15 different randomization seeds are used for each learning problem size. We run HASSLE-SLS and HASSLE-GEN– both using the knowledge-compilation-based evaluation procedure – on each learning problem using a cutoff time of 150 seconds. HASSLE-GEN was run with a population size of 20, $p_c = 0.5$, $p_{m1} = 0.05$, $p_{m2} = 0.05$ and $p_{m3} = 1$, which were determined in a coarse grid search on a separate set of learning problems.

As the first row of Figure 2 shows, HASSLE-GEN consistently achieves a higher score than HASSLE-SLS across many learning problem instances with varying properties. Furthermore, when the size of the considered models increases, the difference between the scores achieved by the search methods grows larger. This can be explained by the fact that HASSLE-SLS only makes a minimal alteration to the current model in every iteration: it changes at most a single literal, the hardness or the weight of a single constraint. These alterations become increasingly inconsequential when considering larger models, resulting in exponentially larger search spaces. By contrast, HASSLE-GEN's constraint crossover allows for much larger steps in the search space, and because it considers the training data, these steps tend to go into an improving direction.

The second and third rows of Figure 2 show how the higher score achieved by HASSLE-GEN in turn translates into lower infeasibility and regret. This shows that the models learned by HASSLE-GEN do not merely achieve a higher training set accuracy than those learned by HASSLE-SLS, but that the solutions they generate are also of a higher quality with respect to the ground-truth model. This answers **Q2**.

8 Conclusion

We have made two contributions aimed at alleviating the runtime issues of the state-of-the-art technique for learning MAX-SAT models from labeled contextual examples. First, to speed up model evaluation, we proposed a knowledge-compilation-based evaluation procedure. Our experiments showed this procedure to be most useful when learning relatively small

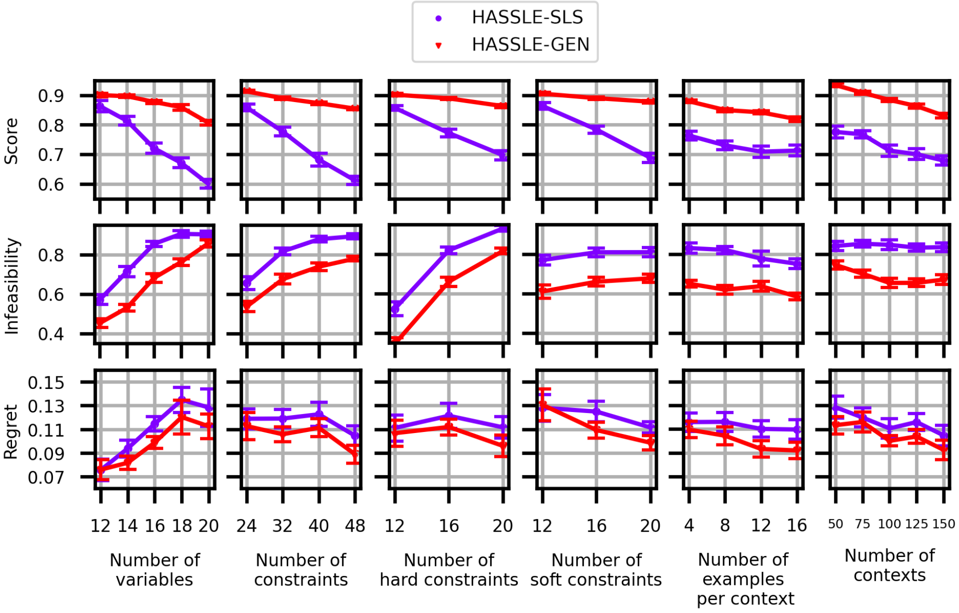


Figure 2 When given the same cutoff time, HASSLE-GEN consistently learns models with a higher score (i.e., training set accuracy) than HASSLE-SLS. This in turn translates into lower infeasibility and lower regret, which means that the models learned by HASSLE-GEN can be used to generate higher-quality solutions than those learned by HASSLE-SLS.

MAX-SAT models or when learning from a large set of training data. Second, to reduce the amount of evaluations required to find a good model, we proposed a genetic algorithm named HASSLE-GEN. In the experiments, HASSLE-GEN consistently beat the state of the art on learning problems of various sizes: when given the same amount of training time, it learned models that can be used to generate higher-quality solutions.

One possible direction for future work is to try to speed up model evaluation even further. In our proposed evaluation procedure, an ADD representing the MAX-SAT model is computed in its entirety, to then be used for context-specific inference. However, some parts of the ADD might not be relevant in any context occurring in the training data, and thus do not strictly have to be computed. A "lazy" decision diagram construction procedure could exploit this fact and lead to even faster evaluation.

Another possible direction is to focus on learning different types of models from examples. One possible extension is to learn maximum satisfiability modulo theories (MAX-SMT) models, where constraints are not limited to disjunctive clauses over binary variables, but can be first-order logic formulas with respect to one or more background theories. Some work has already been done on learning SMT models from examples [14], but learning MAX-SMT models from contextual examples has thus far not been explored. Another possible extension is to learn mixed-integer linear programs (MILP) from contextual data. To this end, recent work [15] has proposed a search strategy that can be seen as a hybrid between stochastic local search and stochastic gradient descent. However, the approach suffers from runtime issues, suggesting that the ideas we proposed here might be of use.

References

1 Roberto Asín Achá and Robert Nieuwenhuis. Curriculum-based course timetabling with SAT and MaxSAT. *Annals of Operations Research*, 218(1):71–91, 2014.

2 R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2):171–206, 1997.

3 Nicolas Beldiceanu and Helmut Simonis. A model seeker: Extracting global constraint models from positive examples. In *International Conference on Principles and Practice of Constraint Programming*, pages 141–157. Springer, 2012.

4 Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Constraint acquisition via partial queries. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

5 Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A SAT-based version space algorithm for acquiring constraint satisfaction problems. In *European Conference on Machine Learning*, pages 23–34. Springer, 2005.

6 Christian Bessiere, Remi Coletta, Barry O'Sullivan, Mathias Paulin, et al. Query-driven constraint acquisition. In *IJCAI*, volume 7, pages 50–55, 2007.

7 Paolo Campigotto, Roberto Battiti, and Andrea Passerini. Learning modulo theories for preference elicitation in hybrid domains. *CoRR*, abs/1508.04261, 2015. `arXiv:1508.04261`.

8 Luc De Raedt, Andrea Passerini, and Stefano Teso. Learning constraints from examples. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

9 Emir Demirović, Nysret Musliu, and Felix Winter. Modeling and solving staff scheduling with partial weighted maxSAT. *Annals of Operations Research*, 275(1):79–99, 2019.

10 Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

11 David Edward Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley Pub. Co., 1989.

12 Michael Huth and Mark Ryan. *Logic in computer science: Modelling and reasoning about systems.* Cambridge University Press, 2004.

13 Samuel Kolb. Learning constraints and optimization criteria. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

14 Samuel Kolb, Stefano Teso, Andrea Passerini, and Luc De Raedt. Learning SMT(LRA) constraints using SMT solvers. In *IJCAI*, volume 18, pages 2333–2340, 2018.

15 Mohit Kumar, Samuel Kolb, Luc De Raedt, and Stefano Teso. Learning mixed-integer linear programs from contextual examples, 2021. `arXiv:2107.07136`.

16 Mohit Kumar, Samuel Kolb, Stefano Teso, and Luc De Raedt. Learning MAX-SAT from contextual examples for combinatorial optimisation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):4493–4500, April 2020. `doi:10.1609/aaai.v34i04.5877`.

17 Mohit Kumar, Samuel Kolb, Stefano Teso, and Luc De Raedt. Learning MAX-SAT from contextual examples for combinatorial optimisation, 2022. `arXiv:2202.03888`.

18 Chu Min Li and Felip Manya. MaxSAT, hard and soft constraints. In *Handbook of satisfiability*, pages 613–631. IOS Press, 2009.

19 Samir W Mahfoud et al. Crowding and preselection revisited. In *PPSN*, volume 2, pages 27–36. Citeseer, 1992.

20 Patrick Mills and Edward Tsang. Guided local search for solving sat and weighted max-sat problems. *Journal of Automated Reasoning*, 24(1):205–223, 2000.

21 Melanie Mitchell. *An introduction to genetic algorithms.* MIT Press, 1998.

22 Tomasz P Pawlak and Krzysztof Krawiec. Synthesis of mathematical programming constraints with genetic programming. In *European Conference on Genetic Programming*, pages 178–193. Springer, 2017.

23 Émilie Picard-Cantin, Mathieu Bouchard, Claude-Guy Quimper, and Jason Sweeney. Learning parameters for the sequence constraint from solutions. In *International Conference on Principles and Practice of Constraint Programming*, pages 405–420. Springer, 2016.

24 Émilie Picard-Cantin, Mathieu Bouchard, Claude-Guy Quimper, and Jason Sweeney. Learning the parameters of global constraints using branch-and-bound. In *International Conference on Principles and Practice of Constraint Programming*, pages 512–528. Springer, 2017.

25 Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. Partial weighted MaxSAT for optimal planning. In *Pacific Rim International Conference on Artificial Intelligence*, pages 231–243. Springer, 2010.

26 Sean Safarpour, Hratch Mangassarian, Andreas Veneris, Mark H Liffiton, and Karem A Sakallah. Improved design debugging using maximum satisfiability. In *Formal Methods in Computer Aided Design (FMCAD'07)*, pages 13–19. IEEE, 2007.

27 Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence*, 171(2-3):107–143, 2007.

Complexity of Minimum-Size Arc-Inconsistency Explanations

Christian Bessiere
CNRS, University of Montpellier, France

Clément Carbonnel
CNRS, University of Montpellier, France

Martin C. Cooper
IRIT, University of Toulouse, France

Emmanuel Hebrard
LAAS CNRS, Toulouse, France

Abstract

Explaining the outcome of programs has become one of the main concerns in AI research. In constraint programming, a user may want the system to explain why a given variable assignment is not feasible or how it came to the conclusion that the problem does not have any solution. One solution to the latter is to return to the user a sequence of simple reasoning steps that lead to inconsistency. Arc consistency is a well-known form of reasoning that can be understood by a human. We consider explanations as sequences of propagation steps of a constraint on a variable (i.e. the ubiquitous revise function in arc consistency algorithms) that lead to inconsistency. We characterize, on binary CSPs, cases for which providing a shortest such explanation is easy: when domains are Boolean or when variables have maximum degree two. However, these polynomial cases are tight. Providing a shortest explanation is NP-hard if the maximum degree is three, even if the number of variables is bounded, or if domain size is bounded by three. It remains NP-hard on trees, despite the fact that arc consistency is a decision procedure on trees. Finally, the problem is not FPT-approximable unless the Gap-ETH is false.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Fixed parameter tractability; Theory of computation → Problems, reductions and completeness

Keywords and phrases Constraint programming, constraint propagation, minimum explanations, complexity

Digital Object Identifier 10.4230/LIPIcs.CP.2022.9

Funding This work was supported by the AI Interdisciplinary Institute ANITI, funded by the French program "Investing for the Future – PIA3" under grant agreement no. ANR-19-PI3A-0004. The first two authors received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952215.

1 Introduction

Constraint Programming (CP) is a technology that allows the user to solve combinatorial problems formulated as constraint networks. A constraint network is characterized by a set of variables taking values in a finite domain that are subject to constraints. Constraints restrict the combinations of values that specified subsets of variables can take. One of the advantages of using CP is that in general constraint networks represent the problem to solve much more compactly than would an integer linear program or a SAT formula. CP formulations are not only compact but also easy to understand for the user thanks to the expressiveness of constraints that allow to remain close to the original problem. However, nowadays, AI becomes even more demanding in terms of *explainability*. A user may want to

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 9; pp. 9:1–9:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

not only understand the formulation of their problem as a constraint network but also to be provided with explanations of why this assignment is the only solution, why that value is not feasible, or why the problem does not have any solution.

An *abductive explanation* for a proposition is often defined as a *prime implicant* of that proposition, i.e. an implicant that cannot be generalized further. For instance, an explanation of a Machine Learning model's prediction is often defined as a minimal subset of features that entails that prediction [16, 10]. Similarly, a *minimal unsatisfiable core* (irreducible unsatisfiable subset of constraints) can be seen as an abductive explanation for unsatisfiability since it is a sufficient and minimal reason for unsatisfiability. At least one term of an abductive explanation must be relaxed in order to change the outcome. This is the viewpoint adopted in many existing approaches. For instance by providing explanations in the form of minimal sets of choices of the user that lead to the given value removal (e.g., product configuration [1]), or explanations in the form of minimal sets of constraints that lead to an inconsistency [11]. The purpose of such approaches is to help the user to repair the inconsistency, not to let them understand why it is an inconsistency.

Intuitively, an explanation is more than a sufficient condition. In particular, if an abductive explanation answers the "*why*" question, it does not answer the "*how*" question. An intuitive definition of an explanation also covers the *demonstration* of how the considered cause has that consequence. For instance, when solving a logic puzzle, we may want to let the user understand why the zebra is necessarily in the middle house, not by providing a set of constraints of the problem that rule out all other positions for the zebra, but by displaying a sequence of simple reasoning steps that lead to that conclusion. This notion of *demonstrative explanation* can be related to proof systems and to the notion of formal proof. A formal proof better explains unsatisfiability by making every step explicit down to axiomatic definitions. For instance, a refutation proof log using the *reverse unit propagation* (RUP) system [8, 9] allows one to formally verify the unsatisfiability of a formula, provided that one can "trust" the application of the unit propagation rule, i.e. trust that a given formula that is refutable via unit propagation is indeed unsatisfiable. This is valid in the context of formal proof verification where each unit propagation refutation can be checked efficiently. However, this may produce very long proofs in which each step might be too complex for an explanation to a non-expert.

We would therefore want to produce demonstrative explanations, allowing a trustworthy verification, however with minimal requirements on the recipient of the explanation. This is of course impossible in general. In [17, 2], the choice was made to provide explanations in the form of sequences of inferences performed by constraint propagation. We consider an even simpler, and also incomplete, proof system: Arc Consistency. Arc consistency has often been considered as a sufficiently strong inference technique on applications where the human is in the loop (configuration [12], logic puzzles [17]).

Our goal is to analyze the complexity of providing the *shortest* possible explanation of arc inconsistency of a problem. For simplicity of presentation, we restrict ourselves to normalized networks of binary constraints. We show that when variables have degree two or domains are Boolean, finding a shortest explanation of arc inconsistency is polynomial. However, the problem is NP-hard in general and the two polynomial cases above are tight. Finding a shortest explanation of arc inconsistency is NP-hard as soon as variables have degree three, even if the number of variables is bounded (even though the problem is obviously polynomial to solve). It is also NP-hard if domain size is bounded by three. Perhaps more surprisingly, it remains NP-hard on trees, where arc consistency is known to be a decision procedure. We also show that there is little hope that we can efficiently find short (if not shortest) explanations: the problem is not FPT-approximable unless the Gap-ETH is false.

2 Background and Definitions

The *constraint satisfaction problem* (*CSP*) involves finding solutions to a constraint network. A *constraint network* (or CSP instance) is defined as a set of n variables $X = X_1, \ldots, X_n$, a set of domains $D = \{D(X_1), \ldots, D(X_n)\}$, where $D(X_i)$ is the finite set of values that X_i can take, and a set C of constraints. A binary constraint $c(X_i, X_j)$ is a binary relation that specifies which combinations of values (tuples) the variables (X_i, X_j) are allowed to take. A CSP is *binary* when all the constraints are binary. A binary CSP is said to be *normalized* if there is at most one constraint per pair of variables. A *degree-2* CSP does not contain any variable involved in more than two constraints. *Arc consistency* (*AC*) is the basic form of inference reasoning on constraint networks. A tuple τ of values on (X_i, X_j) is called a *support* on constraint $c(X_i, X_j)$ for a value $v \in D(X_i)$ (and $\tau[X_j]$ its support in $D(X_j)$) if and only if $\tau[X_i] = v$, $\tau[X_j] \in D(X_j)$ and $\tau \in c(X_i, X_j)$. A value v in $D(X_i)$ is arc consistent if and only if v has a support on every constraint involving X_i. A network is arc consistent if all values in all domains are arc consistent. The operation $revise(X_i, c(X_i, X_j))$, often denoted by $X_i \overset{c}{\leftarrow} X_j$ in the following, removes from $D(X_i)$ all values that do not have any support on $c(X_i, X_j)$. If enforcing arc consistency on a network (that is, applying $revise()$ operations until a fix point is reached) leads to a domain wipe out (i.e. an empty domain), we say that the network is *arc inconsistent*.

▶ **Definition 1** (Arc Inconsistency Explanation). *An* arc inconsistency explanation *for a CSP instance is a sequence of* $revise()$ *operations such that one of the domains is wiped out by the execution of the sequence of* $revise()$ *operations.*

▶ **Definition 2** (Shortest Arc Inconsistency Explanation). *The* shortest arc inconsistency explanation *problem consists in finding an arc inconsistency explanation of minimum length.*

▶ **Example 3** (Explaining the Zebra puzzle). The Zebra puzzle, which may (or may not) be due to Lewis Carroll, has a well known CSP model whereby, for each of the 5 *house colors*, *nationalities*, *beverages*, *cigarette brands*, and *pets*, we have a variable whose value is the number of the corresponding house (e.g., X_{Zebra} stands for the house where the Zebra lives). The constraints are statements such as *The Englishman lives in the red house* or *The Old Gold smoker owns snails.* Moreover, each house has a unique colour, its owner has a unique nationality, drinks a unique beverage, smokes a unique brand, and has a unique pet.

Applying arc consistency on this CSP detects that "**the Kools smoker does not live in the 2nd house**". A demonstrative explanation would be: **The Norwegian lives in the first house**. Since *the Norwegian lives next to the blue house*, then **the 2nd house is blue**. Since *the 2nd house has a single color*, then **it is not yellow**. Since *Kools are smoked in the yellow house*, then **the Kools smoker does not live in the 2nd house**.

Each step corresponds to the arc consistency *revision* of some domain knowledge (in bold) with respect to a constraint (in italic), that is, it corresponds in our framework to the following sequence of $revise()$ operations: $\langle X_{\text{Blue}} \leftarrow X_{\text{Norwegian}}, X_{\text{Yellow}} \overset{\neq}{\leftarrow} X_{\text{Blue}}, X_{\text{Kools}} \overset{=}{\leftarrow} X_{\text{Yellow}}\rangle$.

3 Complexity of Explaining Arc Inconsistency: Structure

We show that if all variables are involved in no more than two constraints, finding shortest arc inconsistency explanations is polynomial. We then show that this class is tight. As soon as we allow a variable to be in the scope of three constraints, the problem becomes NP-hard, even if the CSP has no more than four variables. Perhaps even more surprising, the problem is NP-hard on CSPs structured as trees, despite arc consistency being a decision procedure on trees.

3.1 Tractability on degree-2 CSPs

▶ **Theorem 4.** *Shortest Arc Inconsistency Explanation is solvable in time polynomial in the number of variables and values when restricted to binary normalized networks with maximum degree two.*

Proof. A constraint network of maximum degree two is composed of unconnected cycles and paths. A shortest arc inconsistency explanation clearly always concerns only one of the connected components of the network. An exhaustive search over all connected components only increases complexity by at most a linear factor. Since, furthermore a path can be viewed as a degenerate cycle (a cycle in which one constraint disallows no tuples), it follows that we only need consider the case of a single cycle.

Without loss of generality, we suppose that the cycle is $X_1, \ldots, X_n$, with constraints $c(X_i, X_{i+1})$, where here and in the rest of the proof addition within subscripts is understood to be modulo n, so that for example X_{n+1} actually refers to X_1. We say that $revise()$ operations are clockwise (resp. anticlockwise) if they are of the form $X_{i+1} \leftarrow X_i$ (resp. $X_i \leftarrow X_{i+1}$). We say that a pair of $revise()$ operations R_1, R_2 commute if the two sequences R_1R_2 and R_2R_1 produce the same result. It is easy to verify that the only $revise()$ operations that may not commute are those in which the destination variable of one is the source variable of the other. Furthermore, $revise()$ operations in opposite directions (clockwise and anticlockwise) always commute, even $X_i \leftarrow X_{i+1}$ and $X_{i+1} \leftarrow X_i$. Thus the only pairs of $revise()$ operations that do not commute are of the form $\{X_i \leftarrow X_{i+1}, X_{i+1} \leftarrow X_{i+2}\}$. What's more, if we have the operations $X_i \leftarrow X_{i+1}, X_{i+1} \leftarrow X_{i+2}$ in this order, then the set of value-eliminations cannot decrease if we inverse the order of these two operations.

In a shortest arc inconsistency explanation E, a $revise()$ operation must be useful: it must eliminate a domain value whose elimination is essential for a subsequent $revise()$ operation or for the final domain wipe-out. In the former case, the operation $X_i \leftarrow X_{i+1}$ must be followed later in the sequence by $X_{i-1} \leftarrow X_i$. Let S be the sequence of $revise()$ operations in E between the operation $X_i \leftarrow X_{i+1}$ and the next subsequent occurrence of $X_{i-1} \leftarrow X_i$. By the above discussion on commutativity, we can shift the operation $X_i \leftarrow X_{i+1}$ just after S without *decreasing* the set of value-eliminations since S does not contain $X_{i-1} \leftarrow X_i$. In this way, we can group together all the anticlockwise $revise()$ operations to form a sequence of anticlockwise operations on consecutive edges in the cycle. The same argument holds for clockwise operations which can be grouped together to form a sequence of clockwise operations on consecutive edges in the cycle.

An obvious observation is that a shortest arc inconsistency explanation is necessarily of length bounded by nd, where d is the maximum domain size, since at least one elimination must occur at each operation. Moreover, there are up to n possible starting points for the sequence of clockwise (resp. anticlockwise) operations. Hence a shortest explanation can be found in polynomial time, by exhaustive search over the starting points and lengths of the clockwise/anticlockwise sequences. ◀

3.2 Intractability on CSPs with four variables

The result in Theorem 4 is tight. We show that as soon as we allow variables to have degree 3, finding a shortest explanation becomes NP-hard. This is true even if the number of variables is bounded by four. (Observe that all binary normalized CSPs on three variables have degree at most 2.) We use a reduction from Clique, which is NP-complete [13], to prove hardness.

▶ **Definition 5** (Clique).
Input: *An undirected graph $G = (V, E)$ and an integer k*
Question: *Is there $S \subseteq V$ such that $|S| \leq k$ and for all $i \neq j \in S$, $\{i, j\} \in E$?*

It is noticeable that CSPs with a bounded number of variables have a constant number of possible *revise*() operations available at each step –only 12 in the case of four variables. This is not sufficient to make the problem of finding a shortest explanation easy.

▶ **Theorem 6.** *Shortest Arc Inconsistency Explanation is NP-hard, even on binary normalized networks with four variables.*

▶ **Lemma 7.** *Deciding whether there exists an arc inconsistency explanation of length smaller than or equal to k is NP-complete, even on binary normalized networks with four variables.*

Proof. *Membership.* Given a sequence of *revise*() operations, we decide whether this sequence is an arc inconsistency explanation by executing each *revise*() in the order of the sequence and checking whether one of the domains is empty after these executions. As constraints have bounded arity, executing a *revise*() operation is polynomial, so the whole process is polynomial.

Completeness. We reduce the Clique problem to the problem of deciding whether there is an arc inconsistency explanation of length at most $3n + 3$ for a CSP instance. Let $G = (V, E)$ be a graph with set of vertices $V = \{1, \dots, n\}$.

We construct the CSP instance P_G with four variables $X = \{X_1, X_2, X_3, X_4\}$, all with domain $\{(p, i) : p \in 0..n + 1, i \in 1..n\} \cup \{s_t : t \in 1..k + 1\}$.

We build the set of constraints

$$C = \{c_1(X_1, X_2), c_2(X_1, X_3), c_3(X_2, X_3), X_1 = X_4, X_2 = X_4, X_3 = X_4\}$$

with:

$$\begin{aligned}
c_1(X_1, X_2) = &\ \{((p-1, i), (p, i)) : p \in [0, n+1], \forall i \neq p \in [1, n]\}\\
&\cup \{((p-2, i), (p, i)) : p \in [0, n+1], \forall i \in [1, n]\}\\
&\cup \{(s_t, s_t) : t \in [1, k+1]\}\\
c_2(X_1, X_3) = &\ \{(p-1, i), (p, i)) : p \in [0, n+1], \forall i \in [1, n]\} \cup \{(s_{t-1}, s_t) : t \in [1, k+1]\}\\
c_3(X_2, X_3) = &\ (\{\{(p, i) : p \in [0, n+1], i \in [1, n]\}^2\}\setminus\\
&\quad \{((n+1, i), (n+1, j)) : i = j \ \vee \ \{i, j\} \in E\})\\
&\cup \{\{s_t : t \in [1, k+1]\}^2\}
\end{aligned}$$

The constraint network for a graph with 3 vertices and the edges $\{1, 2\}$ and $\{2, 3\}$ is shown in Figure 1.

We first show that if G contains a k-clique, then, there exists an arc inconsistency explanation of length $3n + 3$ for P_G.

Assume that the set of vertices S is a k-clique. We build the sequence $R(S)$ of *revise*() operations in the following way, and we will say that $R(S)$ *encodes* the set S, since there is a one-to-one mapping between subsets $S \subseteq V$ and this type of explanation:

- If $p \notin S$, the $(3p - 2)$th element in the sequence $R(S)$ is $X_2 \overset{c_1}{\leftarrow} X_1$, the $(3p - 1)$th element is $X_4 \overset{=}{\leftarrow} X_2$, and the $(3p)$th element is $X_1 \overset{=}{\leftarrow} X_4$.
- If $p \in S$, the $(3p - 2)$th element in the sequence $R(S)$ is $X_3 \overset{c_2}{\leftarrow} X_1$, the $(3p - 1)$th element is $X_4 \overset{=}{\leftarrow} X_3$, and the $(3p)$th element is $X_1 \overset{=}{\leftarrow} X_4$.

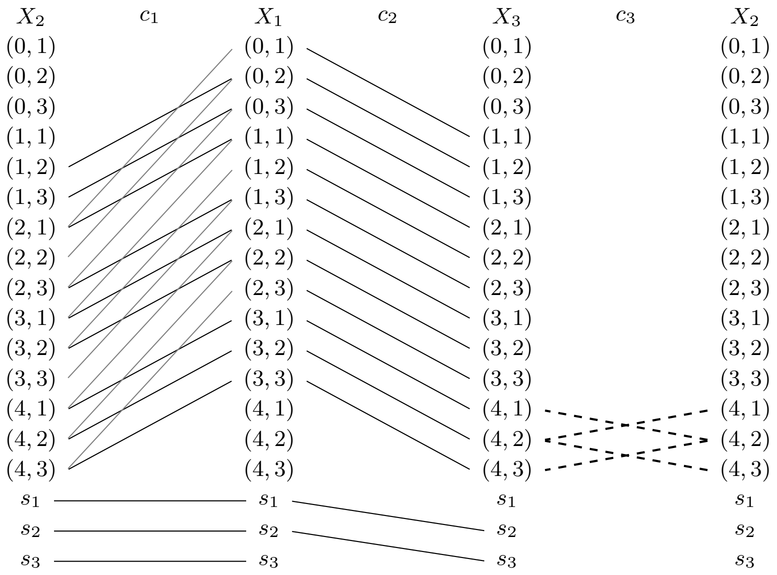


Figure 1 The CSP P_G, reduction of the graph $G = (\{1,2,3\}, \{(1,2),(2,3)\})$. Solid edges represent allowed tuples for c_1 and c_2, while dashed edges stand for forbidden tuples of c_3. The equality constraints are not represented. There are two explanations of Arc-Inconsistency of length 12. The first *encodes* the clique $\{2,3\}$ with the *revise*() operations $X_2 \overset{c_1}{\leftarrow} X_1$, $X_3 \overset{c_2}{\leftarrow} X_1$, $X_3 \overset{c_2}{\leftarrow} X_1$ at positions 1, 4, and 7 in the sequence. The second *encodes* the clique $\{1,2\}$ with the revision operations $X_3 \overset{c_2}{\leftarrow} X_1$, $X_3 \overset{c_2}{\leftarrow} X_1$, $X_2 \overset{c_1}{\leftarrow} X_1$ at positions 1, 4, and 7.

Then the last three elements in the sequence $R(S)$ are $X_2 \overset{c_1}{\leftarrow} X_1$, $X_3 \overset{c_2}{\leftarrow} X_1$, and $X_2 \overset{c_3}{\leftarrow} X_3$. In the following, the subsequence composed of the $(3p-2)$th, the $(3p-1)$th, and the $(3p)$th operations (that is, $\langle X_2 \overset{c_1}{\leftarrow} X_1, X_4 \overset{=}{\leftarrow} X_2, X_1 \overset{=}{\leftarrow} X_4\rangle$ or $\langle X_3 \overset{c_2}{\leftarrow} X_1, X_4 \overset{=}{\leftarrow} X_3, X_1 \overset{=}{\leftarrow} X_4\rangle$), is called the pth *iteration*.

Before each iteration $p \in \{1, \dots, n\}$ of three domain revisions, the invariants are:

$$(q,i) \notin D(X_1) \ \forall q < p-1, \forall i \in [1,n] \tag{1}$$

$$s_j \in D(X_1) \iff k+1 \geq j > |S \cap \{0, \dots, p-1\}| \tag{2}$$

$$(p-1,i) \in D(X_1) \iff i \in S \cup \{p, \dots, n\} \tag{3}$$

All invariants are verified before entering iteration $p = 1$. For each one, we show that if it is true before entering iteration $p \geq 1$ then it remains true before entering iteration $p+1$.

Invariant 1: Notice that a value $(q,i) \in D(X_2)$ (resp. $D(X_3)$) is only supported by values $(q',i) \in D(X_1)$ such that $q' < q$. If Invariant 1 is true before iteration p, then when revising the domain of either X_2 or X_3, $D(X_1)$ contains no value (q,i) with $q < p-1$ and therefore all values $(p-1,i)$ are removed from $D(X_2)$ (resp. $D(X_3)$). The revisions w.r.t. equality constraints make sure that this is propagated back to $D(X_1)$.

Invariant 2. Notice that a value $s_t \in D(X_3)$ is only supported by value $s_{t-1} \in D(X_1)$, whereas the tuple (s_t, s_t) is a support in all other constraints. If Invariant 2 is true before iteration p, then either $p \in S$ in wich case the operation $X_3 \overset{c_2}{\leftarrow} X_1$ removes the value s_j (with $j = |S \cap \{0, \dots, p-1\}| + 1$) from $D(X_3)$ since the value s_{j-1} was its only support and is not in $D(X_1)$; or $X_2 \overset{c_1}{\leftarrow} X_1$ removes no s value and $|S \cap \{0, \dots, p-1\}|$ does not change.

Invariant 3. For any $i \in [1,n]$:

If $i > p$, then we have $(p-1, i) \in D(X_1)$ which is a support for (p, i) w.r.t. c_1 and c_2 hence (p, i) is not removed and the invariant holds because $i \in \{p+1, \ldots, n\}$.

If $i < p$, notice that by Invariant 1, the tuple $((p-2, i), (p, i))$ cannot be a support for $(p, i) \in D(X_2)$. Therefore, both constraints c_1 and c_2 have the same unique potential support for the value (p, i) (in $D(X_2)$ and $D(X_3)$ respectively): $((p-1, i), (p, i))$. So we have: "$(p-1, i) \in D(X_1)$ before iteration p" iff "$(p, i) \in D(X_1)$ before iteration $p+1$". In addition, $i \in S \cup \{p, \ldots, n\} \iff i \in S \cup \{p+1, \ldots, n\}$ because $i < p$. Finally, by the induction hypothesis we have "$(p-1, i) \in D(X_1)$ before iteration p" iff $i \in S \cup \{p, \ldots, n\}$, and hence by transitivity: "$(p, i) \in D(X_1)$ before iteration $p+1$" iff $i \in S \cup \{p+1, \ldots, n\}$.

If $i = p$, there are two cases: If $p \in S$, then the first operation at iteration p is $X_3 \overset{c_2}{\leftarrow} X_1$, (p, i) is not removed since it is supported by $(p-1, i)$, and the invariant is true at iteration $p+1$ since $i \in S$. If $p \notin S$, then the first operation at iteration p is $X_2 \overset{c_1}{\leftarrow} X_1$, (p, i) is removed, and the invariant is true at iteration $p+1$ since $i \notin S \cup \{p+1, \ldots, n\}$.

After n iterations, the invariants hold for $p = n+1$ (i.e. after the $3n$-th operation) and hence $D(X_1)$ is $\{(n, i) \forall i \in S\} \cup \{(n+1, i) \forall i\} \cup \{s_{k+1}\}$. The call to $X_2 \overset{c_1}{\leftarrow} X_1$ then yields $D(X_2) = \{(n+1, i) \forall i \in S\} \cup \{s_{k+1}\}$ and the call to $X_3 \overset{c_2}{\leftarrow} X_1$ yields $D(X_3) = \{(n+1, i) \forall i \in S\}$. Therefore, the last call to $X_2 \overset{c_3}{\leftarrow} X_3$ produces a wipe-out, since on layer $n+1$, the remaining vertices stand for a clique of G and the allowed tuples are non-edges of G.

We then prove that if G does not contain any k-clique, then the shortest arc inconsistency explanation for P_G is of length strictly greater than $3n+3$. We first show that the shortest explanation must use constraint c_3, then we show that only explanations that encode a set $S \subseteq V$ (as defined above) such that S is a clique of size k of G can be the shortest.

Suppose first that the constraint c_3 does not appear in any *revise*() of the explanation. By construction, the values (p, i) are organized in layers, where a layer q is the set of values $(q, i), \forall i$. Wiping out the domain of a variable requires removing the $n+2$ layers 0 to $n+1$ from its domain. Moreover, removing a layer q from X_1 (resp. X_2 or X_3) requires having already removed layer $q+1$ (resp. $q-1$) from X_2 or X_3 (resp. X_1). Removing a layer q from X_4 requires having already removed layer q from X_1, X_2, or X_3. Hence, removing a layer q from a variable requires iteratively removing layers 0 to $q-1$ or $n+1$ down to $q+1$ from other variables. The only way to do that is to execute a sequence of *revise*() operations looping on a cycle of variables $\{X_1, X_2, X_4\}$, or on $\{X_1, X_3, X_4\}$, or both. Looping in the order $\langle X_1 \overset{c_1}{\leftarrow} X_2, X_4 \overset{=}{\leftarrow} X_1, X_2 \overset{=}{\leftarrow} X_4\rangle$ or $\langle X_1 \overset{c_2}{\leftarrow} X_3, X_4 \overset{=}{\leftarrow} X_1, X_3 \overset{=}{\leftarrow} X_4\rangle$ removes layers from $n+1$ down to q, whereas looping in the order $\langle X_2 \overset{c_1}{\leftarrow} X_1, X_4 \overset{=}{\leftarrow} X_2, X_1 \overset{=}{\leftarrow} X_4\rangle$ or $\langle X_3 \overset{c_2}{\leftarrow} X_1, X_4 \overset{=}{\leftarrow} X_3, X_1 \overset{=}{\leftarrow} X_4\rangle$ removes layers from 0 up to q. We can then compute the number of *revise*() operations necessary to remove a layer q from a variable given the order in which we loop. If we execute *revise*() operations in the orders $\langle X_1 \overset{c_1}{\leftarrow} X_2, X_4 \overset{=}{\leftarrow} X_1, X_2 \overset{=}{\leftarrow} X_4\rangle$ or $\langle X_1 \overset{c_2}{\leftarrow} X_3, X_4 \overset{=}{\leftarrow} X_1, X_3 \overset{=}{\leftarrow} X_4\rangle$, layer q is removed from the domain of X_1 (resp. X_2/X_3, or X_4) in $3(n+1-q)+1$ operations (resp. $3(n+1-q)+3$, or $3(n+1-q)+2$ operations). If we execute *revise*() operations in the orders $\langle X_2 \overset{c_1}{\leftarrow} X_1, X_4 \overset{=}{\leftarrow} X_2, X_1 \overset{=}{\leftarrow} X_4\rangle$ or $\langle X_3 \overset{c_2}{\leftarrow} X_1, X_4 \overset{=}{\leftarrow} X_3, X_1 \overset{=}{\leftarrow} X_4\rangle$, layer q is removed from the domain of X_1 (resp. X_2/X_3, or X_4) in $3q+3$ operations (resp. $3q+1$, or $3q+2$ operations). As wiping out a domain requires, given a value q, removing layers 0 to q from below and layers $n+1$ down to $q+1$ from above, we conclude that a domain wipe out, on either X_1, X_2, X_3, or X_4, requires at least $3n+4$ *revise*() operations. This means that there does not exist any arc inconsistency explanation for P_G of length smaller than or equal to $3n+3$ if we do not use c_3 in the explanation.

Hence, we must use c_3. However, by construction of c_3, every value in $D(X_2)$ (resp. $D(X_3)$) is supported as long as at least one value (p, i) with $p \in [0, n]$, and any value s_t is in the domain of $D(X_3)$ (resp. $D(X_2)$). In other words, to remove a layer with a revise on c_3, the domains of X_2 and X_3 must only contain (p, i) values from layer $n + 1$. This requires us to remove all layers from 0 to $n - 1$ from X_1 by executing n loops by a sequence of *revise*() operations $\langle X_2/X_3 \leftarrow X_1, X_4 \leftarrow X_2/X_3, X_1 \leftarrow X_4\rangle$ for a cost of $3n$ operations, plus a $X_2 \overset{c_1}{\leftarrow} X_1$ and a $X_3 \overset{c_2}{\leftarrow} X_1$ to remove layer n from X_2 and X_3. In other words, it must be a sequence of *revise*() operations that *encodes* a set, i.e., $R(S)$ for some set $S \subseteq \{1, \ldots, n\}$. Now, suppose that S is not a clique and let i_1 and i_2 be two non-adjacent vertices in S. By Invariant 3, at iteration $n + 1$, we have $(n, i_1) \in D(X_1)$ and $(n, i_2) \in D(X_1)$ and hence after operations $X_2 \overset{c_1}{\leftarrow} X_1$ and $X_3 \overset{c_2}{\leftarrow} X_1$, we have $(n + 1, i_1) \in D(X_2)$ and $(n + 1, i_2) \in D(X_3)$. Therefore, neither $X_2 \overset{c_3}{\leftarrow} X_3$ nor $X_3 \overset{c_3}{\leftarrow} X_2$ would fail, and at least one more operation is necessary. Finally, suppose that $|S| < k$. Then by Invariant 2, at iteration $n + 1$, we have $s_k \in D(X_1)$ and hence after operations $X_2 \overset{c_1}{\leftarrow} X_1$ and $X_3 \overset{c_2}{\leftarrow} X_1$, we have $s_{k+1} \in D(X_2)$ and $s_{k+1} \in D(X_3)$. Therefore, at least one more operation is necessary. Consequently, the number of operations can be equal to $3n + 3$ only if S is a clique of size k of G. ◀

3.3 Intractability and inapproximability on trees

We have seen in Section 3.2 that Shortest Arc-Inconsistency Explanation is already NP-hard on networks with four variables. This result does not completely settle the intractability of the problem. For example, it is still possible that a polynomial-time algorithm exists for some broad generalization of degree-2 networks that does not contain 4-cliques (for instance, networks of treewidth 2). We show that it is not the case. We use a simple reduction from Dominating Set, which is NP-complete [7], to derive NP-hardness of Shortest Arc-Inconsistency Explanation, even on trees.

▶ **Definition 8** (Dominating Set).
Input: *An undirected graph $G = (V, E)$ and an integer k*
Question: *Is there $S \subseteq V$ such that $|S| \leq k$ and for all $i \in V$, there is $j \in S$ with $\{i, j\} \in E$?*

The NP-hardness of Shortest Arc-Inconsistency Explanation on trees circumscribes even more tightly the degree-2 tractability island of Section 3.1. However, these NP-hardness results do not rule out efficient approximation algorithms nor fixed-parameter tractable algorithms, which could be satisfactory for applications where only short explanations are worth computing and optimality is not strictly necessary. We again show that such desirable scenarios are not possible. We show that our reduction from Dominating Set can be used to derive (conditional) fixed-parameter inapproximability of Shortest Arc-Inconsistency Explanation.

We must briefly introduce some terminology before we can formally present the result. A minimization problem $\mathcal{P}$ is *fpt-approximable* [4] if there exist computable functions $f, \rho : \mathbb{N} \rightarrow \mathbb{R}_{\geq 1}$ such that $n \cdot \rho(n)$ is nondecreasing and an algorithm A that, given as input a non-negative integer k and an instance I of $\mathcal{P}$ that has a solution of cost at most k, computes a solution to I of cost at most $k \cdot \rho(k)$ in time $f(k) \cdot |I|^{O(1)}$. Here, ρ is the approximation ratio and f is possibly exponential. Note that if a problem is not FPT-approximable, then no such algorithm A exists for *any* computable functions f and ρ; such problems are sometimes called *completely inapproximable* [15].

Our FPT-inapproximability result is conditional on a complexity hypothesis known as the *Gap-ETH* [6, 14], which states that there exists a constant $\epsilon > 0$ such that no algorithm with runtime $2^{o(n)}$ can distinguish satisfiable 3-SAT instances from those in which no assignment satisfies a $(1-\epsilon)$ fraction of the clauses. It has been shown recently [3] that the Minimum Dominating Set problem (which consists in finding the smallest dominating set in a graph) is not FPT-approximable unless the Gap-ETH is false.

▶ **Lemma 9.** *Deciding whether there exists an arc inconsistency explanation of length smaller than or equal to k is NP-complete, even on binary tree-structured normalized constraint networks.*

Proof. *Membership.* As in Lemma 7.

Completeness. We reduce the Dominating Set problem to the problem of deciding whether there is an arc inconsistency explanation of length at most k for a CSP instance.

Let $G = (V, E)$ be a graph, $V = \{v_1, \dots, v_n\}$. We construct a constraint network P_G as follows: the set of variables is $\{Y, X_1, \dots, X_n\}$, where the domain of Y is $\{v_1, \dots, v_n\}$ and the domain of each X_i is $\{v_i\}$, and P_G contains a constraint $c(Y, X_i) = \{(v_j, v_i) : \{v_i, v_j\} \notin E \text{ and } v_i \neq v_j\}$ for all $i \geq 1$. An example of this reduction is shown in Figure 2. We claim that G has a dominating set of size k if and only if P_G has an arc-inconsistency explanation of length k.

If G has a dominating set S of size k, then let R be a sequence containing every operation $Y \leftarrow X_i$ such that v_i belongs to S. Since every $v_j \in V$ is dominated by some $v_k \in S$ (which is either v_j itself or one of its neighbours), by construction v_j is removed from $D(Y)$ by $Y \leftarrow X_k$. Therefore $D(Y)$ is empty at the end of the sequence and R is an arc-inconsistency explanation of length k.

Conversely, if R is a minimal arc-inconsistency explanation of P_G of length k then we can assume that it is a sequence of operations of the form $Y \leftarrow X_i$. (Since each $D(X_i)$ contains a single value, only the last operation could be $X_i \leftarrow Y$ for some i, and in that case it can be replaced with $Y \leftarrow X_i$.) Then, the set $S = \{v_i : Y \leftarrow X_i \text{ occurs in } R\}$ must be a dominating set of size k: at the end of R every $v_j \in D(Y)$ has been pruned by some operation $Y \leftarrow X_k$, and every value removed at this step is by construction dominated by v_k in G.

P_G is a tree-structured constraint network and can be constructed in polynomial time from G. Therefore, Shortest Arc-Inconsistency Explanation is NP-hard on such networks. ◀

▶ **Theorem 10.** *Shortest Arc Inconsistency Explanation is NP-hard and not FPT-approximable unless the Gap-ETH is false, even on binary tree-structured normalized constraint networks.*

Proof. In the reduction of the proof of Lemma 9, the size-k dominating sets of G are in one-to-one correspondence with arc-inconsistency explanations of P_G of length k. Furthermore, the dominating set corresponding to an explanation can be computed in polynomial time, so any FPT-approximation algorithm for Shortest Arc-Inconsistency Explanation translates into one for Minimum Dominating Set. By the results of [3], this would imply that the Gap-ETH is false. ◀

As a final remark, we note that the same inapproximability result can be established under the weaker (and more conventional) complexity hypothesis FPT $\neq$ W[2]. However, the proof is significantly more involved and has been left out for the sake of brevity.

4 Complexity of Explaining Arc Inconsistency: Domain Size

We show that finding shortest arc inconsistency explanations is polynomial on binary normalized CSPs with Boolean domains. Again, this class is tight: As soon as we allow three

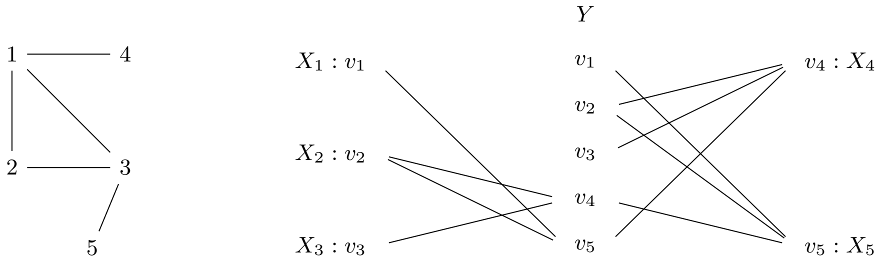


Figure 2 Left: a graph G. Right: the constraint network P_G in the proof of Lemma 9.

values per domain, the problem becomes NP-hard.

4.1 Tractability on Boolean domains

▶ **Theorem 11.** *Shortest Arc Inconsistency Explanation is solvable in polynomial time when restricted to binary normalized networks with all domains of size at most two.*

Proof. Let $P = (X, D, C)$ be a binary CSP with domain size at most two. We assume, without loss of generality, that all domains $D(X_i)$ are non-empty subsets of $\{0, 1\}$ and that no constraint relation is empty. Let X_r be the variable at which a domain wipe-out occurs in a shortest arc inconsistency explanation. Complexity is only multiplied by n if we perform an exhaustive search over all possible variables X_r, so in the following we consider X_r to be fixed. We construct a directed causal graph G_P in which shortest arc inconsistency explanations correspond to particularly simple subgraphs. In G_P there are two types of vertices: source-variable vertices X_i^s $(i = 1, \dots, n)$, and variable-value vertices $\langle X_i, a\rangle$ $(i = 1, \dots, n,\ a \in \{0, 1\})$. G_P has the following directed edges: $(X_i^s, \langle X_j, b\rangle)$ (for all i, j, b such that $b \in D(X_j)$ has no support in $D(X_i)$), and $(\langle X_i, a\rangle, \langle X_j, b\rangle)$ (for all i, j, a, b such that $a \in D(X_i)$ is the only support of $b \in D(X_j)$). Each arc corresponds to a possible revise operation: $(X_i^s, \langle X_j, b\rangle)$ corresponds to the elimination of b from $D(X_j)$ since it has no support in $D(X_i)$, and $(\langle X_i, a\rangle, \langle X_j, b\rangle)$ corresponds to the elimination of b from $D(X_j)$ when its unique support $a \in D(X_i)$ has been eliminated. An example of the causal graph for a simple CSP is shown in Figure 3.

Let R be a shortest arc inconsistency explanation, and let X_r be the variable at which a wipe-out occurs. By minimality of R, each revise operation in R eliminates a value from a domain. Indeed, each operation, except possibly the last, eliminates exactly one value otherwise there would be a domain wipe-out before the end of R. Furthermore, the only way that the final revise operation $X_r \leftarrow X_i$ of R can cause the simultaneous elimination of both 0 and 1 from $D(X_r)$ (without there already being a wipe-out at $D(X_i)$) is that (1) some value $b \in D(X_r)$ never had any support at X_i and (2) the other value $1{-}b$ lost its unique support a at X_i by a previous operation in R. We can deduce from (1) and (2) that just before the execution of $X_r \leftarrow X_i$, the value $1{-}a$ in $D(X_i)$ has no support at X_r. This implies that we can replace the last operation $X_r \leftarrow X_i$ of R by its inverse operation $X_i \leftarrow X_r$ to produce an arc inconsistency explanation of the same length as R but in which the final operation eliminates a single value (namely $1{-}a$ from $D(X_i)$ leading to a wipe-out at X_i).

For any revise operation in R, eliminating b from $D(X_j)$, there is a corresponding arc (u, v) in G_P where v is the vertex $\langle X_j, b\rangle$ and u is the cause of the elimination of b from $D(X_j)$. By the above argument, we can assume that each revise operation in R corresponds to a single elimination and hence a single arc in G_P. Let G_R be the subgraph of G_P consisting

$X_1 < X_2$
$X_2 \leq X_3$
$X_3 \leq X_4$
$X_4 \leq X_5$
$X_5 \leq X_6$
$X_6 \leq X_1$
$X_2 \neq X_4$

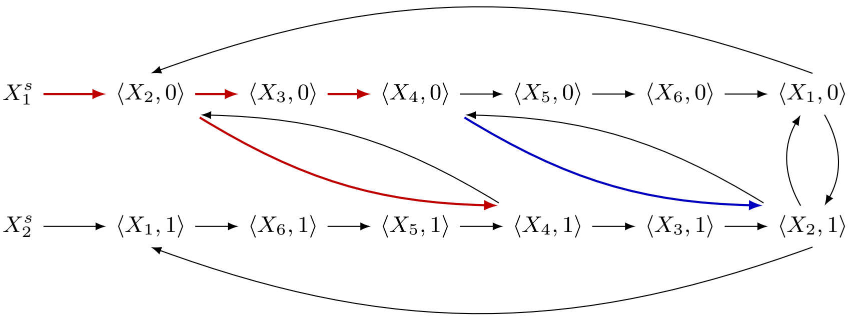


Figure 3 Left: a Boolean binary CSP P (the domain of every variable is $\{0, 1\}$). Right: the causal graph G_P of the proof of Theorem 11. The shortest explanation involves the two paths in red originating from X_1^s and corresponds to the sequence $\langle X_2 \leftarrow X_1, X_3 \leftarrow X_2, X_4 \leftarrow X_3, X_4 \leftarrow X_2\rangle$.

of the arcs corresponding to the operations of R. Let X_r be again the variable at which a wipe-out occurs at the end of R. For each $a \in D(X_r)$, in G_R there must be a directed path P_a from a source-variable vertex to $\langle X_r, a\rangle$. By minimality, the set of arcs of G_R is the union of the set of arcs of P_a ($a \in D(X_r)$). Since each elimination has a unique cause (given by the arc corresponding to the revise operation in R producing the elimination), the in-degree of each vertex in G_R is at most one. Furthermore, source-variable vertices have in-degree 0. It follows that P_0 and P_1 can only possibly share arcs along an initial common subpath.

If $D(X_r)$ is a singleton $\{a\}$, then G_R must be a shortest path in G_P from a source-variable vertex to $\langle X_r, a\rangle$ and hence can be found in polynomial time by a standard shortest-path algorithm. So now suppose that $D(X_r) = \{0, 1\}$. If the set of edges of P_0 and P_1 are disjoint then P_0 and P_1 must both be shortest paths in G_P from source-variable vertices to $\langle X_r, 0\rangle$ and $\langle X_r, 1\rangle$, respectively. If P_0 and P_1 have an initial common subpath, then they must diverge at some vertex v of G_P, the common initial subpath is a shortest path in G_P from a source-variable vertex to v and the remaining divergent paths P_0' and P_1' are shortest paths from v to $\langle X_r, 0\rangle$ and $\langle X_r, 1\rangle$, respectively. By an exhaustive search over the $O(n)$ vertices v of G_P, we can determine the paths P_0 and P_1 in polynomial time. ◀

It is interesting to note that in the proof of Theorem 11, one of the paths P_0', P_1' may actually be empty. In this case, G_R is a path (either P_0 or P_1). This occurs if the elimination of a from $D(X_r)$ triggers a sequence of revise operations that leads to the elimination of $1-a$ from $D(X_r)$. Another interesting point is that if P_0', P_1' are both non-empty, then the revise operations corresponding to P_1' can all be inversed (i.e. each $X_i \leftarrow X_j$ becomes $X_j \leftarrow X_i$) and their order reversed in R to produce an alternative shortest arc inconsistency proof $\tilde{R}$ which ends in a wipe-out at the variable X_k at which P_0' and P_1' diverged. For instance, the sequence $\langle X_2 \leftarrow X_1, X_3 \leftarrow X_2, X_4 \leftarrow X_3, X_2 \leftarrow X_4\rangle$ is also a shortest explanation in the example of Figure 3. In this case, $G_{\tilde{R}}$ is a path (obtained in the example by using the edge in blue $(\langle X_4, 0\rangle, \langle X_2, 1\rangle)$ instead of $(\langle X_2, 0\rangle, \langle X_4, 1\rangle)$). Hence, we can optimise since the exhaustive search over vertices v is unnecessary.

4.2 Intractability on domains with three values

▶ **Theorem 12.** *Shortest Arc Inconsistency Explanation is NP-hard, even on binary normalized networks with all domains of size at most three.*

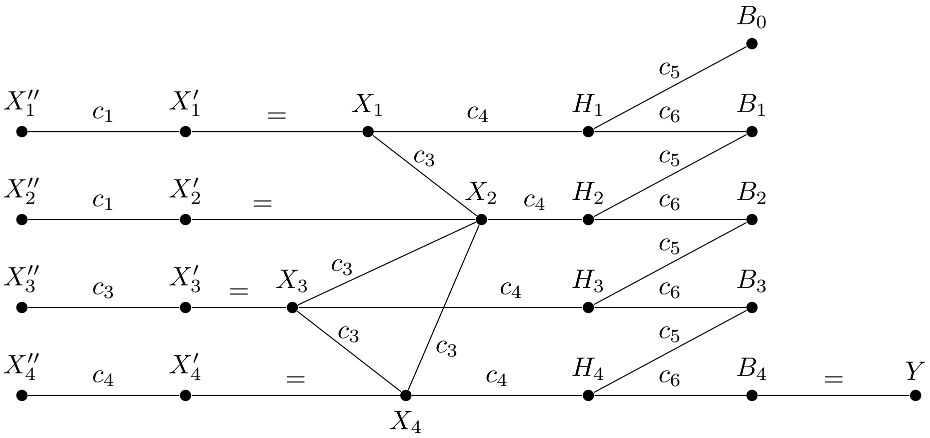


Figure 4 The constraint network P_G in the proof of Lemma 13 when looking for a dominating set in the graph $G = (\{1,2,3,4\}, \{(1,2),(2,4),(2,3),(3,4)\})$.

▶ **Lemma 13.** *Deciding whether there exists an arc inconsistency explanation of length smaller than or equal to k is NP-complete, even on binary normalized networks with all domains of size at most three.*

Proof. *Membership.* As in Lemma 7.

Completeness. We reduce the DOMINATING SET problem (whether a graph G has a dominating set of size at most k) to the problem of deciding whether there is an arc inconsistency explanation of length at most $4n + k + 1$ for a CSP instance. Let $G = (V, E)$ be a graph with $V = \{1, \dots, n\}$.

We construct the CSP instance P_G with $5n + 2$ variables

$$X = \{X_1, \dots, X_n, X_1', \dots, X_n', X_1^{''}, \dots, X_n^{''}, H_1, \dots, H_n, B_0, \dots, B_n, Y\}$$

all with domain $\{0,1,2\}$ except B_0 whose domain is $\{0\}$ and Y whose domain is $\{2\}$.

We build the set of constraints

$$\begin{aligned} C \quad = \quad & \{c_1(X_i^{''}, X_i') : i \in [1,n]\} \ \cup \ \{c_2(X_i', X_i) : i \in [1,n]\} \\ & \cup \{c_3(X_i, X_j) : \{i,j\} \in E\} \ \cup \ \{c_4(X_i, H_i) : i \in [1,n]\} \\ & \cup \{c_5(B_{i-1}, H_i) : i \in [1,n]\} \ \cup \ \{c_6(H_i, B_i) : i \in [1,n]\} \ \cup \ \{B_n = Y\}\} \end{aligned}$$

where

$$c_1(X_i^{''}, X_i') = \{(0,0)\}$$

$$c_2(X_i', X_i) = \{(0,0),(1,1),(2,2)\}$$

$$c_3(X_i, X_j) = \{0,1,2\} \times \{0,1,2\} \setminus \{(0,2),(2,0)\}$$

$$c_4(X_i, H_i) = \{0,1,2\} \times \{0,1,2\} \setminus \{(0,1),(1,1)\}$$

$$c_5(B_{i-1}, H_i) = \{0,1,2\} \times \{0,1,2\} \setminus \{(0,2),(1,2)\}$$

$$c_6(H_i, B_i) = \{0,1,2\} \times \{0,1,2\} \setminus \{(0,2)\}$$

The constraint network is shown in Figure 4 for a graph with $n = 4$ vertices and 4 edges.

We first prove that if G contains a k-dominating set, then there exists an arc inconsistency explanation of length $4n + k + 1$ for P_G. Assume that the set of vertices S is a k-dominating set. We build the sequence R of *revise*() operations in the following way. The first k elements in R are $X'_i \overset{c_1}{\leftarrow} X''_i$ for each vertex i in S. The k next elements in R are $X_i \overset{c_2}{\leftarrow} X'_i$, again for vertices i in S. After those $2k$ *revise*() operations, for all i in S, $D(X_i) = \{0\}$. Then, for each vertex j in $V \setminus S$, R contains $X_j \overset{c_3}{\leftarrow} X_i$, where $i \in S$ and $\{i, j\} \in E$. We know such a vertex i exists for each j because S is a dominating set. After those additional $n-k$ *revise*() operations, for all i not in S, $D(X_i) = \{0, 1\}$. The n next elements in R are $H_i \overset{c_4}{\leftarrow} X_i$, removing value 1 from $D(H_i)$ because $2 \notin D(X_i)$. The $2n$ next elements in R are $\langle H_i \overset{c_5}{\leftarrow} B_{i-1}, B_i \overset{c_6}{\leftarrow} H_i \rangle$ in increasing order of i from 1 to n. Each $H_i \overset{c_5}{\leftarrow} B_{i-1}$ removes value 2 from $D(H_i)$ if $2 \notin D(B_{i-1})$ and $B_i \overset{c_6}{\leftarrow} H_i$ removes value 2 from $D(B_i)$ if $1, 2 \notin D(H_i)$. As $B_0 = 0$ and value 1 has already been removed from all H_i's domains, those $2n$ *revise*() remove value 2 from the domain of all B_i. Finally, after these $2k + (n-k) + n + 2n = 4n + k$ *revise*() operations, the last element in R, $Y \overset{=}{\leftarrow} B_n$, wipes out the domain of Y and proves arc inconsistency.

We then prove that if there exists an arc inconsistency explanation for P_G of length $4n+k+1$, then G contains a k-dominating set. We first observe that if we remove $c_5(B_0, H_1)$ or $B_n = Y$ from P_G, the instance becomes satisfiable. (B_0 is necessary to trigger removals of value 2 from the H_is and Y to trigger removals of value 0.) Hence, no wipe out can occur without executing $2n + 1$ *revise*() operations on the path from B_0 to Y. Furthermore, if a single variable H_i still has value 1 in its domain, the propagation of removals stops. As a result, value 2 needs to be removed from all X_is and a *revise*() needs to be executed on the n constraints c_4. We then have $n + k$ remaining available operations to remove value 2 from all X_is. If we do these removals thanks to the sequence $\langle X'_i \overset{c_1}{\leftarrow} X''_i, X_i \overset{c_2}{\leftarrow} X'_i \rangle$, it costs $2n$ operations, which is more than $n + k$. To reach $n + k$, we need to remove value 2 in a single operation for at least $n - k$ variables. The only way to do that is through a $X_j \overset{c_3}{\leftarrow} X_i$ for $n - k$ variables X_j. Now, $X_j \overset{c_3}{\leftarrow} X_i$ removes value 2 from $D(X_j)$ only if $D(X_i) = \{0\}$ and $c_3(X_i, X_j) \in C$. $D(X_i)$ is equal to $\{0\}$ only if X_i is one of the k variables on which $\langle X'_i \overset{c_1}{\leftarrow} X''_i, X_i \overset{c_2}{\leftarrow} X'_i \rangle$ has been executed. $c_3(X_i, X_j)$ belongs to C only if $\{i, j\} \in E$. As a result, the set of k vertices i corresponding to the k variables with $D(X_i) = \{0\}$ is a dominating set. ◀

5 Conclusion

We have investigated the complexity of finding a shortest proof of inconsistency of a binary CSP in the form of a sequence of arc consistency operations. Our characterisation in terms of structure or domain size shows that this problem is polynomial when variables have degree two or domains are Boolean. The problem is NP-hard if the CSP has four variables of degree three or if the domain size is bounded by three. It is also NP-hard on trees. In addition, the problem is not FPT-approximable unless the Gap-ETH is false. Although our initial motivation was to provide short explanations for human users, there are other possible applications. Virtual Arc Consistency (VAC) algorithms for cost-function networks use arc-inconsistency explanations in the CSP of zero-cost tuples in order to update cost functions [5]. Our NP-hardness results can be seen as a justification for the use of minimal rather than minimum-cardinality arc-inconsistency explanations by VAC algorithms. On a final positive note, the polynomial-time algorithm for the special case of size-2 domains may prove an inspiration for heuristic methods to improve minimal arc inconsistency explanations via the search for shortest paths in the causal graph described in the proof of Theorem 11.

References

1 Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic CSPs application to configuration. *Artif. Intell.*, 135(1-2):199–234, 2002.

doi:10.1016/S0004-3702(01)00162-X.

2 Bart Bogaerts, Emilio Gamba, and Tias Guns. A framework for step-wise explaining how to solve constraint satisfaction problems. *Artif. Intell.*, 300:103550, 2021. doi:10.1016/j.artint.2021.103550.

3 Parinya Chalermsook, Marek Cygan, Guy Kortsarz, Bundit Laekhanukit, Pasin Manurangsi, Danupon Nanongkai, and Luca Trevisan. From Gap-ETH to FPT-inapproximability: Clique, dominating set, and more. In Chris Umans, editor, *Proceedings of the 58th IEEE Annual Symposium on Foundations of Computer Science (FOCS'17)*, pages 743–754. IEEE Computer Society, 2017. doi:10.1109/FOCS.2017.74.

4 Yijia Chen, Martin Grohe, and Magdalena Grüber. On parameterized approximability. In Hans L. Bodlaender and Michael A. Langston, editors, *Parameterized and Exact Computation, Second International Workshop, IWPEC*, volume 4169 of *Lecture Notes in Computer Science*, pages 109–120. Springer, 2006. doi:10.1007/11847250_10.

5 Martin C. Cooper, Simon de Givry, Martí Sánchez-Fibla, Thomas Schiex, and Matthias Zytnicki. Virtual arc consistency for weighted CSP. In Dieter Fox and Carla P. Gomes, editors, *AAAI 2008*, pages 253–258. AAAI Press, 2008. URL: http://www.aaai.org/Library/AAAI/2008/aaai08-040.php.

6 Irit Dinur. Mildly exponential reduction from gap 3SAT to polynomial-gap label-cover. *Electronic Colloquium on Computational Complexity*, page 128, 2016.

7 M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, 1979.

8 Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *ISAIM*, 2008.

9 E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 886–891, 2003. doi:10.1109/DATE.2003.1253718.

10 Alexey Ignatiev, Nina Narodytska, and Joao Marques-Silva. Abduction-based explanations for machine learning models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):1511–1519, 2019. doi:10.1609/aaai.v33i01.33011511.

11 Ulrich Junker. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. 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*, pages 167–172. AAAI Press / The MIT Press, 2004.

12 Ulrich Junker. Configuration. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 837–873. Elsevier, 2006.

13 R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

14 Pasin Manurangsi and Prasad Raghavendra. A birthday repetition theorem and complexity of approximating dense CSPs. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *Proceedings of the 44th International Colloquium on Automata, Languages, and Programming (ICALP'17)*, volume 80 of *LIPIcs*, pages 78:1–78:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.78.

15 Dániel Marx. Completely inapproximable monotone and antimonotone parameterized problems. In *Proceedings of the 25th IEEE Annual Conference on Computational Complexity*, pages 181–187, 2010. doi:10.1109/CCC.2010.25.

16 Andy Shih, Arthur Choi, and Adnan Darwiche. A symbolic approach to explaining Bayesian network classifiers. In *IJCAI'18*, pages 5103–5111. AAAI Press, 2018.

17 Mohammed H. Sqalli and Eugene C. Freuder. Inference-based constraint satisfaction supports explanation. In William J. Clancey and Daniel S. Weld, editors, *AAAI 96, IAAI 96, Volume 1*, pages 318–325. AAAI Press / The MIT Press, 1996.

A Constraint Programming Approach to Ship Refit Project Scheduling

Raphaël Boudreault
Thales Digital Solutions, Québec, Canada

Vanessa Simard
NQB.ai, Québec, Canada

Daniel Lafond
Thales Digital Solutions, Québec, Canada

Claude-Guy Quimper
Université Laval, Québec, Canada

Abstract

Ship refit projects require ongoing plan management to adapt to arising work and disruptions. Planners must sequence work activities in the best order possible to complete the project in the shortest time or within a defined period while minimizing overtime costs. Activity scheduling must consider milestones, resource availability constraints, and precedence relations. We propose a constraint programming approach for detailed ship refit planning at two granularity levels, daily and hourly schedule. The problem was modeled using the CUMULATIVE global constraint, and the *Solution-Based Phase Saving* heuristic was used to speedup the search, thus achieving the industrialization goals. Based on the evaluation of seven realistic instances over three objectives, the heuristic strategy proved to be significantly faster to find better solutions than using a baseline search strategy. The method was integrated into *Refit Optimizer*, a cloud-based prototype solution that can import projects from Primavera P6 and optimize plans.

2012 ACM Subject Classification Computing methodologies → Planning and scheduling; Theory of computation → Constraint and logic programming

Keywords and phrases Ship refit, planning, project management, constraint programming, scheduling, optimization, resource-constrained project scheduling problem

Digital Object Identifier 10.4230/LIPIcs.CP.2022.10

Supplementary Material *Software (Source Code)*: `https://github.com/raphaelboudreault/chuffed/releases/tag/SBPS`; archived at `swh:1:dir:3eb166e6188719513b62f762559a2420c13c9997`

Funding This project has received financial support from the Scale AI Canadian Innovation Supercluster and from the Mitacs Accelerate program.

Acknowledgements Thanks are due to the many members of the Refit Optimizer project team and our collaborators at Dalhousie University, Polytechnique Montréal, Sōdan, Simwell and Genoa Design International. We are very grateful to the many domain experts consulted and to Seaspan Victoria Shipyards for their invaluable feedback.

1 Introduction

Ship refit planning is a complex and tedious endeavor that requires scheduling several hundred (or thousand) tasks across a time horizon that may span several weeks, months or even over a year [2]. Planners must ensure that precedence relations between tasks are respected, that the required human and material resources are available, and that the scheduled work is completed within the maximum allocated project duration. For instance, dry-dock work periods need to be fixed years in advance, thus leaving no flexibility for increasing project

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 10; pp. 10:1–10:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

time span. Potential goals of planners in this context are to create schedules minimizing the project total duration or, in case the planning horizon prevents the work to be accomplished in time, minimizing the overtime labor costs. Some planners focus on other needs, such as creating robust schedules leaving flexibility to adjust to unforeseen delays. Indeed, while initial plans must be free of conflicts, unplanned events, delays and their arising work require ongoing re-planning efforts throughout the project. While initial planning may take several weeks for large projects, leaving time for planners to manually attempt optimizing the task scheduling, replanning leaves very little time for planners to consider their options and thus is mainly an opportunistic and reactive process.

There exist multiple enterprise resource planning tools such as Microsoft Project [25], Primavera P6 [33], and IBM Maximo [18], which are typical software solutions to support ships refit planning. Yet beyond the core project management functionalities, the support for optimizing schedules with computational methods from operations research remains limited to resources leveling, i.e. spreading the workload more evenly across the project duration. Some optimization solutions have been previously created for custom projects, yet lack reusability. To our knowledge, the only generic and reusable schedule optimization capability currently available is the Aurora (Stottler Henke) intelligent scheduling solution [35]. While the available information about Aurora's proprietary optimization algorithms is limited, these are described as being based on heuristics, as opposed to exact methods, derived from domain experts. While this satisfying approach is highly relevant and effective for human problem solving given the human brain's bounded computational capabilities, we posit that optimization algorithms can be developed to do better than human-derived heuristics. The current work aims to push further the state-of-the-art in this area by producing a general purpose exact method enhanced with metaheuristics.

Thales Canada has set out to create *Refit Optimizer*, a prototype solution for multi-objective optimization in the ship refit domain, while also designing it to be reusable across a wide variety of other scheduling contexts [22]. The key motivation for this work comes from challenges and innovation opportunities identified in the context of the Arctic/Offshore Patrol Ships and Joint Support Ships in-service support (AJISS) program with the Royal Canadian Navy. Herein, the focus is on detailed planning, using either days or hours as the basic time unit. The Refit Optimizer prototype is currently operational, deployed on a secure cloud platform, and combines several complementary services for importing/exporting project data (from/to Primavera P6), visualizing the schedule using an interactive Gantt chart and tasks list, editing the schedule, freezing scheduled tasks, and optimizing the schedule according to one of three different objectives (*makespan*, *overtime costs* or *robustness*). Additional capabilities include comparing options, analyzing and visualizing geospatial conflicts, forecasting progress, modeling uncertainties and assessing risks using discrete-event simulations (see [22]).

We propose a constraint programming approach for detailed ship refit planning that is currently fully integrated into Refit Optimizer. The problem considered is closely related to the *Resource-Constrained Project Scheduling Problem* (RCPSP) [16, 17, 34] and is modeled using the efficient CUMULATIVE global constraint [1, 38]. We use the *Solution-Based Phase Saving* (SBPS) value selection heuristic [13, 42] to speedup the search and obtain better solutions in a reasonable time. We evaluate our approach on a benchmark formed of seven instances supplied by our industrial partners, namely Sōdan, the AJISS team and Seaspan Victoria Shipyards, which are compared to each other using RCPSP complexity metrics from the literature.

The paper is organized as follows. Section 2 describes the ship refit planning problem. Section 3 presents background notions on scheduling, constraint programming, SBPS and RCPSP complexity measures. The CP model is presented in Section 4, as well as its extensions and the search heuristics developed in our context. Our approach is evaluated on the benchmark instances in Section 5. Finally, Section 6 addresses the applicability of the solution to an industrial setting, followed by a conclusion suggesting directions for future work.

2 Problem Description

Ship refitting is an important shipyard event during which all ship's activities are suspended for improvements. The objective of a refit is to restore, customize, modify, or modernize part of a ship. Most of the time, however, stopping all activities can become costly, which makes efficient ships refit planning important. The time window, or *horizon*, during which a refit takes place can be decided years in advance. When the horizon is exceeded, the dock is no longer available and the ship has to leave. Thus, in order to estimate the required duration, planners have to consider a large number of daily or even hourly tasks depending on multiple capacity-limited resources, both human and material. Furthermore, due to physical limitations, a maximum number of workers must be simultaneously allowed in some work areas. Precedence relationships must be considered between many of the tasks, while some date constraints, such as milestones, must be achieved. Finally, specific tasks must be idle over the weekends. In practice, the initial planned time is often insufficient, in which case overtime for some tasks can be scheduled to fit in the restrictive horizon.

Three main objectives are targeted in this project according to the challenges faced by shipyards. First, to support planners in their horizon estimation, the prototype solution has to offer to minimize the refit total duration, also known in scheduling as **makespan**. This helps the planner at the tactical level by identifying the minimum time needed for a certain ship refit, according to the constraints on tasks. Then, it has to allow the user to produce an operational plan over a fixed horizon that minimizes the **overtime** costs. Since a lot of unplanned delays happen during the actual refit execution, this option helps a shipyard respect their obligation while minimizing costs associated with overtime labor. Finally, the solution has to propose an operational plan taking the **robustness** into account when planning overtime. The idea is to minimize the risk of exceeding the refit deadline by planning the overtime work, as much as possible, at the beginning of the horizon. Thus, if unforeseen events during the execution increase the need for overtime before the end of the refit, it is still possible to proceed. With the option of these three objectives, a user can efficiently estimate a ship refit duration, while being supported during its execution.

Our research is based on the practical needs of our industrial partners which supplied us with realistic use cases. Table 1 presents seven instances of different sizes that were made available for our tests. Each instance is defined by a horizon, given in days or in hours depending on the planning granularity, which represents the available time to complete all tasks. The proposed horizon comes from the initial planners overestimate and can be seen as a baseline for the makespan minimization. The number of tasks, precedence relations, and resources, as well as the task duration range (without overtime), help to better evaluate and compare the size of the instances. The number of tasks that can be performed in overtime (**#O**) is given, where a "*" indicates that those tasks must be idle during weekends. Some tasks do not follow any work hours requirement and thus cannot be shortened nor suspended. A common example of this in a ship refit is paint drying tasks. The number of work areas (**#WA**) is also presented and included as a specific type of resource.

Table 1 Size comparison of the seven instances supplied by our industrial partners.

Instance	Horizon	#Tasks (#O)	Task duration	#Precedence relations	#Resources (#WA)
day-yacht21	29 days	21 (20)	1-3 days	32	9 (2)
hour-yacht21	704 hours	21 (20)	1-8 hours	32	9 (2)
generic136	178 days	136 (136*)	1-20 days	99	9 (4)
software138	183 days	138 (138*)	1-10 days	341	8 (0)
navy253	728 hours	253 (253*)	1-8 hours	246	92 (87)
cruise510	268 days	510 (464*)	1-15 days	550	32 (24)
navy830	6200 hours	830 (830*)	1-200 hours	816	146 (128)

The first four instances, `day-yacht21`, `hour-yacht21`, `generic136`, and `software138`, were artificially created by the team for testing purposes. They are used to test the limits of the optimization algorithms, since realistic instances are somewhat simpler because they are manually created by human experts. The instances `day-yacht21` and `hour-yacht21` are two versions of the same ship with different task durations and planning granularities, respectively days and hours. This simple problem has been used in user workshops to compare the result of a manual optimization to the result of our approach. Instance `generic136` does not describe a particular ship refit and was created to test simultaneously various precedence and date constraint types. Instance `software138` describes the management of a software development project, which is used to show the genericity of our approach to scheduling problems with resources. Other instances are anonymized versions of real, or closely inspired by real, refit use cases from recent years. Instances `navy253` and `navy830` are two versions of a real use case provided by the AJISS team, while `cruise510` is inspired by a sample problem provided by Seaspan Victoria Shipyards.

3 Background

3.1 Scheduling

The problem we consider is part of the great family of *scheduling problems*. In the operations research and optimization literature, scheduling problems are many and varied. Given a set of tasks $\mathcal{I}$, these problems require finding when to execute each task over a definite timeline $\mathcal{T} := \{0, 1, \ldots, t_m\}$, where each $t \in \mathcal{T}$ is a discrete time point, so that an objective function is optimized while different constraints are satisfied. Specifically, our scheduling use case is highly related to the *Resource-Constrained Project Scheduling Problem* (RCPSP). Introduced in 1969 by Pritsker et al. [34], its standard definition (see e.g. [16, 17]) supposes first that *preemption* is forbidden, i.e. that each task cannot be interrupted once started. Then, a finite set of resources $\mathcal{R}$ is considered, where each task $i \in \mathcal{I}$ requires an amount $h_{i,r} \in \mathbb{Z}^{\geq 0}$ of resource $r \in \mathcal{R}$ used for its whole duration. Each resource $r \in \mathcal{R}$ has a constant usage capacity $c_r \in \mathbb{Z}^{>0}$ and is fully available at any time (*renewable*). Also, the resources are *cumulative*, i.e. more than one task can use a resource at a time. Thus, the RCPSP assumes at each time point $t \in \mathcal{T}$ that each resource's total usage by tasks does not exceed its capacity. Finally, precedence relationships between some tasks are considered. The objective is to find a schedule with the earliest project ending date or, in other words, the minimal makespan.

Blazewicz et al. [7] have shown that the RCPSP belongs to the strongly NP-hard problems. Thus, its computation complexity and industrial application interest has led to a plethora of techniques, both exact and heuristic, in various research domains. These approaches include notably specialized *branch-and-bound methods* [12, 21, 40], *mixed-integer programming* [11, 20] and, as presented in Section 3.2, *constraint programming*. We refer the reader to Pellerin et al. [30] for a recent survey of current heuristic approaches.

Among the various RCPSP benchmark instance sets in the literature, three are usual: PSPLIB [19], BL [5] and Pack [8]. While these contain from 17 to 120 tasks, our instances listed in Table 1 are significantly larger. Furthermore, the number of resources in these benchmarks is at most 5, while ours can go up to 146. Planning horizons, as well as `navy830` maximal task duration, are also greater than the ones in these benchmarks that go up to 139 and 19 time points respectively, but are comparable to the ones in the more realistic instances of Koné et al. [20].

3.2 Constraint Programming

Constraint Programming (CP) is a powerful programming paradigm to solve combinatorial problems. In particular, it can be used to optimally solve many large-scale optimization problems under constraints [36]. A CP model is formed of *decision variables*, each provided with a finite set of possible values called *domain* and denoted $\text{dom}(X)$ for a variable X. The relationships between the variables are defined by *constraints*, each provided with a specialized inference algorithm. Optimization problems also have an *objective function* to minimize. CP solvers generally perform a tree search to find feasible solutions, where each node of the tree corresponds to a partial solution, and each *branching* is a node created from its parent with an additional assignment of an unfixed variable to a value. The branching selection rules for variables and values are defined using *heuristics*.

Significant efforts have been made in the constraint programming community to efficiently solve scheduling problems involving resource constraints. Introduced by Aggoun and Beldiceanu [1], the Cumulative global constraint enforces the usage of a resource by tasks to be at most its capacity for each time point in the optimization timeline. Formally, for a resource $r \in \mathcal{R}$, given variables S_i and D_i are respectively the starting time and duration of task $i \in \mathcal{I}$, $\textsc{Cumulative}([S_i \mid i \in \mathcal{I}], [D_i \mid i \in \mathcal{I}], [h_{i,r} \mid i \in \mathcal{I}], c_r)$ is logically equivalent to

$$\sum_{\substack{i \in \mathcal{I}: \\ S_i \leq t < S_i + D_i}} h_{i,r} \leq c_r \qquad \forall t \in \mathcal{T}.$$

Over the years, many efficient rules for the Cumulative constraint have been developed to detect failures and filter variables' domains (see e.g. [4, 6, 14, 29, 41]). Furthermore, important progress towards solving large-scale RCPSP instances with CP has been made by Schutt et al. [37, 38] by combining some of these rules with *lazy clause generation*. Introduced by Ohrimenko et al. [28], this technique is a hybrid between CP and *boolean satisfiability* (SAT) solvers. During the search, each filtered value is now recorded with an explanation as a SAT clause. When a failure is detected, the solver uses its explanations to learn a *nogood*, a core reason for what led to this conflict. This nogood is then added as a new constraint to the solver's underlying SAT mechanism. As a result, this process allows avoiding reproducing the same choices later during the search. Furthermore, it enables SAT-based branching heuristics depending on variables' activity in conflicts, notably *Variable State Independent Decaying Sum* (VSIDS) [26]. Several modern and efficient CP solvers, such as Chuffed [9] and OR-Tools [31], are based on the lazy clause generation technique.

3.3 Solution-Based Phase Saving

Large Neighborhood Search (LNS) [32, 39] is a metaheuristic that has been successfully used in many contexts for scaling exact solving methods to large optimization problems. Given an initial solution, the technique iteratively improves the best known solution according to the considered objective. At each iteration, a *neighborhood* is chosen such as a part of the variables are fixed to their value in the current solution, whereas the others are relaxed, which generates a smaller subproblem. Solutions of the latter can then be quickly found by any chosen method.

One of the major drawbacks of relying on LNS to find better solutions for an optimization problem is the loss of exactness from the initial solving method. Thus, a relatively simple, efficient and closely related to LNS value selection heuristic for CP solvers has been introduced by Vion and Piechowiak [42] as *Best-Solution.* Demirović et al. [13] introduced the same heuristic soon after under the name *Solution-Based Phase Saving* (SBPS). We refer in the following to this heuristic as the latter terminology.

Given an optimization problem with variables $X_1, \dots, X_n$, a variable selection heuristic H_{var} and a value selection heuristic H_{val}, let $b = (b_1, \dots, b_n)$ denote the current best solution, if one exists, where b_i corresponds to the value of X_i in this solution. If X_k is the variable chosen by H_{var}, the SBPS branching strategy does the following:

a) If b exists and $b_k \in \text{dom}(X_k)$, then choose the value b_k for X_k;

b) Else, choose a value for X_k following H_{val}.

Thus, the strategy focuses the search around the current best solution as much as possible. Combined with a restart strategy and a dynamic variable selection heuristic such as VSIDS, SBPS effectively mimics LNS [13]. Indeed, starting from the root node, the search fixes almost all variables to their current best solution value, and then searches around this solution for a subset of unassigned variables with backtracking, thus implicitly building a neighborhood. The size of the latter is then limited by the restart strategy. Also, the dynamic aspect of the search produces a built-in diversification of neighborhoods, besides that VSIDS tends to select closely related variables. The resulting strategy produced interesting results on a variety of instances [13, 42].

3.4 RCPSP Complexity

In order to properly assess the performance of our approach on the instances in Table 1, it is important to compare them on a similar scale. To do so, Artigues et al. [3] listed a selection of state-of-the-art indicators that characterize the complexity of RCPSP instances. These indicators are typically used to generate instances of a targeted complexity level, but are also relevant to evaluate existing instances. They can be classified in four categories: precedence-oriented, time-oriented, resource-oriented, and hybrid.

The *Order Strength* (OS) is a precedence-oriented indicator showing how much the instance's precedence constraints induce an ordering of the tasks [3, 24]. If P denotes the set of task pairs $\{i, j\}$ $(i, j \in \mathcal{I},\ i \neq j)$ which cannot be executed in parallel due to a chain of precedence constraints between them, OS is defined as the ratio of $|P|$ over the total number of task pairs:

$$\text{OS} := \frac{|P|}{|\mathcal{I}|(|\mathcal{I}| - 1)/2}.$$

We have $\text{OS} \in [0, 1]$. It has been observed that the closer the value is to 1, the more ordered the tasks and the lower the complexity.

The *Resource Factor* (RF) is a resource-oriented indicator which evaluates the resource usage by tasks [3]. It is defined as the ratio of the average number of required resources by task over the number of resources:

$$\text{RF} := \frac{\sum_{i \in \mathcal{I}, r \in \mathcal{R}} u_{i,r}}{|\mathcal{I}||\mathcal{R}|}.$$

where $u_{i,r}$ equals 1 if task $i \in \mathcal{I}$ requires resource $r \in \mathcal{R}$ ($h_{i,r} > 0$), and 0 otherwise. We have RF $\in [0, 1]$. It has been shown that as the RF value increases, the complexity also does.

The *Resource Strength* (RS) indicator combines a time-oriented view with the resource complexity [3, 10]. For each resource $r \in \mathcal{R}$, it considers its maximal usage $c_r^{\max}$ when tasks are scheduled at their earliest while satisfying precedence constraints,

$$c_r^{\max} := \max_{t \in \mathcal{T}} \sum_{i \in \mathcal{I}_t^{\text{ES}}} h_{i,r},$$

where $\mathcal{I}_t^{\text{ES}} \subseteq \mathcal{I}$ is the subset of tasks executed at time point t in this schedule. The resource's strength RS_r is then defined by the ratio of its overall availability over its availability in the earliest schedule:

$$RS_r := \frac{c_r - c_r^{\min}}{c_r^{\max} - c_r^{\min}},$$

where $c_r^{\min} := \max_{i \in \mathcal{I}} h_{i,r}$. In the case where $c_r^{\max} \leq c_r$, RS_r is instead fixed to 1. Thus, every resource $r \in \mathcal{R}$ with $RS_r = 1$ is always sufficiently available and is not a constraint. The RS value is obtained by averaging RS_r over all resources. We have RS $\in [0, 1]$, and the complexity generally increases as RS decreases.

The *Disjunctive Ratio* (DR) indicator is a hybrid between the precedence and resource complexities [3, 5]. The set of task pairs P which cannot be executed in parallel from OS is extended with a set D of pairs that would violate a resource constraint if both tasks overlapped in time, i.e. $D := \{\{i, j\} : \exists\, r \in \mathcal{R}, h_{i,r} + h_{j,r} > c_r\}$. DR is then defined as the ratio of the number of elements in this new set over the total number of task pairs:

$$\text{DR} := \frac{|P \cup D|}{|\mathcal{I}|(|\mathcal{I}| - 1)/2}.$$

We also have DR $\in [0, 1]$. It has been established that the higher DR is, the more disjunctive the instance is.

4 Methodology

4.1 Main model

In the following, we present the main CP model developed for the ship refit planning problem, as described in Section 2. It is inspired from the classical RCPSP model for CP [38]. Note that in this model, we assume a planning granularity in days. Extensions to support hours and other specific constraints are discussed in Section 4.2.

Reusing the scheduling notation introduced in Section 3.1, we define the input parameters. The horizon $t_m \in \mathbb{Z}^{>0}$ determines the scheduling timeline as the set $\mathcal{T} = \{0, 1, \ldots, t_m\}$. If $\mathcal{I}$ is the given set of tasks to schedule, each task $i \in \mathcal{I}$ is associated with s_i^L, $s_i^U \in \mathcal{T}$ and e_i^L, $e_i^U \in \mathcal{T}$ which are lower (L) and upper (U) bounds on the task starting (s) and ending (e) times as implied by the date constraints. Each task $i \in \mathcal{I}$ also has a processing time

$p_i \in \mathcal{T}$ which corresponds to the task duration in days without overtime. The precedence requirements form the set $\mathcal{P}$ and are encoded as triples (i, j, l), asking for task $i \in \mathcal{I}$ to be completed $l \in \mathcal{T}$ days before task $j \in \mathcal{I} \setminus \{i\}$ starts. Each resource r in the given set of resources $\mathcal{R}$ has a constant capacity $c_r \in \mathbb{Z}^{>0}$, a daily standard usage cost $w_r^S \in \mathbb{R}^{\geq 0}$, and a daily overtime usage cost $w_r^O \in \mathbb{R}^{\geq 0}$, with $w_r^S \leq w_r^O$. The amount of resource $r \in \mathcal{R}$ required by task $i \in \mathcal{I}$ is given by $h_{i,r} \in \mathbb{Z}^{\geq 0}$. A working day is defined by three parameters, $d^S, d^O, d^E \in \{0, 1, \ldots, 23\}$, with $d^S < d^O \leq d^E$, where $[d^S, d^O - 1]$ are standard hours and $[d^O, d^E]$ are overtime hours. Finally, tasks that can be performed in overtime are contained in the set $\mathcal{I}^* \subseteq \mathcal{I}$.

The integer decision variables of our model are as follows. For each task $i \in \mathcal{I}$, S_i is the task starting time, while E_i is the task total elapsed time between its start and its completion. We define $\operatorname{dom}(S_i) = [s_i^L, s_i^U]$ and $\operatorname{dom}(E_i) = \mathcal{T}$, for each $i \in \mathcal{I}$.

The constraints of the model are presented below.

$$\textsc{Cumulative}([S_i \mid i \in \mathcal{I}], [E_i \mid i \in \mathcal{I}], [h_{i,r} \mid i \in \mathcal{I}], c_r) \qquad \forall r \in \mathcal{R} \tag{1}$$

$$e_i^L \leq S_i + E_i \leq e_i^U \qquad \forall i \in \mathcal{I} \tag{2}$$

$$S_i + E_i + l \leq S_j \qquad \forall (i, j, l) \in \mathcal{P} \tag{3}$$

$$\left\lceil \frac{\left(d^O - d^S\right) p_i}{d^E - d^S} \right\rceil \leq E_i \leq p_i \qquad \forall i \in \mathcal{I}^* \tag{4}$$

$$E_i = p_i \qquad \forall i \in \mathcal{I} \setminus \mathcal{I}^* \tag{5}$$

The Cumulative constraints (1) ensure that the cumulative usage of each resource by tasks does not overload its capacity at any given point in the timeline. Since $S_i + E_i$ represents the ending time of task $i \in \mathcal{I}$, constraints (2) force the ending time of each task to respect its upper and lower bounds, as implied by the problem date constraints. Constraints (3) impose the precedence requirements from set $\mathcal{P}$. Constraints (4) define the possible values for the elapsed time E_i of task $i \in \mathcal{I}^*$ that can include some overtime work hours. First, the value must be at most p_i since it corresponds to the task duration without overtime. Second, note that performing overtime hours reduces the overall elapsed time in days of a task. The number of standard hours required by task $i \in \mathcal{I}^*$ is given by $\left(d^O - d^S\right) p_i$, which is redistributed over longer days of $d^E - d^S$ hours, fully using the overtime hours each day, thus a lower bound on E_i. For example, if $(d^S, d^O, d^E) = (8, 16, 20)$, a typical day is formed of 8 standard hours and 4 overtime hours. A task $i \in \mathcal{I}^*$ with $p_i = 3$ requires $3 \times 8 = 24$ standard hours, but can be completed in $\lceil \frac{24}{12} \rceil = 2$ days when working 12-hour shifts (8 hours is in overtime). Finally, constraints (5) force tasks that cannot be executed in overtime $(\mathcal{I} \setminus \mathcal{I}^*)$ to have their standard duration.

We considered three different objective functions, where the model can be used with either of them. First, the **makespan** objective, which is the minimization of the schedule duration, is modeled as follows:

$$\min \max_{i \in \mathcal{I}} \left(S_i + E_i\right).$$

This objective is considered without allowing overtime, which is done by assuming $\mathcal{I}^* := \emptyset$.

Then, the **overtime** objective is to minimize the costs associated with overtime work. For a task $i \in \mathcal{I}^*$, the number of standard working days transformed into overtime is given by $p_i - E_i$. Since each overtime day costs $w_r^O - w_r^S$ per resource $r \in \mathcal{R}$ used, the total cost is given by $\sum_{i \in \mathcal{I}^*} \sum_{r \in \mathcal{R}} h_{i,r}(w_r^O - w_r^S)(p_i - E_i)$. Equivalently, we minimize the following linear expression where the inner summation is pre-computed for each $i \in \mathcal{I}^*$:

$$\min \sum_{i \in \mathcal{I}^*} (p_i - E_i) \left(\sum_{r \in \mathcal{R}} h_{i,r}(w_r^O - w_r^S) \right).$$

Finally, the **robustness** objective is to minimize the risk of exceeding the deadline of a schedule by planning the overtime early in the project. To evaluate this criterion for a task $i \in \mathcal{I}^*$, we multiply its amount of used overtime $p_i - E_i$ by its starting time S_i, leading to the following non-linear function:

$$\min \sum_{i \in \mathcal{I}^*} (p_i - E_i) S_i.$$

4.2 Extensions

The model presented in Section 4.1 has been extended in several ways to better suit the ship refit planning reality. First of all, more types of precedence constraints were considered other than the ones in (3). Indeed, our current model supports any precedence of the form $X_i + l \leq Y_j$, where $X, Y \in \{S, S + E\}$ for $l \in \mathcal{T}$ and $i, j \in \mathcal{I}$ $(i \neq j)$. It was also asked that our model consider that some tasks, but not all, should be suspended on weekends. To this end, additional variables N_i representing the non-working (idle) time points of task $i \in \mathcal{I}$ were considered. In the model, the non-working time is included in the elapsed time of the task. Thus, constraints (4) and (5) are instead applied on the working time $E_i - N_i$. Additional constraints are considered to enforce a value for N_i when the task overlaps at least one weekend. Finally, the model has been extended to support a scheduling granularity in hours. In this case, the overtime constraints (4) and (5) are replaced by additional variables O_i encoding the number of hours in overtime for task $i \in \mathcal{I}^*$. Each of these variables is closely related via special constraints to its associated N_i which also includes the non-working hours during the nights. Thus, in hours, the elapsed times are simply equal to $p_i + N_i$.

4.3 Search Heuristics

Two branching heuristics for the CP model are considered herein as a basis. For the **makespan** objective, we select the starting time variable S_i, $i \in \mathcal{I}$, with the smallest value in its domain, and we assign it to this value. This way, the search focuses as much as possible around the schedule where each task begins at its earliest starting time. For the **overtime** and **robustness** objectives, the branching heuristic selects the task $i \in \mathcal{I}$ that has a starting time variable S_i with the smallest value in its domain. Then, it assigns, in order, the smallest value in $\text{dom}(S_i)$ to S_i and the greatest value in $\text{dom}(E_i)$ to E_i. In the hour granularity case, this last branching is replaced by assigning the smallest value in $\text{dom}(O_i)$ to O_i, and then the smallest value in $\text{dom}(E_i)$ to E_i. In both cases, the intuition is to place the tasks as early as possible, while simultaneously choosing a task duration with as few overtime time points as possible. The resulting heuristic can be formulated with the *priority search* annotation from the MiniZinc modeling language [15, 27], and is supported by Chuffed CP solver [9].

5 Experimentation

The benchmark we considered for our experiments is formed of the seven instances presented in Section 2. In order to compare them on the same basis and assess their complexity, we computed for each instance the indicators OS, RF, RS and DR introduced in Section 3.4. The resulting values are presented in Table 2. In the case of RS, since many resources gave a value $RS_r = 1$ which makes the comparison difficult, we decided to compute RS by averaging RS_r over all resources $r \in \mathcal{R}$ that are in fact restrictive $(RS_r < 1)$. Values in bold font highlight the most complex instances according to each indicator.

Table 2 Complexity indicators of the seven instances.

Instance	OS	RF	RS	DR
`day-yacht21`	0.72	**0.28**	0.40	**0.82**
`hour-yacht21`	0.72	**0.28**	0.40	**0.82**
`generic136`	**0.02**	0.23	0.24	0.06
`software138`	0.27	0.12	**0.01**	0.32
`navy253`	0.19	0.02	0.71	0.18
`cruise510`	0.07	0.06	0.27	0.07
`navy830`	**0.02**	0.01	0.66	0.02

Each instance has its strengths and weaknesses. Looking first at OS, instance `generic136` is one with the least ordered structure. It can be explained by the fact that the instance was created with arbitrary precedence relations as a means of testing. It seems nonetheless that our three realistic use cases, `navy253`, `cruise510` and `navy830`, are also relatively complex according to this indicator. In terms of resource usage (RF) and disjunctive structure (DR), the two smallest instances, `day-yacht21` and `hour-yacht21` are the most complex ones. A big part of that comes from the fact these instances contain a lot less resources, but are more often used at full capacity. Finally, considering the time-oriented view (RS), the typical RCPSP instance `software138` is the most constrained in terms of resources. Although the realistic instances seem to be less affected by their resource constraints, their complexity lies in the large number of tasks to schedule and the lack of artificially induced ordering.

The CP model presented in Section 4 was modeled in the MiniZinc 2.5.5 language [27]. We implemented[1] the SBPS scheme as described in Section 3.3 into the solver Chuffed [9], which we used to solve the instances. The CUMULATIVE constraint was set to apply the optional Time-Table-Edge-Finding (TTEF) checking and filtering rules [37, 41]. The experiments were performed on a MSI GP63 Leopard 8RE machine with an Intel i7-8750H CPU at 2.2 GHz, 6 cores and 16 GB of RAM. Each optimization execution was given a timeout of 4 hours, and a constant restart strategy of 100 failures.

Each instance was solved for each objective, **makespan**, **overtime** and **robustness**. For the last two objectives, we have empirically chosen a "restricted" horizon for each instance corresponding to a reduction between 2% and 30% of the best known makespan. However, due to its specific constrained nature preventing overtime to be performed, `generic136` could not be considered for these objectives.

For each optimization, two search methods were compared. The *BASELINE* strategy consists simply of using the search heuristics defined in Section 4.3. The *SBPS* strategy, on the other hand, also uses these heuristics until a first solution is found. When it is the case, the SBPS branching procedure activates. Furthermore, the variable selection scheme of choosing S_i, $i \in \mathcal{I}$, with the smallest value in $\text{dom}(S_i)$ is replaced by selecting S_i with the greatest conflict activity, as provided by the VSIDS score of Chuffed [9, 26]. The latter modification allows the resulting procedure to effectively reproduce an LNS [13]. We did not directly use the *free search* (`-f`) option of Chuffed, which is alternating between the user-defined heuristic and the VSIDS strategy (on all the variables), since we observed the solving process was generally slowed down by its usage. However, since no solution was found before the timeout without it, we added the free search for instance `software138` when optimizing the **overtime** and the **robustness**.

[1] The code is available at `https://github.com/raphaelboudreault/chuffed/releases/tag/SBPS`. We thank Emir Demirović for providing his original implementation [13].

Table 3 Results on the benchmark instances when considering the **makespan** objective.

Instance	*Baseline*		*SBPS*		Time (s) improv.
	Objective	**Time (s)**	**Objective**	**Time (s)**	
`day-yacht21`	**28 days**	0.2*	**28 days**	0.2*	0.2
`hour-yacht21`	**78 hours**	0.4*	**78 hours**	0.4*	0.4
`generic136`	**178 days**	0.7*	**178 days**	0.7*	0.7
`software138`	144 days	1.4	**119 days**	41.6	1.1
`navy253`	**389 hours**	4.2	**389 hours**	3.7	3.7
`cruise510`	228 days	14.7	**227 days**	785.7	229.3
`navy830`	5216 hours	18.7	**5144 hours**	199.7	18.2

Table 4 Results on the benchmark instances when considering the **overtime** objective.

Instance	*Baseline*		*SBPS*		Time (s) improv.
	Objective	**Time (s)**	**Objective**	**Time (s)**	
`day-yacht21`	**1560**	0.3*	**1560**	0.3*	0.3
`hour-yacht21`	**485**	0.4*	**485**	0.4*	0.4
`software138`	5600	14 359.6	**2600**	153.4	34.3
`navy253`	70	4.2	**66**	5.0	4.0
`cruise510`	26 000	11.7	**15 760**	7555.3	5.8
`navy830`	227	25.2	**36**	276.5	26.6

Table 5 Results on the benchmark instances when considering the **robustness** objective.

Instance	*Baseline*		*SBPS*		Time (s) improv.
	Objective	**Time (s)**	**Objective**	**Time (s)**	
`day-yacht21`	**47**	0.3*	**47**	0.3*	0.3
`hour-yacht21`	**192**	0.4*	**192**	0.4*	0.4
`software138`	900	13 571.8	**258**	320.2	15.3
`navy253`	10 686	5057.6	**3480**	1411.9	6.7
`cruise510`	4870	13 022.5	**842**	1321.7	14.2
`navy830`	146 794	11 208.9	**9076**	13 863.4	41.1

Tables 3, 4, and 5 present respectively the results obtained when considering the **makespan**, **overtime** and **robustness** objectives. In each table, we report the best objective value found (**Objective**) as well as the solving time (**Time**) in seconds. When the timeout was reached, we instead show the required time to find the best solution. A "*" next to a solving time value indicates that the instance was optimally solved. For comparison purposes, we also report the time in seconds taken with *SBPS* to find a solution with an objective value less than or equal to the best one found with *Baseline* (**Time improv.**).

For the **makespan** objective (Table 3), instances `day-yacht21`, `hour-yacht21`, and `generic136` are quickly solved optimally using both strategies. For the other instances, the *Baseline* method finds its best solution in the first 20 seconds of the search, without being able to improve it after. In comparison, the *SBPS* strategy improves the minimal makespan for `software138` by 25 days, `cruise510` by 1 day and `navy830` by 9 days, while instance `navy253` gave the same solution. The use of *SBPS* thus reduced the best makespan by 5% on average. Furthermore, the improved solutions are found in a similar time than *Baseline*, except for `cruise510` where the solution of 227 days is found 15.6 times slower.

For the **overtime** objective (Table 4), the objective value corresponds to the costs induced by overtime work. Since our benchmark is formed of abstract and anonymized realistic instances, the obtained costs are of different sizes and units, thus incomparable in-between instances. Note that instances `day-yacht21` and `hour-yacht21` are still trivially solved optimally with both strategies. Bigger instances see their best objective value from *Baseline* considerably improved with *SBPS*. The best cost is reduced by 48% on average using *SBPS*, while the best solution of *Baseline* is found 94 times faster for `software138`, and 2 times faster for `cruise510`.

For the **robustness** objective (Table 5), the unconventional way of representing it is a challenge for the *Baseline* method. While `day-yacht21` and `hour-yacht21` are still optimally solved, the larger instances need a lot of computation time to settle on a good solution. In fact, for `software138` and `cruise510`, the time to find the best solution is close to the timeout (14 400 seconds). In comparison, the *SBPS* strategy finds a solution with the same robustness value much faster, in no more than 42 seconds. For `cruise510`, the solution is found 917 times faster. Furthermore, the best objective found by *Baseline* is reduced on average by 79%.

6 Discussion

The main goal of this research was to create a prototype solution for multi-objective optimization in the ship refit domain. By successfully proposing plans to the targeted instances supplied by our industrial partners within a reasonable time limit, we have demonstrated the applicability of our constraint programming model. It was important for practical use to obtain a good solution under predetermined time limits: 15 minutes for instances under 100 tasks, one hour for instances between 100 and 500 tasks, and four hours for instances over 500 tasks. In comparison, an expert manually planning smaller instances like `day-yacht21` could take up to four hours. While it was possible to consider the *Baseline* strategy as an attempt to solve the biggest instances, the computation time and the solution's quality were sometimes less than satisfactory for industrial purposes. The use of *SBPS* proved to be a fairly good strategy, leading to improved objective values in a significantly shorter amount of time for the **makespan**, **overtime** and **robustness** objectives. These experiments also demonstrated the Refit Optimizer prototype's relevance for real-world project planning and ongoing project management with re-optimization. Qualitative user feedback from usability tests with domain experts also supports this assessment.

A lot of effort was put in designing Refit Optimizer to be reusable across a wide variety of other scheduling contexts. The terminology and architecture of the product database were made consistent with the terms and structures from the project management field. Plus, the genericity of the constraints formulation allows to consider more than ten different types of precedence and time constraints on tasks. Resources can include workers as well as locations and equipment, which opens to future improvements with geospatial constraints [22]. Preliminary tests on real projects in the naval, avionics, and ground transportation domains also show that the solution has a strong cross-domain potential.

The main reason why we chose Chuffed [9] over other CP solvers was its proven efficiency on large-scale RCPSP instances by combining state-of-the-art Cumulative filtering algorithms with lazy clause generation [37, 38]. Its built-in VSIDS branching heuristic allowed us to easily reproduce the gains obtained in recent SBPS-related work [13, 42]. Furthermore, Chuffed could directly support the *priority search* MiniZinc annotation [15, 27] used to formulate our baseline search heuristics. We did try the OR-Tools CP-SAT solver [31] via its FlatZinc implementation, but preliminary results showed greater computation times to find similar or worse solutions. We did also try a standard LNS procedure prior implementing SBPS. However, we rapidly found that the technique was rather inefficient for the **overtime** and **robustness** objectives, while it was difficult to find a suitable solution deconstruction rule.

There were many challenges in working in an industrial setting. First, the importance of anonymity for the industrial partners made it difficult to analyze some results. For many instances, the estimated workforce costs were changed to abstract values, which produced unrealistic execution costs. Having access to real data would have allowed us to produce more complex realistic instances to challenge our prototype. Explainability of results was also an important challenge. Since the tool needs to be used in an industrial setting by many different people, it is important to document and explain each potential source of incoherent results encountered. Thus, a lot of effort was put on explaining the input data format and importance of each parameter to untrained users in order to avoid as much illogical data as possible. It was also important to focus on results interpretation and solution selection. Furthermore, one recurring issue was that, in a lot of situations, real projects could not be optimized because of an unsatisfiability proof by the algorithm. Without any feedback to the users as to why it is the case, users were at a loss for identifying which constraints to relax or remove. An automated method for identifying causes and potential solutions to help overcome over-constrained problems appears to be an essential requirement for the successful use of the prototype in the field. We did use FindMUS [23] from the MiniZinc tool suite to help us identify the data inconsistencies. However, its usage required complete knowledge of the optimization model, thus was not a viable option for end users.

A comparison of the complexity of our seven instances to the complexity of PSPLIB [19], BL [5] and Pack [8] benchmarks, as evaluated by Artigues et al. [3], shows the difference between real-life and theoretical applications. Our set of realistic instances is more complex on average when looking at OS (precedence) and RS (resources over time), although the difference is small. This can be explained by the significant greater size of our instances. However, the literature benchmarks are widely more complex in terms of RF (resource usage) and DR (resources and precedence). That can be explained by the changes made by experts so the instances could be manually planned. Manually defined resources are also less restrictive than in computer generated instances.

It would have been interesting to test our CP approach on the three literature benchmarks, as well as the extended ones from Koné et al. [20]. We established that our set of seven instances, although closer to the ship refit reality, were less complex than the computer generated instances, thus more extensive tests would be necessary to complete the constraint programming prototype. Although our industrial partners may not be subject to high levels of precedence-resource complexity, it is important to be aware of the prototype's limits in the hopes of further improving the solution. Being able to consider more complex problems could also become a strategic advantage for shipyards, allowing them to include more constraints normally not considered with manual plans.

7 Conclusion

In this paper, we introduced a CP approach to the ship refit planning problem. Our prototype solution was successfully tested on seven realistic instances supplied by our industrial partners with varying levels of complexity, which demonstrated the CP applicability for this problem. The proposed CP model is highly related to the classical RCPSP model, while multiple extensions are considered to address problem-specific constraints and objectives. As a means to speedup the search of better solutions, we proposed to use the SBPS value selection heuristic. Its usage improved on average the objective value by 5%, 48% and 79% when minimizing respectively the makespan, the overtime costs and the robustness, as better solutions are found significantly faster than with our baseline heuristics.

Directions for future work include the integration of an optimization algorithm based on mixed-integer programming, and extending algorithms by considering task priority levels for scope optimization, i.e. when work requirements surpass the capacity. We also aim to further explore the use of probabilistic discrete-event simulations for robustness assessment, and the use of geospatial modeling and visualization to improve planning as well as users understanding.

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, April 1993. doi:10.1016/0895-7177(93)90068-A.

2 Rashpal Ahluwalia and Denis Pinha. Decision support system for production planning in the ship repair industry. *Industrial and Systems Engineering Review*, 2(1):52–61, July 2014.

3 Christian Artigues, Oumar Koné, Pierre Lopez, Marcel Mongeau, Emmanuel Néron, and David Rivreau. Benchmark instance indicators and computational comparison of methods. In *Resource-Constrained Project Scheduling*, pages 107–135. John Wiley & Sons, Ltd, 2008.

4 Philippe Baptiste, Claude Le Pape, and Wim Nuitjen. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. International Series in Operations Research & Management Science. Springer, Boston, MA, first edition, 2001.

5 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, January 2000. doi:10.1023/A:1009822502231.

6 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*, Lecture Notes in Computer Science, pages 63–79, Berlin, Heidelberg, 2002. Springer. doi:10.1007/3-540-46135-3_5.

7 J. Blazewicz, J. K. Lenstra, and A. H. G. Rinnooy Kan. Scheduling subject to resource constraints: Classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24, January 1983. doi:10.1016/0166-218X(83)90012-4.

8 Jacques Carlier and Emmanuel Néron. On linear lower bounds for the resource constrained project scheduling problem. *European Journal of Operational Research*, 149:314–324, September 2003. doi:10.1016/S0377-2217(02)00763-4.

9 Geoffrey G. Chu. *Improving Combinatorial Optimization*. PhD thesis, The University of Melbourne, 2011. GitHub: https://github.com/chuffed/chuffed.

10 Bert De Reyck and Willy Herroelen. On the use of the complexity index as a measure of complexity in activity networks. *European Journal of Operational Research*, 91(2):347–366, June 1996. doi:10.1016/0377-2217(94)00344-0.

11 Sophie Demassey. Mathematical programming formulations and lower bounds. In *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*, pages 49–62. John Wiley & Sons, Ltd, 2008.

12 Erik L. Demeulemeester and Willy S. Herroelen. New benchmark results for the resource-constrained project scheduling problem. *Management Science*, 43(11):1485–1492, November 1997. doi:10.1287/mnsc.43.11.1485.

13 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 John Hooker, editor, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 99–108, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-98334-9_7.

14 Hamed Fahimi, Yanick Ouellet, and Claude-Guy Quimper. Linear-time filtering algorithms for the disjunctive constraint and a quadratic filtering algorithm for the cumulative not-first not-last. *Constraints*, 23(3):272–293, July 2018. doi:10.1007/s10601-018-9282-9.

15 Thibaut Feydy, Adrian Goldwaser, Andreas Schutt, Peter J Stuckey, and Kenneth D Young. Priority Search with MiniZinc. In *ModRef 2017: The Sixteenth International Workshop on Constraint Modelling and Reformulation*, 2017.

16 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, February 2022. doi:10.1016/j.ejor.2021.05.004.

17 Willy Herroelen, Bert De Reyck, and Erik Demeulemeester. Resource-constrained project scheduling: A survey of recent developments. *Computers & Operations Research*, 25(4):279–302, April 1998. doi:10.1016/S0305-0548(97)00055-5.

18 IBM Maximo Application Suite. IBM, 2021. Website: https://www.ibm.com/ca-en/products/maximo.

19 Rainer Kolisch and Arno Sprecher. PSPLIB - A project scheduling problem library. *European Journal of Operational Research*, 96(1):205–216, January 1997. doi:10.1016/S0377-2217(96)00170-1.

20 Oumar Koné, Christian Artigues, Pierre Lopez, and Marcel Mongeau. Event-based MILP models for resource-constrained project scheduling problems. *Computers & Operations Research*, 38(1):3–13, January 2011. doi:10.1016/j.cor.2009.12.011.

21 Philippe Laborie. Complete MCS-based search: Application to resource constrained project scheduling. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, pages 181–186, San Francisco, CA, USA, July 2005. Morgan Kaufmann Publishers Inc.

22 Daniel Lafond, Dave Couture, Justin Delaney, Jessica Cahill, Colin Corbett, and Gaston Lamontagne. Multi-objective schedule optimization for ship refit projects: Toward geospatial constraints management. In Tareq Ahram, Redha Taiar, and Fabienne Groff, editors, *Human Interaction, Emerging Technologies and Future Applications IV*, Advances in Intelligent Systems and Computing, pages 662–669, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-74009-2_84.

23 Kevin Leo and Guido Tack. Debugging unsatisfiable constraint models. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming*, Lecture Notes in Computer Science, pages 77–93, Cham, 2017. Springer International Publishing. GitLab: https://gitlab.com/minizinc/FindMUS. doi:10.1007/978-3-319-59776-8_7.

24 Anthony A. Mastor. An experimental investigation and comparative evaluation of production line balancing techniques. *Management Science*, 16(11):728–746, July 1970. doi:10.1287/mnsc.16.11.728.

25 Microsoft Project. Microsoft, 2019. Website: https://www.microsoft.com/en-ca/microsoft-365/project/project-management-software.

26 M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535, June 2001. doi:10.1145/378239.379017.

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 Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, Lecture Notes in Computer Science, pages 529–543, Berlin, Heidelberg, 2007. Springer. Website: `https://www.minizinc.org/`. `doi:10.1007/978-3-540-74970-7_38`.

28 Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, September 2009. `doi:10.1007/s10601-008-9064-x`.

29 Yanick Ouellet and Claude-Guy Quimper. A $O(n \log^2 n)$ checker and $O(n^2 \log n)$ filtering algorithm for the energetic reasoning. In Willem-Jan van Hoeve, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Lecture Notes in Computer Science, pages 477–494, Cham, 2018. Springer International Publishing. `doi:10.1007/978-3-319-93031-2_34`.

30 Robert Pellerin, Nathalie Perrier, and François Berthaut. A survey of hybrid metaheuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 280(2):395–416, January 2020. `doi:10.1016/j.ejor.2019.01.063`.

31 Laurent Perron and Vincent Furnon. OR-Tools. Google, 2022. Website: `https://developers.google.com/optimization/`.

32 David Pisinger and Stefan Ropke. Large Neighborhood Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, International Series in Operations Research & Management Science, pages 399–419. Springer US, Boston, MA, 2010. `doi:10.1007/978-1-4419-1665-5_13`.

33 Primavera P6 Enterprise Project Portfolio Management (P6 EPPM). Oracle, 2022. Website: `https://docs.oracle.com/en/industries/construction-engineering/primavera-p6-project/index.html`.

34 A. Alan B. Pritsker, Lawrence J. Waiters, and Philip M. Wolfe. Multiproject scheduling with limited resources: A zero-one programming approach. *Management Science*, September 1969. `doi:10.1287/mnsc.16.1.93`.

35 Robert Richards and Richard Stottler. Complex project scheduling lessons learned from NASA, boeing, general dynamics and others. In *2019 IEEE Aerospace Conference*, pages 1–9, March 2019. `doi:10.1109/AERO.2019.8741996`.

36 Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

37 Andreas Schutt, Thibaut Feydy, and Peter J. Stuckey. Explaining time-table-edge-finding propagation for the cumulative resource constraint. In Carla Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Lecture Notes in Computer Science, pages 234–250, Berlin, Heidelberg, 2013. Springer. `doi:10.1007/978-3-642-38171-3_16`.

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

39 Paul Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming — CP98*, Lecture Notes in Computer Science, pages 417–431, Berlin, Heidelberg, 1998. Springer. `doi:10.1007/3-540-49481-2_30`.

40 Arno Sprecher. Scheduling resource-constrained projects competitively at modest memory requirements. *Management Science*, 46(5):710–723, 2000.

41 Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In Tobias Achterberg and J. Christopher Beck, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Lecture Notes in Computer Science, pages 230–245, Berlin, Heidelberg, 2011. Springer. `doi:10.1007/978-3-642-21311-3_22`.

42 Julien Vion and Sylvain Piechowiak. Une simple heuristique pour rapprocher DFS et LNS pour les COP. In *Actes des 13e Journées Francophones de la Programmation par Contraintes, JFPC 2017*, pages 38–45, Montreuil sur Mer, France, June 2017.

On Redundancy in Constraint Satisfaction Problems

Clément Carbonnel

CNRS, LIRMM, University of Montpellier, France

Abstract

A constraint language Γ has non-redundancy $f(n)$ if every instance of CSP(Γ) with n variables contains at most $f(n)$ non-redundant constraints. If Γ has maximum arity r then it has non-redundancy $O(n^r)$, but there are notable examples for which this upper bound is far from the best possible. In general, the non-redundancy of constraint languages is poorly understood and little is known beyond the trivial bounds $\Omega(n)$ and $O(n^r)$.

In this paper, we introduce an elementary algebraic framework dedicated to the analysis of the non-redundancy of constraint languages. This framework relates redundancy-preserving reductions between constraint languages to closure operators known as pattern partial polymorphisms, which can be interpreted as generic mechanisms to generate redundant constraints in CSP instances. We illustrate the power of this framework by deriving a simple characterisation of all languages of arity r having non-redundancy $\Theta(n^r)$.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Mathematics of computing → Discrete mathematics

Keywords and phrases Constraint satisfaction problem, redundancy, universal algebra, extremal combinatorics

Digital Object Identifier 10.4230/LIPIcs.CP.2022.11

Funding This work was supported by the AI Interdisciplinary Institute ANITI, funded by the French program "Investing for the Future – PIA3" under grant agreement no. ANR-19-PI3A-0004. The author also received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 952215.

1 Introduction

The constraint satisfaction problem (CSP) is a fundamental computer science problem with many applications in artificial intelligence and operational research. An instance of the CSP is a set of variables, a set of domain values, and a set of constraints, which are relations imposed upon certain sequences of variables. The goal is to decide whether it is possible to assign domain values to variables in such a way that all constraints are satisfied. The CSP is a natural common framework for a wide variety of well-studied combinatorial problems, such as satisfiability and graph homomorphism, and is in general intractable.

Following early work of Schaefer on the Boolean domain [25], Feder and Vardi initiated the systematic study of CSPs with fixed constraint languages and famously conjectured that all these "non-uniform" CSPs are either polynomial-time solvable or NP-complete [14]. This conjecture prompted a considerable research effort aimed at identifying generic sufficient conditions for the tractability of non-uniform CSPs, which eventually coalesced into a powerful, unified algebraic framework for analysing and classifying the complexity of constraint languages [2, 4]. After more than two decades of research, the Feder-Vardi conjecture was finally settled in the affirmative with two independent proofs by Bulatov [8] and Zhuk [26].

The success and flexibility of the algebraic framework motivated the study of constraint languages from a broader perspective. Beyond the classical "P versus NP-complete" question, classifications of constraint languages have been obtained for a wide variety of properties,

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 11; pp. 11:1–11:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

including solvability by specific classes of polynomial-time algorithms [3, 16], membership in fine complexity classes within P [12], learnability [10, 5], definability in certain logics [1, 22], and more.

In this paper we will study non-uniform CSPs from a different perspective. The central question we ask is the following: given a finite constraint language Γ of arity r, what is the maximum number of non-redundant constraints in a CSP instance over Γ? If we denote by n the number of variables, then this quantity (which we call the *non-redundancy* of Γ) is $O(n^r)$, and if Γ is non-trivial (i.e. at least one relation is neither empty nor complete) then it is $\Omega(n)$. As extreme examples, a set of affine relations over a finite field has non-redundancy $\Theta(n)$, while sets of r-clauses are easily seen to have non-redundancy $\Theta(n^r)$. Curiously, very little is known beyond these trivial bounds, especially outside the Boolean domain. The purpose of this paper is to describe an elementary algebraic framework for classifying the non-redundancy of constraint languages, which we illustrate by deriving a simple combinatorial characterisation of r-ary constraint languages with non-redundancy $\Theta(n^r)$.

We draw motivation for studying non-redundancy from two different lines of work. The task of learning a constraint network from answers to queries (sometimes called *constraint acquisition*) has attracted considerable interest in the past decades [7, 10], and a significant effort has been devoted to designing systems that can learn CSPs with as few queries as possible. In this context, it was observed in [5, 6] that the non-redundancy of a language Γ corresponds exactly to its *VC-dimension*, which is a lower bound on the number of yes/no queries (of any kind) that is necessary in order to learn exactly a constraint network over Γ. Therefore, any progress on lower bounds for non-redundancy immediately translates into unconditional, universal lower bounds for constraint acquisition. More generally, for applications where non-uniform CSPs are used to represent knowledge, the non-redundancy of a constraint language is a good estimate of its representational power: if Γ has non-redundancy $f(n)$ and arity r, then the number of n-variable CSP instances over Γ with pairwise distinct solution sets is $\Omega(2^{f(n)})$ and $O(2^{f(n)r \log n})$.

Our second motivation comes from a series of recent results on the sparsification of non-uniform Boolean CSPs [9, 20]. In these papers, the goal is to determine whether there exists a polynomial-time algorithm that takes as input an instance of CSP(Γ) (with up to roughly n^r constraints if Γ has arity r) and outputs an equisatisfiable instance of size $q(n)$, $q(n) = o(n^r)$. On the surface, this question looks quite different from estimating the non-redundancy of Γ: sparsification is in essence an algorithmic question, and sparsification algorithms are not limited to removing redundant constraints because they only have to maintain equisatisfiability. Nevertheless, all sparsification algorithms for NP-hard Boolean CSPs presented in [9, 20] operate purely by removing redundant constraints, and to the best of our knowledge all CSPs whose non-redundancy is known to be $O(n^q)$ also have an $O(n^q)$ sparsification algorithm. While non-redundancy and sparsifiability cannot be equivalent in general (for instance, all polynomial-time non-uniform CSPs have a sparsification algorithm that outputs an instance of size $O(1)$), this suggests that an improved understanding of non-redundancy in constraint languages would help design sparsification algorithms.

Our results

Our first contribution is a generic algebraic framework for the asymptotic study of non-redundancy in non-uniform CSPs. More precisely, we establish a tight connection between redundancy-preserving reductions for constraint languages and *pattern partial polymorphisms*, a type of closure operator that was recently introduced in the context of exponential algorithms for certain classes of non-uniform Boolean CSPs [21]. A key property of this algebraic duality is that both sides are easily interpretable in terms of non-redundancy. We observe that each pattern partial polymorphism of a constraint language Γ describes a rule to identify (or produce) redundant constraints in CSP instances over Γ. In some cases, knowledge of a single non-trivial pattern partial polymorphism of Γ can be sufficient to establish an improved upper bound on its non-redundancy.

Then, we combine our framework with a theorem of Erdős on the maximum cardinality of K_2^r-free hypergraphs [13] to obtain an explicit characterisation of those constraint languages of arity r having non-redundancy $\Theta(n^r)$. Incidentally, we show the existence of a small gap: either a constraint language of arity r has non-redundancy $\Theta(n^r)$, or it has non-redundancy $O(n^{r-\epsilon})$ for $\epsilon = 2^{1-r}$. This (improperly) extends a result of Chen et al. [9] for Boolean languages, which was obtained using very different methods. Beyond non-redundancy, our main result has direct consequences for sparsification, which will be discussed towards the end of the paper.

Related work

A recent series of papers on the sparsification of Boolean languages have established a number of results on the non-redundancy of constraint languages as byproducts. In [9], Chen et al. show that every Boolean language of arity r that does not contain an r-clause can be expressed using multivariate polynomials of total degree at most $r - 1$. Coupled with elementary arguments on Boolean clauses (see e.g. the proof of Lemma 15 in Section 3), this implies that the non-redundancy of any Boolean constraint language of arity at most r is either $\Theta(n^r)$ or $O(n^{r-1})$. Other results in the same paper imply a non-redundancy classification for Boolean constraint languages of arity at most 3, and a characterisation of symmetric Boolean constraint languages with linear non-redundancy. The framework presented in our paper is inspired from their methods, although it is extended to work with arbitrary domains and adapted to study specifically the non-redundancy of constraint languages.

Building upon these results, Lagerkvist and Wahlstrom [20] devised an $O(n)$ sparsification algorithm for the class of languages with a *Mal'tsev embedding*, which generalises linear equations over finite fields. Their algorithm operates by removing redundant constraints, and hence implies a similar bound on the non-redundancy of these languages. To the best of our knowledge, all languages known to have non-redundancy $O(n)$ belong to this class. The same paper also provides a sufficient condition for having non-redundancy $O(n^q), q > 1$ based on the closely related notion of *k-edge embedding*.

Bessiere et al. [5] initiated the direct study of non-redundancy of constraint languages, with a focus on applications in machine learning. They established the equivalence between non-redundancy and VC-dimension, classified the non-redundancy of constraint languages of arity at most 2, and identified a class of ternary constraint languages whose non-redundancy is $o(n^2)$ and cannot be fully determined using results based on algebraic embeddings.

2 Preliminaries

Relations, languages and constraint satisfaction problems

A *relation* R of arity $r = \text{ar}(R)$ over a domain D is a subset of D^r. Given a tuple t of length r and $S \subseteq \{1, \dots, r\}$, we denote by $t[S]$ the tuple obtained from t by discarding elements whose index is not in S. Similarly, the projection on $S \subseteq \{1, \dots, r\}$ of a relation R of arity r is denoted by $R[S] = \{t[S] \mid t \in R\}$. A (finite) *constraint language* Γ is a finite set of relations over a finite domain D, and the arity of a constraint language Γ is defined as the maximum arity of its relations. Given a constraint language Γ, a CSP instance over Γ is a pair (X, C), where X is a finite set of variables and C is a finite set of constraints, that is, pairs (R, S) with $R \in \Gamma$ and $S \in X^{\mathrm{ar}(R)}$. A *solution* to a CSP instance (X, C) is a mapping $\phi : X \to D$ such that for every $(R, S) \in C$, we have $\phi(S) \in R$. We will denote the set of all solutions to a CSP instance I by $\mathrm{sol}(I)$. The *constraint satisfaction problem over* Γ, denoted by $\mathrm{CSP}(\Gamma)$, takes as input a CSP instance I over Γ and asks whether $\mathrm{sol}(I)$ is non-empty.

Primitive-positive definitions and polymorphisms

Given a constraint language Γ, a relation R of arity r is *primitive-positive definable* (*pp-definable*) over Γ if there exists a first-order formula ψ with r free variables $x_1, \ldots, x_r$ that only uses existential quantification, conjunction, equality, and relations from Γ such that $R = \{(f(x_1), \ldots, f(x_r)) \mid \text{ f is a model of } \psi\}$. In that case, we will often write $R(x_1, \ldots, x_r) \equiv \psi$. If ψ is quantifier-free, then R is *qfpp-definable* over Γ. We denote by $\langle\Gamma\rangle$ (resp. $\langle\Gamma\rangle_{\nexists}$) the set of all relations that are pp-definable (resp. qfpp-definable) from Γ. It is well-known that $\mathrm{CSP}(\Gamma')$ is log-space reducible to $\mathrm{CSP}(\Gamma)$ for all $\Gamma' \subseteq \langle\Gamma\rangle$ [17]. If in addition we have $\Gamma' \subseteq \langle\Gamma\rangle_{\nexists}$, then the reduction is tighter: if $\mathrm{CSP}(\Gamma)$ is solvable in time $O(c^n)$, then so is $\mathrm{CSP}(\Gamma')$ [18].

Given a set D, a *partial operation* over D of arity k is an operation $f : D_f \to D$ with $D_f \subseteq D^k$. Given a relation R of arity r over D, f is a *partial polymorphism* of R if for all tuples $t_1, \ldots, t_k \in R$ such that for all $1 \leq i \leq r$ we have $(t_1[i], \ldots, t_k[i]) \in D_f$, the tuple $f(t_1, \ldots, t_k) = (f(t_1[1], \ldots, t_k[1]), \ldots, f(t_1[r], \ldots, t_k[r]))$ belongs to R. By extension, an operation is a partial polymorphism of a language if it is a partial polymorphism of each of its relations. A *polymorphism* of a relation over D is a partial polymorphism f with $D_f = D^k$. Given a language Γ, we denote by $\mathrm{pol}(\Gamma)$ the set of polymorphisms of Γ.

Geiger's theorem [15] states that for any two languages Γ, Γ' over the same domain, we have $\Gamma' \subseteq \langle\Gamma\rangle$ if and only if $\mathrm{pol}(\Gamma) \subseteq \mathrm{pol}(\Gamma')$. A similar duality was observed between qfpp-definability and partial polymorphisms by Romov [24]. These results form the foundation of the algebraic approach to non-uniform CSPs, in which the complexity of constraint languages is studied through the lens of their (partial) polymorphisms. We refer the reader to recent surveys for a more in-depth treatment of the subject [4][11].

Redundancy

In a CSP instance (X, C), a constraint $c \in C$ is *non-redundant* if and only if (X, C) and $(X, C\backslash\{c\})$ have different solution sets. Given a constraint language Γ, the *non-redundancy* of Γ, denoted by NRD_Γ, is the function that maps each $n \in \mathbb{N}$ to the maximum number of non-redundant constraints in an instance of $\mathrm{CSP}(\Gamma)$ with n variables. It is easily seen that if Γ is a constraint language of arity r that does not contain only empty or complete relations, then $\mathrm{NRD}_\Gamma(n) = O(n^r)$ and $\mathrm{NRD}_\Gamma(n) = \Omega(n)$. It is also known that the asymptotic behaviour of the NRD_Γ function for a finite language Γ is governed by that of its individual relations, as witnessed by these two inequalities:

$$\mathrm{NRD}_\Gamma \leq \sum_{R \in \Gamma} \mathrm{NRD}_{\{R\}} \qquad\qquad \mathrm{NRD}_\Gamma \geq \max_{R \in \Gamma}(\mathrm{NRD}_{\{R\}})$$

The second inequality holds because each instance over $\{R\}$ is also over Γ, and the first holds because the property of being non-redundant is monotone. (If $c = (S, R)$ is non-redundant in I, then it is non-redundant in the subinstance of I consisting only of those constraints with relation R. Repeating this reasoning with all $R \in \Gamma$ provides the desired upper bound.) Formal proofs can be found in [5]. In this paper we are only interested in the asymptotic behaviour of the NRD_Γ function; it follows from the inequalities above that classifying single-relation languages is sufficient to deduce a classification for all finite constraint languages.

3 Redundancy-preserving reductions

It is easily observed that primitive-positive definability does not preserve non-redundancy in general, in the sense that two constraint languages Γ_1 and Γ_2 with $\Gamma_1 \subseteq \langle \Gamma_2 \rangle$ and $\Gamma_2 \subseteq \langle \Gamma_1 \rangle$ may have very different non-redundancy asymptotics. (An extreme example is $\Gamma_1 = \{(0,0,1),(0,1,0),(1,0,0)\}$ and Γ_2 being the set of all ternary Boolean clauses. By the results of [9], $\mathrm{NRD}_{\Gamma_1}(n) = \Theta(n)$ but $\mathrm{NRD}_{\Gamma_2}(n) = \Theta(n^3)$. The pp-interdefinability of these languages is well known and can be verified by inspecting Post's lattice [23].) On the other hand, qfpp-definitions do preserve non-redundancy, but have limited expressive power. In this section, we attempt to construct an ideal notion of definability tailored for non-redundancy, with three goals in mind: the corresponding reductions between constraint languages must preserve non-redundancy bounds, a useful algebraic duality must exist, and the framework should be as general as possible.

We start by presenting our proposed notion of definability.

▶ **Definition 1.** *Let D be a set and Γ be a constraint language over D. We say that a relation R of arity r has an* fgpp-definition *over Γ if R has a pp-definition*

$$R(x_1,\dots,x_r) \equiv \exists y_1,\dots,y_q : \psi(x_1,\dots,x_r,y_1,\dots,y_q)$$

over $\Gamma \cup \{Q_g \mid g : D \to D\}$, where $Q_g = \{(d,g(d)) \mid d \in D\}$, and for each existentially quantified variable y_i there exists some x_j such that $Q_g(x_j,y_i)$ is an atom in ψ.

In Definition 1, "fgpp-definition" stands for *functionally guarded pp-definition*. Note that qfpp-definability implies fgpp-definability, but that fgpp-definability does not imply pp-definability in general. (This is due to the functional atoms Q_g, which may not belong to Γ.) On the Boolean domain, fgpp-definitions are equivalent to the *cone-definitions* of Chen et al. [9].

Given a constraint language Γ over D, let $\langle \Gamma \rangle_{\mathrm{fg}}$ denote the set of relations over D that are fgpp-definable over Γ. The next proposition is the first step towards proving that fgpp-definitions are suitable for studying the NRD function.

▶ **Proposition 2.** *Let Γ_1 and Γ_2 be two non-trivial languages over the same finite domain D. If $\Gamma_2 \subseteq \langle \Gamma_1 \rangle_{fg}$, then $NRD_{\Gamma_2}(n) = O(NRD_{\Gamma_1}(n))$.*

Proof. Let I be an instance of $\mathrm{CSP}(\Gamma_2)$ with variable set X, $|X| = n$, and exactly $\mathrm{NRD}_{\Gamma_2}(n)$ non-redundant constraints. Without loss of generality, we assume that no constraint in I is redundant.

Let $R \in \Gamma_2$ be some relation and $R(x_1,\dots,x_r) \equiv \exists y_1,\dots,y_q : \psi(x_1,\dots,x_r,y_1,\dots,y_q)$ be an fgpp-definition of R over Γ_1. For each constraint $c_i = (R,(x^i_1,\dots,x^i_r))$ in I, we introduce a set Y^i of q fresh variables $y^i_1,\dots,y^i_q$ and replace c_i with the set of constraints

$$S^i = \{(P,(z^1_j,\dots,z^k_j)) \mid P(z^1_j,\dots,z^k_j) \text{ is an atom in } \psi(x^i_1,\dots,x^i_r,y^i_1,\dots,y^i_q)\}$$

Repeating this process for all $R \in \Gamma_2$ and constraint c_i yields a CSP instance I^* over $\Gamma_1 \cup \{Q_g \mid g : D \to D\}$ whose solution set, when projected onto X, is exactly $\mathrm{sol}(I)$.

By construction, for each $y \in Y = \cup_i Y^i$ there exist $g : D \to D$ and $x \in X$ such that for all $\phi \in \mathrm{sol}(I^*)$, we have $\phi(y) = g(\phi(x))$. In particular, if there exist $y_1, y_2 \in Y$, $x \in X$ and $g : D \to D$ such that $y_1 = g(x)$ and $y_2 = g(x)$ then we have $\phi(y_1) = \phi(y_2)$ for all $\phi \in \mathrm{sol}(I^*)$. It follows that y_1 and y_2 can be merged into a single variable without changing the number of non-redundant constraints in I^*. After exhaustive application of this rule, we have $|Y| \leq n \cdot |D|^{|D|}$.

Now, we greedily remove redundant constraints from I^* until all constraints are non-redundant. Observe that this process cannot remove all constraints from a set S^i, for any i. Indeed, by assumption, for each constraint c_i in I there exists an assignment $\phi : X \to D$ that only violates c_i in I. This assignment can be extended to an assignment $\phi^* : X \cup Y \to D$ that is not a solution to I^* and may only violate constraints in S^i. Therefore, removing all of S_i would increase the solution set of I^*, which cannot happen since only redundant constraints are removed.

In addition, the language $\{Q_g \mid g : D^c \to D\}$ contains only functional constraints and hence has linear non-redundancy. (This follows, for example, from [5, Theorem 13].) Since $|X|+|Y| \leq n\cdot(1+|D|^{|D|})$, we deduce that I^* contains $O(n)$ constraints that are not from Γ_1.

By the three paragraphs above, I^* has at most $n \cdot (1 + |D|^{|D|})$ variables and at least $\mathrm{NRD}_{\Gamma_2}(n) - O(n)$ non-redundant constraints from Γ_1. By definition of NRD this implies $\mathrm{NRD}_{\Gamma_2}(n) = O(\mathrm{NRD}_{\Gamma_1}(n)) + O(n)$, and finally $\mathrm{NRD}_{\Gamma_2}(n) = O(\mathrm{NRD}_{\Gamma_1}(n))$ since Γ_1 is non-trivial. ◀

▶ **Example 3.** Let $p > 1$ be a prime number, $D = \{0, \dots, p-1\}$ and consider the relation $R = \{(x, y, z) \mid x^3 + y^3 + z^2 = 1\}$, where sum and product are understood as operations over the finite field of order p. If we let $R_{\mathrm{lin}} = \{(x, y, z) \mid x + y + z = 1\}$ and $f, g : D \to D$ such that $f(d) = d^3$ and $g(d) = d^2$, we can equivalently define R as

$$R(x, y, z) \equiv \exists a, b, c : R_{\mathrm{lin}}(a, b, c) \wedge Q_f(x, a) \wedge Q_f(y, b) \wedge Q_g(z, c)$$

which implies that $R \in \langle\{R_{\mathrm{lin}}\}\rangle_{\mathrm{fg}}$. From Proposition 2 and the fact that linear equations over finite fields have linear non-redundancy, we deduce that $\{R\}$ has non-redundancy $O(n)$.

▶ **Example 4.** Following [20], a language Γ_1 over non-empty domain D_1 has an *embedding* over a language Γ_2 over domain $D_2 \supseteq D_1$ if there exists a bijective function $h : \Gamma_1 \to \Gamma_2$ such that for all $R \in \Gamma_1$, $\mathrm{ar}(R) = \mathrm{ar}(h(R))$ and $R = h(R) \cap D_1$. If we interpret both Γ_1 and Γ_2 as languages over D_2 and define $g : D_2 \to D_2$ such that $g(d) = d$ if $d \in D_1$ and $g(d) = d_1^*$ otherwise (where d_1^* is an arbitrary value in D_1), then each $R \in \Gamma_1$ can be written as

$$R(x_1, \dots, x_r) \equiv h(R)(x_1, \dots, x_r) \bigwedge_{1 \leq i \leq r} Q_g(x_i, x_i)$$

and hence $\Gamma_1 \subseteq \langle\Gamma_2\rangle_{\mathrm{fg}}$. Therefore, by Proposition 2, embeddings preserve the non-redundancy asymptotics of constraint languages.

We will establish an algebraic duality for fgpp-definitions based on *pattern partial polymorphisms*, which were introduced by Lagerkvist and Wahlstrom [21] in a different context (the study of exponential algorithms for sign-symmetric Boolean languages).

A *polymorphism pattern* of arity k is a set of pairs (t, x), where t is a sequence of variables of length k and x occurs in t. A k-ary partial operation $f : D_f \to D$ *satisfies* a k-ary polymorphism pattern P if

$$D_f = \{(\phi(x_1), \dots, \phi(x_k)) \mid ((x_1, \dots, x_k), x) \in P, \phi : \{x_1, \dots, x_k\} \to D\}$$

and $f(\phi(x_1),\ldots,\phi(x_k)) = \phi(x)$ for all $((x_1,\ldots,x_k),x) \in P$, $\phi : \{x_1,\ldots,x_k\} \to D$. It follows from definition that for any pattern P and finite set D, there is at most one partial operation on D that satisfies P. We denote this function by f_P^D and call it the *interpretation* of P on D.

We say that a partial operation f is a *pattern partial operation* if it satisfies some polymorphism pattern P. We will often use the following equivalent characterisation.

▶ **Observation 5.** *Let D be a finite set, k be a nonnegative integer and $D_f \subseteq D^k$. A partial operation $f : D_f \to D$ is a pattern partial operation if and only if for every $t \in D_f$ and $g : D \to D$, we have that $g(t) \in D_f$ and $f \circ g(t) = g \circ f(t)$.*

Proof. Suppose that f is a pattern partial operation because it satisfies a certain polymorphism pattern P. In particular, for every $t \in D_f$ there exists some $((x_1,\ldots,x_k),x) \in P$ and $\phi : \{x_1,\ldots,x_k\} \to D$ such that $t = (\phi(x_1),\ldots,\phi(x_k))$. Then, for any mapping $g : D \to D$ we have $g(t) = (g(\phi(x_1)),\ldots,g(\phi(x_k)))$, which must belong to D_f as witnessed by the mapping $\phi' = g \circ \phi$. Furthermore, by definition we have $f(\phi'(x_1),\ldots,\phi'(x_k)) = \phi'(x)$, or equivalently $f \circ g(t) = g \circ f(t)$.

Conversely, suppose that for every $t \in D_f$ and $g : D \to D$, we have that $g(t) \in D_f$ and $f \circ g(t) = g \circ f(t)$. Let $D^P = \{x_1,\ldots,x_q\}$ be a set of variables in bijection with $D = \{d_1,\ldots,d_q\}$, and let P denote the pattern

$$\{((x_{i_1},\ldots,x_{i_k}),x_j) \mid (d_{i_1},\ldots,d_{i_k}) \in D_f, f(d_{i_1},\ldots,d_{i_k}) = d_j\}$$

Then, we must have $x_j \in \{x_{i_1},\ldots,x_{i_k}\}$ for any $((x_{i_1},\ldots,x_{i_k}),x_j) \in P$. Indeed, if it were not the case then there would exist a tuple $t = (d_{i_1},\ldots,d_{i_k}) \in D_f$ such that $f(t) \notin \{d_{i_1},\ldots,d_{i_k}\}$, and we would have $f \circ g(t) \neq g \circ f(t)$ for the mapping $g : D \to D$ such that $g(d) = d$ if $d \in \{d_{i_1},\ldots,d_{i_k}\}$ and $g(d) = d_{i_1}$ otherwise.

Furthermore, mappings ϕ from D^P to D can be identified with mappings from D to D, so with a slight abuse of notation we have

$$f(\phi(x_{i_1}),\ldots,\phi(x_{i_k})) = \phi(f(x_{i_1},\ldots,x_{i_k})) = \phi(x_j)$$

for all $\phi : D^P$, $((x_{i_1},\ldots,x_{i_k}),x_j) \in P$ and f satisfies P. ◀

On the Boolean domain, pattern partial operations are called *pSDI operations* [21] (for *partial self-dual idempotent operations*). Beyond the Boolean domain, notable examples of pattern partial operations are the *first Pixley partial operation* of [5] and the *universal Mal'tsev partial operations* of [20], the simplest of which is presented in Example 6.

▶ **Example 6.** Let P_2^M denote the polymorphism pattern

$$((x,x,y),y)$$
$$((y,x,x),y)$$

and consider the partial operation $f_{P_2^M}^D$ over some set D, which is an example of a pattern partial operation with domain $\{(d_1,d_2,d_3) \in D^3 \mid (d_1 = d_2) \text{ or } (d_2 = d_3)\}$. By definition, a binary relation R admits $f_{P_2^M}^D$ as a partial polymorphism if and only if it is *rectangular*, that is, R does not contain three tuples $(a,b),(a,c),(d,c)$ such that $(d,b) \notin R$. It can be further observed (although it is not immediately obvious) that a binary relation admits $f_{P_2^M}^D$ as a partial polymorphism if and only if it is fgpp-definable from the empty constraint language. This polymorphism pattern plays a critical role in the characterisation of the non-redundancy of binary constraint languages obtained in [5], and we will revisit it in the next section.

Throughout this note we will use $\mathrm{p^2pol}(\Gamma)$ to denote the set of all pattern partial polymorphisms of Γ. The following proposition shows that $\mathrm{p^2pol}(\Gamma)$ determines precisely the set of relations that are fgpp-definable over Γ.

▶ **Proposition 7.** *Let Γ_1 and Γ_2 be two constraint languages over the same finite domain D. Then, $p^2pol(\Gamma_1) \subseteq p^2pol(\Gamma_2)$ if and only if $\Gamma_2 \subseteq \langle \Gamma_1 \rangle_{fg}$.*

Proof. We first prove the backward implication. Suppose that $\Gamma_2 \subseteq \langle \Gamma_1 \rangle_{\mathrm{fg}}$ but there exists some pattern partial operation $f \in \mathrm{p^2pol}(\Gamma_1)$ of arity k that is not a partial polymorphism of some relation $R \in \Gamma_2$. Let $R(x_1, \ldots, x_r) \equiv \exists y_1, \ldots, y_q : \psi(x_1, \ldots, x_r, y_1, \ldots, y_q)$ be an fgpp-definition of R over Γ_1 and define $R_{\nexists}(x_1, \ldots, x_r, y_1, \ldots, y_q) \equiv \psi(x_1, \ldots, x_r, y_1, \ldots, y_q)$. First, observe that for all $g : D \to D$ and k tuples $t_1 = (d_1, g(d_1)), \ldots, t_k = (d_k, g(d_k))$ of Q_g such that $f(t_1, \ldots, t_k)$ is defined, it holds that

$$f(t_1, \ldots, t_k) = (f(d_1, \ldots, d_k), f(g(d_1), \ldots, g(d_k))) = (f(d_1, \ldots, d_k), g(f(d_1, \ldots, d_k))) \in Q_g$$

so f is a partial polymorphism of $\Gamma_1 \cup \{Q_g \mid g : D \to D\}$. Since $R_{\nexists}$ is qfpp-definable over $\Gamma_1 \cup \{Q_g \mid g : D \to D\}$, this implies that f is a partial polymorphism of $R_{\nexists}$. However, f is not a partial polymorphism of R, so there exist $k = \mathrm{ar}(f)$ tuples $t_1, \ldots, t_k \in R$ such that $f(t_1, \ldots, t_k)$ is defined and does not belong to R. Let $t'_1, \ldots, t'_k \in R_{\nexists}$ be such that $t'_l[1, \ldots, r] = t_l$ for all $l \leq k$. By Definition 1, there exists for each $r < i \leq r + q$ an index $j \leq r$ and a mapping $g : D \to D$ such that $t'_l[i] = g(t'_l[j])$ for all $l \leq k$. Since the domain of f is closed under all unary operations from D to D, $t_f = f(t'_1, \ldots, t'_k)$ is defined and belongs to $R^{\nexists}$, a contradiction since $t_f[1, \ldots, r] = f(t_1, \ldots, t_k) \notin R = R^{\nexists}[1, \ldots, r]$.

The forward implication is a bit more difficult. Let $\mathcal{R}$ denote the set of all relations R over D such that $R \notin \langle \Gamma_1 \rangle_{\mathrm{fg}}$ and every pattern partial polymorphism of Γ_1 is a partial polymorphism of R. Towards a contradiction, suppose that $\mathcal{R}$ is non-empty. Let R be a relation in $\mathcal{R}$ with minimum arity r. Note that $\langle \Gamma_1 \rangle_{\mathrm{fg}}$ contains all unary relations over D, so we may assume that $r \geq 2$. Now, we define

$$\hat{R} = \bigcap_{\substack{Q \in \langle \Gamma_1 \rangle_{\mathrm{fg}} \\ R \subseteq Q}} Q$$

and observe that $\hat{R}$ is well defined (because $D^r \in \langle \Gamma_1 \rangle_{\mathrm{fg}}$) and strictly contains R. In particular, there exists a certain tuple $t \in \hat{R} \backslash R$. We pick an arbitrary ordering $t_1, \ldots, t_m$ of the tuples of R, and for all $l \leq r$ we define the lth column of R as $c_l = (t_1[l], \ldots, t_m[l])$. Then, we define

$$D_f = \{g(c_l) \mid 1 \leq l \leq r,\ g : D \to D\}$$

and let $p = |D_f|$, as well as $\sigma : D_f \to \{1, \ldots, p\}$ be an arbitrary bijection such that $\sigma^{-1}(i) = c_i$ for $i \leq r$. Now, consider the relation $R_f(y_1, \ldots, y_r) \equiv \exists y_{r+1}, \ldots, y_p : \psi(y_1, \ldots, y_p)$, where $\psi(y_1, \ldots, y_p)$ is given by

$$\bigwedge_{\substack{Q \in \Gamma_1 \\ (t_1^q, \ldots, t_m^q) \in Q}} Q(y_{\sigma(t_1^q[1], \ldots, t_m^q[1])}, \ldots, y_{\sigma(t_1^q[\mathrm{ar}(Q)], \ldots, t_m^q[\mathrm{ar}(Q)])}) \quad \bigwedge_{\substack{i,j \leq p,\ g : D \to D: \\ \sigma^{-1}(i) = g(\sigma^{-1}(j))}} Q_g(y_i, y_j)$$

and the first conjunction is restricted to tuples of variables that are well-defined with respect to σ. By construction, the tuples of R_f are in one-to-one correspondance with the pattern partial polymorphisms of Γ_1 of arity m whose domain is the closure of $c_1, \ldots, c_r$ under all unary operations $D \to D$. In particular, R_f contains the tuples corresponding to the m partial projection operations on D_f and hence R_f contains R. Then, since R_f is fgpp-definable over Γ_1, it follows that $t \in R_f$. This particular tuple t corresponds to a certain pattern partial polymorphism f_t of Γ_1, of arity m, domain D_f and such that $f(c_l) = t[l]$ for all $l \leq r$. Since $t \notin R$, f_t is not a partial polymorphism of R, which concludes the proof. ◀

4 Pattern partial polymorphisms and redundancy

Recall from Section 2 that in order to study the function NRD_Γ, we can assume without loss of generality that Γ contains a single relation R. Then, it will be convenient to rephrase $\mathrm{CSP}(\Gamma)$ as a homomorphism problem: given a relation R_X over some finite set X of the same arity as R, is there a homomorphism from R_X to R? Here, a homomorphism is a mapping ϕ from X to D such that $\phi(t) \in R$ for all $t \in R_X$. We will use $\mathrm{hom}(R_X, R)$ to denote the set of all homomomorphisms from R_X to R. In this formulation, the constraint scopes are given by the tuples of R_X and a constraint (R, t), $t \in R_X$, is redundant if and only if $\mathrm{hom}(R_X, R) = \mathrm{hom}(R_X \backslash \{t\}, R)$.

▶ **Lemma 8.** *Let R_X, R be relations with respective domains X, D and let f_P^D be a k-ary partial polymorphism of R that satisfies a pattern P. If $t, t_1, \dots, t_k$ are tuples of R_X such that $t = f_P^X(t_1, \dots, t_k)$, then $hom(R_X, R) = hom(R_X \backslash \{t\}, R)$.*

Proof. For the sake of contradiction, suppose that there exists a homomorphism $h : X \to D$ such that $h(t) \notin R$ but $h(t_1), \dots, h(t_k) \in R$. Observe that $f_P^{X \cup D}$ is a partial polymorphism of R (when interpreted as a relation over $X \cup D$) and define $g : X \cup D \to X \cup D$ such that $g(u) = h(u)$ if $u \in X$ and $g(u) = u$ otherwise. Since $f_P^{X \cup D}$ is a pattern partial operation, we have that

$$f_P^{X \cup D}(g(t_1), \dots, g(t_k)) = g(f_P^{X \cup D}(t_1, \dots, t_k)) = g(f_P^X(t_1, \dots, t_k)) = g(t) = h(t) \notin R$$

which contradicts the fact that $f_P^{X \cup D}$ is a partial polymorphism of R. ◀

In essence, a (partial) polymorphism is an operator that combines solutions (tuples of values) to produce new ones. What this lemma says is that *pattern* partial polymorphisms can also be used to combine *constraints* and produce new ones that are valid for the instance, i.e. redundant. The is particularly interesting in light of the algebraic duality uncovered in Proposition 7: if Γ can fgpp-define a relation R with high non-redundancy, then Γ has high non-redundancy by Proposition 2, and if it cannot then Proposition 7 and Lemma 8 provide a non-trivial mechanism to identify redundant constraints that is valid for $\mathrm{CSP}(\Gamma)$ but not for $\mathrm{CSP}(\{R\})$.

▶ **Example 9.** Let R be a relation with the operation $f_{P_2^M}^D$ of Example 6 as partial polymorphism. Consider a CSP instance (R_X, R) and suppose that there exist four variables $x_1, x_2, y_1, y_2 \in X$ such that $(x_1, y_1), (x_1, y_2), (x_2, y_2), (x_2, y_1)$ are tuples of R_X (i.e. are scopes of constraints with relation R). Then, the pattern partial polymorphism $f_{P_2^M}^D$ combined with Lemma 8 implies that the constraint $(R, (x_2, y_1))$ is redundant, as it is the image through $f_{P_2^M}^X$ of the first three constraints.

Given a relation R over a set X and a set $\mathcal{F}$ of partial operations on X, we denote by $\mathcal{F}(R)$ the transitive closure of R under operations from $\mathcal{F}$. If no tuple t of R can be generated from tuples in $R \backslash \{t\}$ via an operation in $\mathcal{F}$, we say that R is *$\mathcal{F}$-independent.* The following two propositions are natural consequences of Lemma 8 regarding upper bounds on the NRD function.

▶ **Proposition 10.** *Let R be a relation over a set D, P_R be the set of polymorphism patterns that are satisfied by partial polymorphisms of R, and P_R^S denote the set of interpretations of P_R on set S. If for every relation R_X over a set X of n elements such that $ar(R_X) = ar(R)$ there exists a relation R_X^* of cardinality at most $f(n)$ such that $R_X^* \subseteq R_X \subseteq P_R^X(R_X^*)$, then $NRD_{\{R\}}(n) \leq f(n)$.*

Proof. Suppose that such a relation R^*_X exists for every relation R_X. Let (R_X, R) be an instance of CSP($\{R\}$) and $R_1 \subset R_2 \subset \ldots \subset R_q$ be the sequence of distinct relations obtained by transitive closure of R^*_X under P^X_R, with $R_1 = R^*_X$ and $R_j = R_X$. For every $1 \leq i < q$, there exists a pattern $P \in P_R$ and tuples $t_1, \ldots, t_k$ in R_i such that $R_{i+1} = R_i \cup \{t\}$, $t = f^X_P(t_1, \ldots, t_k)$. By Lemma 8, we have $\hom(R_i, R) = \hom(R_{i+1}, R)$. This is true for all i, so in particular we have $\hom(R^*_X, R) = \hom(R_X, R)$. Therefore, every non-redundant constraint in (R_X, R) must be of the form (R, t) with $t \in R^*_X$, and their total number is at most $f(n)$. ◀

▶ **Example 11.** Consider a relation R of arity r over a set D, and suppose that R has the pattern partial polymorphism $f^D_{P^M_2}$ of Examples 6 and 9. We will use Proposition 10 to show that $\mathrm{NRD}_{\{R\}}(n) \leq 2n^q$, where $q = \lceil r/2 \rceil$.

Let (R_X, R) be an instance of CSP($\{R\}$) with n variables and no redundant constraint. Let R^1_X denote the projection of R_X onto its first q indices and R^2_X be its projection onto the remainder. For simplicity, we will interpret R_X as a binary relation over disjoint domains R^1_X and R^2_X. Let G_X be the bipartite graph with domain $R^1_X \cup R^2_X$ and edge relation R_X. Observe that for any path $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ in G_X with an odd number of edges, repeated application of $f^X_{P^M_2}$ on the corresponding tuples of R_X eventually produces the edge (v_1, v_k). Therefore, the smallest subrelation R^*_X of R_X that contains R_X in its transitive closure via $f^X_{P^M_2}$ corresponds to a forest subgraph F_X of G_X. In particular, $|R^*_X| = |E(F_X)| \leq |R^1_X| + |R^2_X| \leq 2n^q$, and by Proposition 10, $\mathrm{NRD}_{\{R\}}(n) \leq 2n^q$.

▶ **Proposition 12.** *Let R be a relation over a set D, P_R be the set of polymorphism patterns that are satisfied by partial polymorphisms of R, and P^S_R denote the set of interpretations of P_R on set S. If every relation R_X over a set X of n elements that is P^X_R-independent and such that $ar(R_X) = ar(R)$ has cardinality at most $f(n)$, then $NRD_{\{R\}}(n) \leq f(n)$.*

Proof. Follows immediately from Proposition 10 as every minimal relation R^*_X with $R^*_X \subseteq R_X \subseteq P^X_R(R^*_X)$ is P^X_R-independent. ◀

We conclude this section with a straightforward lower bound on the non-redundancy of constraint languages that do not admit a certain polymorphism pattern related to the fgpp-definability of k-clauses, whose properties are well known on the Boolean domain.

▶ **Definition 13.** *Let $k \geq 2$ and $c_1, \ldots, c_{2^k-1}$ be the lexicographic ordering of the relation $\{x, y\}^k \backslash \{(y, \ldots, y)\}$ with respect to $y > x$. The* k-universal polymorphism pattern *P^u_k is the set of all pairs (t_i, y) with $t_i = (c_1[i], \ldots, c_{2^k-1}[i])$, $i \leq k$.*

The interpretation of P^u_k on the Boolean domain is called the *Boolean k-universal partial operation* [21]. In the definition above, the ordering of $\{x, y\}^k \backslash \{(y, \ldots, y)\}$ is not important: a different ordering would produce a different pattern, but it would be equivalent in the sense that it would have the same interpretation on all sets, up to a permutation of the arguments. (For instance, the pattern P^M_2 of Example 6 is equivalent to P^u_2.)

▶ **Example 14.** P^u_3 is the pattern given by the following pairs:

$$((x, x, x, x, y, y, y), y)$$
$$((x, x, y, y, x, x, y), y)$$
$$((x, y, x, y, x, y, x), y)$$

Observe that the left-hand side of these pairs corresponds to the three columns of the relation corresponding to a 3-clause with no negated literals, modulo the renaming $x \leftarrow 1$, $y \leftarrow 0$. The right-hand side is the missing tuple $(0, 0, 0)$ in the clause.

▶ **Lemma 15.** *Let R be a relation of arity r over a domain D and $r \geq k \geq 2$. If $f^D_{P^u_k} \notin$ $\mathrm{p}^2pol(\{R\})$, then $NRD_{\{R\}}(n) = \Omega(n^k)$.*

Proof. Suppose that $f^D_{P^u_k} \notin \mathrm{p}^2\mathrm{pol}(\{R\})$. For simplicity of notation we write $f = f^D_{P^u_k}$ and assume that $\{0, 1\} \subseteq D$. (Note that $|D| > 1$ since otherwise we would have $f \in \mathrm{p}^2\mathrm{pol}(R)$.) We claim that $C_k \in \langle\{R\}\rangle_{\mathrm{fg}}$, where $C_k(x_1, \dots, x_k) \equiv x_1 \vee \dots \vee x_k$. Let $p = |D_f|$ and $\sigma : D_f \to \{1, \dots, p\}$ be a bijection such that $\sigma^{-1}(i) = \phi(t_i)$ for all $i \leq k$, where $(t_i, y) \in P^u_k$ is as in Definition 13 and $\phi : \{x, y\} \to D$ is such that $\phi(x) = 1$ and $\phi(y) = 0$. We define

$$\psi(y_1, \dots, y_p) \equiv \bigwedge_{\substack{(t^*_1,\dots,t^*_k)\in R\\ \forall i,\, (t^*_1[i],\dots,t^*_k[i])\in D_f}} R(y_{\sigma((t^*_1[1],\dots,t^*_k[1]))}, \dots, y_{\sigma((t^*_1[r],\dots,t^*_k[r]))})$$

and note that the set of models of this formula are in one-to-one correspondance with the partial polymorphisms of R with domain D_f. Then, the formula

$$\phi(y_1, \dots, y_k) \equiv \exists y_{k+1}, \dots, y_p \, : \, \psi(y_1, \dots, y_p) \bigwedge_{\substack{i,j\leq p:\\ \exists g:D\to D \,:\, \sigma^{-1}(i)=g(\sigma^{-1}(j))}} Q_g(y_j, y_i)$$

is an fgpp-definition of a relation of arity k that contains every tuple of C_k (as projections with domain D_f are pattern partial operations) and cannot contain the tuple $(0, \dots, 0)$ (otherwise this tuple would extend to the model of ψ that corresponds to f, and by assumption $f \notin \mathrm{p}^2\mathrm{pol}(\{R\})$). Since the unary relation $\{(0), (1)\}$ is fgpp-definable from any language, we have $C_k \in \langle\{R\}\rangle_{\mathrm{fg}}$, as claimed.

Now, by Proposition 2 we need only prove that the language $\{C_k\}$ has non-redundancy $\Omega(n^k)$. A simple argument is to define for any n a CSP instance $I_n = (X, C)$ over $\{C_k\}$ with n variables and such that C contains one constraint $(C_k, (x_1, \dots, x_k))$ for all distinct $x_1, \dots, x_k$ in X. This instance has $\Omega(n^k)$ constraints, and none is redundant: for any constraint $c = (C_k, (x_1, \dots, x_k)) \in C$, the assignment that maps every variable to 1 except $x_1, \dots, x_k$ satisfies every constraint except c. ◀

▶ **Example 16.** Let $p > 2$, $D = \{1, \dots, p\}$ and consider the relation $R = \{(x, y, z) \in D^3 \mid \max(x, y) > z\}$. Observe that the set of tuples $S = (\{1, 2\}^2 \times \{2, 3\}) \backslash \{(2, 2, 2)\}$ is a subset of R, while the missing tuple $(2, 2, 2)$ does not belong to R. It follows that there exist some ordering $t_1, \dots, t_8$ of S such that $f^D_{P^u_3}(t_1, \dots, t_8)$ is defined and is equal to $(2, 2, 2)$, and hence $f^D_{P^u_3} \notin \mathrm{p}^2\mathrm{pol}(\{R\})$. By Lemma 15, $\{R\}$ has non-redundancy $\Omega(n^3)$, which is tight since R has arity 3.

In general, the largest value k for which Lemma 15 applies on a relation R gives a simple lower bound on its non-redundancy. This bound is unlikely to be tight in general, although we do not know any counter-examples.

5 A classification for languages with maximum non-redundancy

In this section, we will combine the lower bound of Lemma 15 with an upper bound derived from Proposition 12 and a well-known theorem in extremal hypergraph theory to prove our main result: a characterisation of constraint languages of arity r whose non-redundancy has the fastest possible asymptotic growth $\Theta(n^r)$. Our approach suggests a simple connection between the non-redundancy of constraint languages and hypergraph Turán numbers.

▶ **Definition 17** ([19]). *Let $\mathcal{H}$ be a family of r-uniform hypergraphs. The nth Turán number of $\mathcal{H}$, denoted by $ex(n, \mathcal{H})$, is the maximum number of edges in an r-uniform hypergraph with n vertices that does not contain any hypergraph in $\mathcal{H}$ as a subgraph.*

The following theorem of Erdős is a fundamental result on this topic.

▶ **Theorem 18** ([13]). *If K_2^r be the complete r-uniform r-partite hypergraph with vertex classes of size two and $K_2^r \in \mathcal{H}$, then $ex(n, \mathcal{H}) = O(n^{r-\epsilon})$, where $\epsilon = 2^{1-r}$.*

We will link relations and hypergraphs in a slightly unusual way. If R is a relation over X, then we define $\mathcal{H}^M(R)$ as the r-partite r-uniform hypergraph over vertex set $X_1, \ldots, X_r$, where each $X_i = \{x^i \mid x \in X\}$ is a copy of X, and edge set $\{\{x_1^1, \ldots, x_r^r\} \mid (x_1, \ldots, x_r) \in R\}$. Note that $\mathcal{H}^M(R)$ has cardinality exactly $|R|$, and its vertex set is of size $r \cdot |X| = O(|X|)$.

▶ **Lemma 19.** *Let R be a relation of arity $r \geq 2$ over a domain D with partial polymorphism $f^D_{P^u_r}$. If $I = (R_X, R)$ is an instance of $CSP(\{R\})$ and $\mathcal{H}^M(R_X)$ contains K_2^r as a subgraph, then I contains a redundant constraint.*

Proof. Suppose that K_2^r occurs in $\mathcal{H}^M(R_X)$ as a subgraph. Let $t_1, \ldots, t_{2^r}$ be 2^r tuples of R_X whose images in $\mathcal{H}^M(R_X)$ are the edges of a subgraph H isomorphic to K_2^r. Because both $\mathcal{H}^M(R_X)$ and K_2^r are r-uniform r-partite hypergraphs and K_2^r contains for each vertex class $\{x, y\}$ two edges e_1, e_2 with $e_1 = e_2 \backslash \{x\} \cup \{y\}$, the vertex classes of H are subsets of the vertex classes of $\mathcal{H}^M(R_X)$. This implies that for each $j \leq r$, there exist two elements x_j, y_j such that $t_i[j] \in \{x_j, y_j\}$ for all $i \leq 2^r$. Furthermore, all tuples t_i are distinct, so $\{t_i \mid i \leq 2^r\} = \Pi_{j \leq r}\{x_j, y_j\}$ and some tuple, say t_1, is exactly $(y_1, \ldots, y_r)$. After reordering lexicographically the other tuples $t_2, \ldots, t_2^r$ with respect to $y_j > x_j$, we obtain that $f^X_{P^u_r}(t_2, \ldots, t_{2^r}) = t_1$, so by Lemma 8 the constraint (R, t_1) is redundant in I and the claim follows. ◀

▶ **Corollary 20.** *Let R be a relation of arity $r \geq 2$ over a domain D. If $f^D_{P^u_r} \in p^2pol(\{R\})$, then $NRD_{\{R\}}(n) = O(n^{r-\epsilon})$, where $\epsilon = 2^{1-r} > 0$.*

Proof. Let $I = (R_X, R)$ be an instance of CSP($\{R\}$) with exactly $\mathrm{NRD}_{\{R\}}(n)$ constraint, all of which are non-redundant. By Lemma 19, $\mathcal{H}^M(R_X)$ does not contain K_2^r as a subgraph. Since $\mathcal{H}^M(R_X)$ has $O(n)$ vertices, by Theorem 18 it has $O(n^{r-\epsilon})$ edges. By construction we have $|R_X| = |\mathcal{H}^M(R_X)|$, so $|R_X| = O(n^{r-\epsilon})$ and finally $\mathrm{NRD}_{\{R\}}(n) = O(n^{r-\epsilon})$. ◀

Combining Corollary 20 with Lemma 15, we can fully characterise constraint languages with worst-case non-redundancy $\Theta(n^r)$.

▶ **Theorem 21.** *Let Γ be a constraint language with domain D and maximum arity $r \geq 2$. If $f^D_{P^u_r} \notin p^2pol(\Gamma)$ then $NRD_\Gamma(n) = \Theta(n^r)$, and otherwise $NRD_\Gamma(n) = O(n^{r-\epsilon})$, where $\epsilon = 2^{1-r} > 0$.*

Proof. Recall from Section 2 that the non-redundancy of a constraint language is asymptotically determined by the non-redundancy of its individual relations, i.e. $\mathrm{NRD}_\Gamma(n) = \Theta(\max_{R \in \Gamma} \mathrm{NRD}_{\{R\}}(n))$. If $f^D_{P^u_r} \notin \mathrm{p^2pol}(\Gamma)$ then there exists $R \in \Gamma$ such that $f^D_{P^u_r} \notin \mathrm{p^2pol}(\{R\})$, and by Lemma 15 we have $\mathrm{NRD}_\Gamma(n) = \Theta(n^r)$. If instead $f^D_{P^u_r} \in \mathrm{p^2pol}(\Gamma)$, then by Corollary 20 we obtain $\mathrm{NRD}_\Gamma(n) = O(n^{r-\epsilon})$. ◀

It is unlikely that the literature on Turán numbers can be used to derive tight upper bounds. Most results on this topic focus on forbidding a single fixed subhypergraph, while in our case the list of forbidden structures in irredundant instances is typically infinite and equipped with an algebraic structure; this discrepancy makes any bound obtained this way quite loose. For instance, on the elementary case $r = 2$, Corollary 20 only produces an upper bound of $O(n^{3/2})$ for binary rectangular relations while more direct arguments (Example 11) easily establish the tight bound $\Theta(n)$. Similarly, on Boolean languages the same result holds for $\epsilon = 1$, but proving such a bound using Lemma 8 (rather than polynomials, as in [9]) would necessitate a much deeper analysis of the pattern partial polymorphisms of constraint languages preserved by $f^D_{P^u_r}$.

Finally, we remark that the proof of Corollary 20 implies a simple polynomial-time sparsification algorithm for all languages Γ of arity r with $\text{NRD}_\Gamma(n) = o(n^r)$.

▶ **Theorem 22.** *Let Γ be a constraint language with domain D and maximum arity $r \geq 2$. If $f^D_{P^u_r} \in p^2pol(\Gamma)$, then there exists a polynomial time algorithm that takes an instance of CSP(Γ) as input and outputs an equisatisfiable instance of CSP(Γ) with $O(n^{r-\epsilon})$ constraints, where $\epsilon = 2^{1-r} > 0$.*

Proof. Let $I = (X, C)$ be an instance of CSP(Γ). For each relation $R \in \Gamma$, the algorithm constructs the relation $R_X = \{(x_1, \ldots, x_r) \mid (R, (x_1, \ldots, x_r)) \in C\}$ and enumerates all sequences $t_1, \ldots, t_{2^r}$ of tuples of R_X. For each sequence, it tests whether $t_1 = f^X_{P^u_r}(t_2, \ldots, t_{2^r})$ and discards the constraint (R, t_1) from I when the test succeeds. By Lemma 8, this process only removes redundant constraints. The algorithm then outputs the residual instance.

After this algorithm has terminated, for each relation R the corresponding relation R_X contains at most $O(n^{r-\epsilon})$ tuples because the r-uniform r-partite hypergraph $\mathcal{H}^M(R_X)$ has cardinality $|R_X|$ and does not contain K^r_2 as a subhypergraph. There are $O(1)$ distinct relations in Γ, so the total number of remaining constraints is $O(n^{r-\epsilon})$. ◀

6 Conclusion

We have presented an algebraic framework based on fgpp-definitions and pattern partial polymorphisms dedicated to the study of non-redundancy of constraint languages, extending earlier work on Boolean languages [9, 21]. Based on this framework, we have established a loose connection with extremal hypergraph theory and deduced a characterisation of constraint languages of arity r with non-redundancy $\Theta(n^r)$. The progress we have made in this paper is modest, and much is still unknown on this topic. We believe that the following challenges are the natural next steps towards a better understanding of non-redundancy.

Find a characterisation of constraint languages with non-redundancy $O(n)$. In this paper we have characterised constraint languages whose non-redundancy is the highest possible with respect to their arity, so it would be interesting to do the same for languages whose non-redundancy is the *lowest* possible. It is conceivable that this class coincides with that of languages with a finite Mal'tsev embedding [21] since no counter-example is known. However, proving that it is the case will likely require a better understanding of the pattern partial polymorphisms of these languages and lower bounds more sophisticated than those based on Boolean clauses.

Determine whether all r-ary constraint languages with non-redundancy $o(n^r)$ have non-redundancy $O(n^{r-1})$. This is known to be true for the Boolean domain by the results of Chen et al. [9], but for larger domains we are only able to prove the existence of a considerably smaller gap which vanishes as r grows. Both our approach and that of Chen et al. have intrinsic limitations when dealing simultaneously with large domains and large arities, so it would be interesting to see how they could be combined.

Determine the non-redundancy of all ternary constraint languages. A classification is known for binary languages (see [5], although a more direct proof follows from Example 11 and Lemma 15) and ternary Boolean languages [9], but not on ternary languages with arbitrary domains.

Clarify the relationship between non-redundancy, sparsification, and learnability. In particular, it would be interesting to determine whether non-redundancy $O(n^q)$ implies sparsification algorithms with output size $O(n^q)$ and whether non-redundancy is asymptotically equivalent to chain length, a closely related measure that characterises the efficiency of a class of learning algorithms for constraint acquisition [5].

References

1 Albert Atserias, Andrei A. Bulatov, and Anuj Dawar. Affine systems of equations and counting infinitary logic. *Theoretical Compututer Science*, 410(18):1666–1683, 2009. doi:10.1016/j.tcs.2008.12.049.

2 Libor Barto, Zarathustra Brady, Andrei Bulatov, Marcin Kozik, and Dmitriy Zhuk. Minimal taylor algebras as a common framework for the three algebraic approaches to the CSP. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'21)*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470557.

3 Libor Barto and Marcin Kozik. Constraint satisfaction problems solvable by local consistency methods. *Journal of the ACM*, 61(1):3:1–3:19, 2014. doi:10.1145/2556646.

4 Libor Barto, Andrei Krokhin, and Ross Willard. Polymorphisms, and how to use them. In *Dagstuhl Follow-Ups*, volume 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

5 Christian Bessiere, Clément Carbonnel, and George Katsirelos. Chain length and csps learnable with few queries. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI'20)*, pages 1420–1427, 2020. doi:10.1609/aaai.v34i02.5499.

6 Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Constraint acquisition via partial queries. In Francesca Rossi, editor, *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13)*, pages 475–481, 2013.

7 Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artificial Intelligence*, 244:315–342, 2017. doi:10.1016/j.artint.2015.08.001.

8 Andrei A. Bulatov. A dichotomy theorem for nonuniform csps. In Chris Umans, editor, *58th IEEE Annual Symposium on Foundations of Computer Science (FOCS'17)*, pages 319–330. IEEE Computer Society, 2017. doi:10.1109/FOCS.2017.37.

9 Hubie Chen, Bart M. P. Jansen, and Astrid Pieterse. Best-case and worst-case sparsifiability of Boolean csps. *Algorithmica*, 82(8):2200–2242, 2020. doi:10.1007/s00453-019-00660-y.

10 Hubie Chen and Matthew Valeriote. Learnability of solutions to conjunctive queries. *Journal of Machine Learing Research*, 20:67:1–67:28, 2019. URL: http://jmlr.org/papers/v20/17-734.html.

11 Miguel Couceiro, Lucien Haddad, and Victor Lagerkvist. A survey on the fine-grained complexity of constraint satisfaction problems based on partial polymorphisms. *J. Multiple Valued Log. Soft Comput.*, 38(1-2):115–136, 2022.

12 László Egri, Pavol Hell, Benoît Larose, and Arash Rafiey. Space complexity of list H-colouring: a dichotomy. In Chandra Chekuri, editor, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'14)*, pages 349–365. SIAM, 2014. doi:10.1137/1.9781611973402.26.

13 P Erdös. On extremal problems of graphs and generalized graphs. *Israel Journal of Mathematics*, 2(3):183–190, 1964.

14 Tomás Feder and Moshe Y. Vardi. The computational structure of monotone monadic snp and constraint satisfaction: A study through datalog and group theory. *SIAM Journal on Computing*, 28(1):57–104, 1998.

15 David Geiger. Closed systems of functions and predicates. *Pacific journal of mathematics*, 27(1):95–100, 1968.

16 Pawel M. Idziak, Petar Markovic, Ralph McKenzie, Matthew Valeriote, and Ross Willard. Tractability and learnability arising from algebras with few subpowers. *SIAM Journal on Computing*, 39(7):3023–3037, 2010. doi:10.1137/090775646.

17 Peter Jeavons, David Cohen, and Marc Gyssens. Closure properties of constraints. *J. ACM*, 44(4):527–548, July 1997. doi:10.1145/263867.263489.

18 Peter Jonsson, Victor Lagerkvist, Gustav Nordh, and Bruno Zanuttini. Strong partial clones and the time complexity of SAT problems. *J. Comput. Syst. Sci.*, 84:52–78, 2017. doi:10.1016/j.jcss.2016.07.008.

19 Peter Keevash. Hypergraph turán problems. *Surveys in combinatorics*, 392:83–140, 2011.

20 Victor Lagerkvist and Magnus Wahlström. Kernelization of constraint satisfaction problems: A study through universal algebra. In *Proceedings of the 23rd Conference on Principles and Practice of Constraint Programming (CP'17)*, pages 157–171, 2017. doi:10.1007/978-3-319-66158-2_11.

21 Victor Lagerkvist and Magnus Wahlström. Which np-hard SAT and CSP problems admit exponentially improved algorithms? *CoRR*, abs/1801.09488, 2018. arXiv:1801.09488.

22 Benoît Larose, Cynthia Loten, and Claude Tardif. A characterisation of first-order constraint satisfaction problems. *Logical Methods in Computer Science*, 3(4), 2007. doi:10.2168/LMCS-3(4:6)2007.

23 Emil L. Post. The two-valued iterative systems of mathematical logic. *Annals of Mathematics studies*, 1941. doi:10.2307/2268608.

24 Boris A Romov. The algebras of partial functions and their invariants. *Cybernetics*, 17(2):157–167, 1981.

25 Thomas J. Schaefer. The complexity of satisfiability problems. In *STOC '78: Proceedings of the tenth annual ACM Symposium on Theory of Computing (STOC'78)*, pages 216–226. Association for Computing Machinery, 1978. doi:10.1145/800133.804350.

26 Dmitriy Zhuk. A proof of the CSP dichotomy conjecture. *Journal of the ACM*, 67(5):30:1–30:78, 2020. doi:10.1145/3402029.

From Crossing-Free Resolution to Max-SAT Resolution

Mohamed Sami Cherif
Aix-Marseille Univ, Université de Toulon, CNRS, LIS, France

Djamal Habet
Aix-Marseille Univ, Université de Toulon, CNRS, LIS, France

Matthieu Py
Aix-Marseille Univ, Université de Toulon, CNRS, LIS, France

Abstract

Adapting a SAT resolution proof into a Max-SAT resolution proof without considerably increasing its size is an open problem. Read-once resolution, where each clause is used at most once in the proof, represents the only fragment of resolution for which an adaptation using exclusively Max-SAT resolution is known and trivial. Proofs containing non read-once clauses are difficult to adapt because the Max-SAT resolution rule replaces the premises by the conclusions. This paper contributes to this open problem by defining, for the first time since the introduction of Max-SAT resolution, a new fragment of resolution whose proofs can be adapted to Max-SAT resolution proofs without substantially increasing their size. In this fragment, called crossing-free resolution, non read-once clauses are used independently to infer new information thus enabling to bring along each non read-once clause while unfolding the proof until a substitute is required.

2012 ACM Subject Classification Theory of computation → Proof theory

Keywords and phrases Satisfiability, Proof, Max-SAT Resolution

Digital Object Identifier 10.4230/LIPIcs.CP.2022.12

1 Introduction

The maximum satisfiability (Max-SAT) problem is an optimization extension of the satisfiability (SAT) problem and consists, given a formula in Conjunctive Normal Form (CNF), in determining the maximum number of clauses that it is possible to satisfy by an assignment of the variables. This well known formalism is used to represent and solve many real-world and crafted problems making it of great academic and industrial interest [3, 4]. SAT and Max-SAT are strongly related and share many aspects. In fact, SAT solving techniques are often used in the context of Max-SAT solving, particularly in SAT-based and Branch and Bound (BnB) algorithms for Max-SAT [1, 2, 21]. Yet, in theory, bridging the gap between SAT and Max-SAT inference remains one of the main challenges in the last decade.

One of the first proof systems for Max-SAT is based on an inference rule called Max-SAT resolution [6, 7, 16, 17], which is an extension of the resolution rule [28] introduced in the context of SAT. Max-SAT resolution is sound, complete and is the most studied inference rule for Max-SAT, both in theory and practice [1, 5, 18, 19, 23, 24, 27]. However, adapting a resolution proof to get a valid Max-SAT resolution proof of reasonable size remains an open problem. Bonet et al. state that "*it seems difficult to adapt a classical resolution proof to get a Max-SAT resolution proof, and it is an open question if this is possible without increasing substantially*[1] *the size of the proof*" [7]. Indeed, unlike resolution, the Max-SAT resolution rule replaces the premises with the conclusions, which is necessary to maintain

[1] typically when the size of the adapted proof is exponential with respect to the size of the initial one.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 12; pp. 12:1–12:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Max-SAT equivalence after its application. Moreover, aside from the traditional resolvent clause, additional clauses[2] are also added to ensure Max-SAT equivalence. In [17], Larrosa et al. describe Max-SAT resolution as "*a movement of knowledge*". As such, read-once resolution proofs, where each clause is used once, represent the only fragment of resolution for which an immediate and trivial adaptation is possible [6, 7, 12]. Recent works [11, 24] try to circumvent this problem by allowing the use of the split rule, which intuitively allows to duplicate a clause by adding one literal, to linearly adapt tree-like resolution refutations. More specifically, the adaptation takes advantage of the structure of such proofs and applies the split rule to fix the non read-once input clauses. However, the resulting proofs are in the ResS proof system [18] in which Max-SAT resolution is augmented with the split rule. To bridge the gap between SAT and Max-SAT resolution, non read-once clauses need to be inferred using the clauses produced by Max-SAT resolution.

In this paper, we contribute to this open problem by identifying a new fragment of resolution, that we call crossing-free resolution, for which an adaptation using only Max-SAT resolution is possible without substantially increasing the size of the proof. Crossing-free derivations are defined using the ensuing derivations of non read-once clauses. Intuitively, non read-once clauses are used independently to infer new information in crossing-free resolution proofs. The adaptation of such proofs to Max-SAT resolution proofs is shown possible modulo some minor syntactic subtleties. Furthermore, we show that k-stacked diamond patterns, which were shown exponential for the adaptation in [24], fall within the crossing-free resolution fragment and can be adapted into Max-SAT resolution proofs without increasing their size.

This paper is organized as follows. Section 2 gives some necessary definitions and notations and presents the necessary background on resolution for SAT and Max-SAT as well as related work. The crossing-free resolution refinement is introduced in Section 3 and its adaptation to Max-SAT resolution is presented in Section 4. We study (k-stacked) diamond patterns and show that they can be adapted without increasing their size in Section 5. Finally, we conclude in Section 6.

2 Preliminaries

2.1 Definitions and Notations

Let X be the set of propositional variables. A literal l is a variable $x \in X$ or its negation $\overline{x}$. A clause C is a disjunction (or a set) of literals. If $|C| = 1$, C is a unit clause. A formula in Conjunctive Normal Form (CNF) ϕ is a conjunction (or a multiset) of clauses. An assignment $I : X \to \{true, false\}$ maps each variable to a boolean value and can be represented as a set of literals. A literal l is satisfied (resp. falsified) by an assignment I if $l \in I$ (resp. $\overline{l} \in I$). A clause C is satisfied by an assignment I if at least one of its literals is satisfied by I, otherwise it is falsified by I. The empty clause $\square$ contains zero literals and is always falsified. A clause C is a tautology if it contains both a literal l and its negation $\overline{l}$, i.e., $\exists l \in C$ s.t $\overline{l} \in C$, and in such case it is always satisfied. A clause C opposes a clause C' if C contains a literal whose negation is in C', i.e., $\exists l \in C$ s.t $\overline{l} \in C'$. We denote $var(l)$, $var(C)$ and $var(\phi)$ the variables appearing respectively in the literal l, the clause C and the formula ϕ. The width of a clause C is the number of literals occurring in it. A CNF formula ϕ is satisfied by an assignment I, that we call model of ϕ, if each clause $C \in \phi$ is satisfied by I, otherwise it is falsified by I.

[2] referred to as compensation clauses

Solving the Satisfiability (SAT) problem consists in determining whether there exists an assignment I that satisfies a given CNF formula ϕ. In the case where such an assignment exists, we say that ϕ is satisfiable, otherwise we say that ϕ is unsatisfiable or inconsistent. The cost of an assignment I, denoted $cost_I(\phi)$, is the number of clauses falsified by I. The Maximum Satisfiability (Max-SAT) problem is an optimization extension of SAT which, for a given CNF formula ϕ, consists in determining the maximum number of clauses that can be satisfied by an assignment of the variables. Equivalently, it consists in determining the minimum number of clauses that each assignment must falsify, i.e., $\min_I cost_I(\phi)$.

2.2 Resolution for SAT

A well-known proof and refutation system for SAT is based on the resolution rule [28]. Given two opposed clauses, this rule, defined below, deduces a resolvent clause which can be added to the formula. A resolution proof or derivation of a clause C is a finite sequence of resolutions starting from the clauses of ϕ and deducing C usually represented as a finite sequence of clauses. If C is the empty clause $\square$, the proof is referred to as a refutation of ϕ. A resolution proof can also be represented in the form of a Directed Acyclic Graph (DAG) whose nodes are clauses in the proof either having two or zero incoming arcs (resp. if they are resolvents or clauses of the initial formula). The size of a resolution derivation π, denoted $s(\pi)$, is the number of resolvents in it whereas its width, denoted $w(\pi)$, is the maximum width of all its clauses.

▶ **Definition 1** (Resolution [28]). *Given two opposed clauses C_1 and C_2, the resolution rule is defined as follows:*

$$\frac{C_1 = x \vee A \qquad\qquad C_2 = \overline{x} \vee B}{C_3 = A \vee B}$$

Many restricted classes of resolution have been studied in the literature, e.g read-once resolution [13], tree (or tree-like) resolution [15] and linear resolution [22] among others. In particular, a resolution proof is read-once if each clause is used at most once in the proof. Similarly, a resolution derivation is tree-like if every intermediate clause, i.e., resolvent, is used at most once in the derivation. Linear resolution, defined below, lies between tree-like and general resolution in terms of proof complexity [8, 9]. In this fragment, the proofs are linear in the sense that each deduced clause is used as premise in the next resolution step. Note that, when the first condition of (c) holds in the definition, the clause D_i is called the input parent clause of C_{i+1}.

▶ **Definition 2** (Linear resolution [22]). *Let ϕ be a CNF formula and C be a clause. A linear resolution derivation of C from ϕ is a sequence of clauses $C_1, ..., C_m$ such that:*

(a) *C_1 is a clause in ϕ*

(b) *C_m is the clause C*

(c) *For every $i < m$, C_{i+1} is the resolvent of C_i either with a clause D_i from ϕ or with a clause C_k for some $k < i$.*

2.3 Resolution for Max-SAT

One of the first and most studied proof systems for Max-SAT is the Max-SAT resolution calculus (MaxRes) which relies on an inference rule extending resolution for Max-SAT [6, 7, 16, 17]. Other than the resolvent clause, this rule, called Max-SAT resolution and defined below, introduces new clauses referred to as compensation clauses and essential to preserve Max-SAT equivalence. As a sound and complete rule for Max-SAT [6, 7], Max-SAT resolution plays an important role in the context of Max-SAT theory and solving [5, 18, 24, 27]. In particular, for a given CNF formula, it is possible to generate a Max-SAT resolution proof of its optimum by applying the saturation algorithm [7]. Furthermore, it is extensively used and studied in the context of Branch and Bound algorithms for Max-SAT [1, 10, 14, 19] and more marginally in the context of SAT-based ones [12, 23].

▶ **Definition 3** (Max-SAT equivalence)**.** *Let ϕ and ϕ' be two CNF formulas. ϕ and ϕ' are Max-SAT equivalent iff for any assignment $I : var(\phi) \cup var(\phi') \rightarrow \{true, false\}$, we have $cost_I(\phi) = cost_I(\phi')$.*

▶ **Definition 4** (Max-SAT resolution [6, 7, 16, 17])**.** *Given two opposed clauses C_1 and C_2, the Max-SAT resolution rule is defined as follows:*

$$\begin{array}{c} C_1 = x \lor A \qquad C_2 = \overline{x} \lor B \\ \hline C_r = A \lor B \\ CC_1 = x \lor A \lor \overline{B} \\ CC_2 = \overline{x} \lor \overline{A} \lor B \end{array}$$

where C_r is the resolvent clause and CC_1, CC_2 are compensation clauses.

Note that the following rewriting is used to represent the compensation clauses in compacted form: $C \lor \overline{a_1 \lor a_2 \lor ... \lor a_n} = (C \lor \overline{a_1}) \land (C \lor a_1 \lor \overline{a_2}) \land ... \land (C \lor a_1 \lor a_2 \lor ... \lor \lor \overline{a_n})$. This rewriting was introduced in [17] as a recursive rule to transform the compensation clauses into CNF form. This also entails that the Max-SAT resolution rule depends on the ordering of the literals, as reported in [7, 17]. For the sake of simplification, we will allow the use of this rewriting as two full-fledged rules to manipulate clauses in compacted form. We will refer to the left-right rewriting as expansion and right-left one as compaction. This may entail abusing some notations but it is useful to further simplify the proofs. Furthermore, given three sets of literals A, B and C, the equality $C \lor A \lor \overline{B} \overset{*}{=} C \lor A \lor \overline{A \lor B}$ is sound for Max-SAT as reported in [17] (c.f. Remark 13) and may be as such used in the proofs. We discuss these subtleties following Theorem 14 in Section 4.

A Max-SAT resolution proof or derivation of a formula ϕ' from ϕ is a finite sequence of Max-SAT resolutions starting from the clauses of ϕ and deducing ϕ' and is usually represented as a finite sequence of formulas. Note that we may allow the addition of tautological clauses to any formula in the proof. We discuss this syntactic subtlety at the end of Section 4. A Max-SAT resolution proof can also be represented as a bipartite DAG whose nodes are either clauses or inference steps (in which case they will be omitted for more simplicity). A sequence of Max-SAT resolution steps deducing one empty clause is referred to as Max-SAT resolution refutation. For a given CNF formula, it is possible to generate a Max-SAT resolution proof of its optimum by applying the saturation algorithm [7]. Note that other inference rules and proof systems were also studied in the context of Max-SAT [5, 11, 18, 20, 27].

Unlike resolution, the Max-SAT resolution rule replaces the premises by the conclusions. Larrosa et al. describe Max-SAT resolution as "*a movement of knowledge*" [17]. Because of this specificity, it is not easy to adapt a resolution proof to obtain a Max-SAT resolution proof. Indeed, in resolution proofs, several resolution steps can share the same premise, because the premises are not consumed after the application of a resolution step. On the other hand, the premises of a Max-SAT resolution step are consumed after its application. Consequently, the immediate adaptation of a resolution proof for Max-SAT is only possible if it is read-once [6, 7, 12]. In this fragment, it is simply sufficient to replace every resolution step in the proof by a Max-SAT resolution step to produce a Max-SAT resolution proof of similar size. However, adapting any resolution proof to a Max-SAT proof without substantially increasing its size remains an open problem.

Recent works [11, 24] augment the Max-SAT resolution rule by the split rule defined below, forming a new system stronger than MaxRes and called ResS [18], to linearly adapt tree-like resolution refutations into ResS refutations. More specifically, the adaptation takes advantage of the structure of such proofs and applies the split rule, which intuitively allows to duplicate a clause by adding one literal, to fix the non read-once input clauses. Furthermore, the substitution algorithm introduced in [26] also enables to generate substitutes for non read-once clauses using SAT oracles but no guarantee is provided for the size of the computed ResS refutations. To the best of our knowledge, read-once resolution remains the only fragment of resolution for which an adaptation using exclusively Max-SAT resolution is possible without substantially increasing the proof size. In the next section, we define a new refinement of resolution for which this is possible.

▶ **Definition 5** (Split). *Given a clause C and variable x, the split rule is defined as follows:*

$$\frac{C}{x \vee C \qquad \overline{x} \vee C}$$

3 Crossing-Free Resolution

The main difficulty in adapting resolution proofs to Max-SAT resolution ones lies in inferring a substitute for non read-once clauses. Indeed, such clauses must be naturally inferred using Max-SAT resolution while unfolding (i.e., reading and applying) the initial resolution proof, contrary to previous works [11, 24] where non read-once clauses are artificially fixed using the split rule before the actual unfolding of the proof. In this section, we define a new fragment of resolution, referred to as crossing-free resolution. The idea behind this refinement is to ensure enough manoeuvrability of proofs in terms of structure in order to infer substitutes for non read-once clauses when necessary. To this end, we define below the notion of ensuing derivation of a non read-once clause. Intuitively, this particular derivation is ensued from a non read-once clause in the sense that it is sufficient to delimit the impact of its multiple use. Note that a node where a set of given paths in a resolution proof intersect will be referred to as their junction node.

▶ **Definition 6** (Ensuing derivation). *Let ϕ be a CNF formula and π a resolution derivation of clause C from ϕ. The ensuing derivation of a non read-once clause C' in π, denoted $ED(C')$, is the sub-derivation of π formed by all the resolution steps in the paths starting from C' in π until their first junction node. We call the clause derived in the junction node, the ensued clause of C', denoted $EC(C')$.*

▶ **Example 7.** We consider the resolution derivation π represented in Figure 1 of clause $C = x_6$ from the formula $\phi = \{\overline{x_1} \vee x_3 \vee \overline{x_4},\ x_4 \vee x_5,\ x_4 \vee \overline{x_5},\ x_1 \vee \overline{x_4},\ x_2 \vee \overline{x_4} \vee x_6,\ x_5 \vee \overline{x_7},\ \overline{x_2} \vee \overline{x_3} \vee x_7,\ \overline{x_5} \vee \overline{x_7}\}$. The non read-once clauses x_4 and $\overline{x_2} \vee \overline{x_3} \vee x_7$ and their ensuing derivations are respectively represented in red and blue. Furthermore, we have $EC(x_4) = x_6$ and $EC(\overline{x_2} \vee \overline{x_3} \vee x_7) = \overline{x_2} \vee \overline{x_3}$.

Recall that clauses are consumed after the application of Max-SAT resolution. Therefore, it seems difficult to adapt resolution derivations in which ensuing derivations of non read-once clauses cross. Indeed, in such cases, the formula can significantly evolve as compensation clauses may be used while others may be generated. As such, crossing-free resolution ensures that ensuing derivations are disjoint, i.e., do not cross, as defined below.

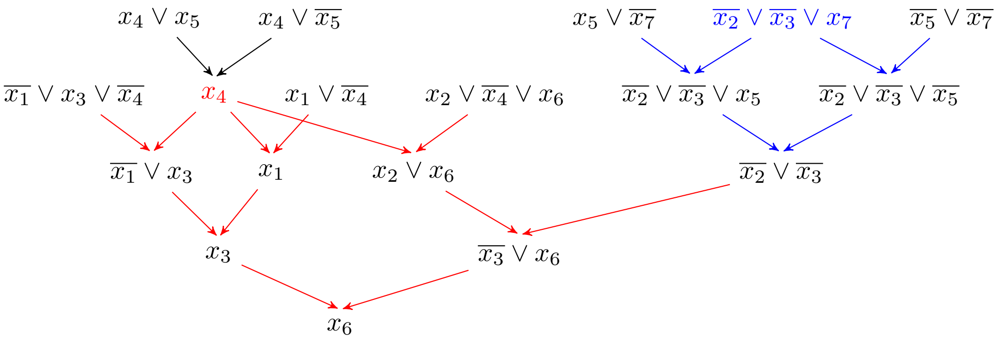

$x_4 \vee x_5$
$x_4 \vee \overline{x_5}$
$x_5 \vee \overline{x_7}$
$\overline{x_2} \vee \overline{x_3} \vee x_7$
$\overline{x_5} \vee \overline{x_7}$
$\overline{x_1} \vee x_3 \vee \overline{x_4}$
x_4
$x_1 \vee \overline{x_4}$
$x_2 \vee \overline{x_4} \vee x_6$
$\overline{x_2} \vee \overline{x_3} \vee x_5$
$\overline{x_2} \vee \overline{x_3} \vee \overline{x_5}$
$\overline{x_1} \vee x_3$
x_1
$x_2 \vee x_6$
$\overline{x_2} \vee \overline{x_3}$
x_3
$\overline{x_3} \vee x_6$
x_6

Figure 1 Ensuing derivations in a crossing-free resolution proof.

▶ **Definition 8** (Crossing-free resolution derivation). *Let ϕ be a CNF formula and π a resolution derivation of clause C from ϕ. π is crossing-free iff for every pair of non read-once clauses (C_1, C_2), $ED(C_1)$ and $ED(C_2)$ are disjoint, i.e., they do not contain a shared arc.*

▶ **Example 9.** We consider the same formula ϕ in Example 7. The resolution derivation π of clause $C = x_6$ from ϕ represented in Figure 1 is crossing-free since the ensuing derivations of the non read-once clauses x_4 and $\overline{x_2} \vee \overline{x_3} \vee x_7$ are disjoint.

Note that the crossing-free resolution refinement entails an interesting property established in the following proposition. Intuitively, this property ensures that non read-once clauses are used independently to infer new information in crossing-free resolution proofs. This entails that each ensuing derivation in a crossing-free resolution proof can be adapted independently as described in the next section.

▶ **Proposition 10.** *Let ϕ be a CNF formula, π be a crossing-free resolution derivation of clause C from ϕ and C' a non read-once clause in π. Every clause Cl in $ED(C')$ s.t $Cl \notin \{C', EC(C')\}$ is read-once.*

Proof. Let Cl be a clause in $ED(C')$ s.t $Cl \notin \{C', EC(C')\}$. Clearly, if Cl is not read once, $ED(Cl)$ shares at least one arc with $ED(C')$ which is absurd since π is crossing-free. ◀

4 From Crossing-Free Resolution to Max-SAT Resolution

In this section, we show that crossing-free resolution derivations can be adapted to Max-SAT resolution derivations modulo some minor syntactic subtleties without substantially increasing their size. In the following proposition, we first provide some patterns which will be encountered in the adaptation.

▶ **Proposition 11.** *Let A, B, C and $\{l\}$ be four sets of literals s.t $|C| > 0$. The following deductions can be done in $O(|C|)$ inference steps:*

(a) $(A \vee C) \wedge (B \vee \overline{C}) \vdash_{MaxRes} A \vee B$
(b) $(l \vee A \vee \overline{C}) \wedge (\overline{l} \vee B) \vdash_{MaxRes} A \vee B \vee \overline{C}$
(c) $(l \vee A \vee \overline{C}) \wedge (\overline{l} \vee B \vee \overline{C}) \vdash_{MaxRes} A \vee B \vee \overline{C}$

Proof. We provide the proof for case (a) by induction on $|C| = n$:

- If $n = 1$, then $C = \{l'\}$. Clearly, $(A \vee l') \wedge (B \vee \overline{l'}) \vdash_{MaxRes} A \vee B$ by application of a Max-SAT resolution step on literal $var(l')$.

- Suppose $n > 1$ and let $l' \in C$. By the induction hypothesis, we can deduce $(A \vee C) \wedge (B \vee \overline{C \setminus \{l'\}}) \vdash_{MaxRes} A \vee B \vee l'$ in $n-1$ inference steps. Furthermore, $B \vee \overline{C} = (B \vee \overline{C \setminus \{l'\}}) \wedge (B \vee \overline{l'})$ by expansion and $(A \vee B \vee l') \wedge (B \vee \overline{l'}) \vdash_{MaxRes} A \vee B$ by application of a Max-SAT resolution step on variable $var(l')$. Therefore, we conclude that we can deduce $(A \vee C) \wedge (B \vee \overline{C})\} \vdash_{MaxRes} A \vee B$ in $O(n)$ inference steps.

Proofs for cases (b) and (c) are similar by induction on $|C|$. ◀

Next, we start dealing with the adaptation of crossing-free resolution derivations and particularly ensuing derivations. To generate a substitute for a non read-once clause, note that we can use the literals in the junction nodes (c.f. Lemma 1 in [24]) of an ensuing derivation, i.e., nodes where paths starting from the non read-once clause intersect. To generate such substitutes using Max-SAT resolution, we start by dealing with read-once linear parts in the proof. Informally, we want to drag (i.e., bring along) each non read-once clause while unfolding the proof until they are reused. This is formally established for read-once linear parts of the proof in the following lemma. Note that the implications of equality $\stackrel{*}{=}$ in the proof will be further discussed at the end of the section.

▶ **Lemma 12.** *Let ϕ be a CNF formula, $\pi = C_1, ..., C_{s(\pi)}$ be a read-once linear resolution derivation of clause $C \neq \square$ from ϕ. We can deduce $\phi \vdash_{MaxRes} C \wedge (C_1 \vee \overline{C})$ in $O(s(\pi) \times w(\pi))$ inference steps.*

Proof. Let $m = s(\pi)$. Since π is read-once, it can be trivially adapted into a Max-SAT resolution derivation of C from ϕ of the same size by replacing every resolution step with a Max-SAT resolution step [6, 7, 12]. Next, we prove by induction on $i \in \{1, .., m-1\}$ that we can infer $C'_i = C_1 \vee \overline{C_{i+1}}$ at the i^{th} Max-SAT resolution step:

- For $i = 1$, the first Max-SAT resolution on clauses $C_1 = l_1 \vee A_1$ and $D_1 = \overline{l_1} \vee B_1$ w.r.t $var(l_1)$ generates the following compensation clause:

  $$CC_{1|1} = l_1 \vee A_1 \vee \overline{B_1} \stackrel{*}{=} l_1 \vee A_1 \vee \overline{A_1 \vee B_1} = C_1 \vee \overline{C_2} = C'_1$$

  Note that to establish the equality $\stackrel{*}{=}$, we can add the tautological clauses $l_1 \vee A_1 \vee B_1 \vee \overline{A_1}$ (or alternatively $l_1 \vee A_1 \vee \overline{A_1}$) to the formula) in which case $l_1 \vee A_1 \vee \overline{A_1 \vee B_1}$ can be trivially inferred by compaction. Furthermore, if D_1 is a unit clause, $CC_{1|1}$ is not generated. However, we can simply add the tautological clauses $l_1 \vee A_1 \vee \overline{A_1}$ which correspond to $C_1 \vee \overline{C_2}$ since $D_1 = \overline{l_1}$ (i.e., B_1 is empty).

$$\begin{tikzcd}
C_i = l_i \vee A_i \arrow[d, "var(l_i)"] & D_i = \overline{l_i} \vee B_i \arrow[dl] & & C'_{i-1} = C_1 \vee \overline{C_i} = C_1 \vee \overline{l_i \vee A_i} \arrow[dl, dashed] \arrow[ddr, dashed] & \\
C_{i+1} = A_i \vee B_i & & C_1 \vee \overline{l_i} \vee A_i \arrow[dd, "var(l_i)"'] & & \\
CC_{1|i} = l_i \vee A_i \vee \overline{B_i} \arrow[drr] & & & & C_1 \vee \overline{A_i} \arrow[ddl, dashed] \\
 & & C_1 \vee A_i \vee \overline{B_i} \arrow[dr, dashed] & & \\
 & & & C'_i = C_1 \vee \overline{A_i \vee B_i} = C_1 \vee \overline{C_{i+1}} &
\end{tikzcd}$$

Figure 2 Induction step to infer C'_i at the i^{th} step. Solid lines represent the application of the Max-SAT resolution rule whereas dashed lines represent compaction or expansion. Unused compensation clauses are omitted.

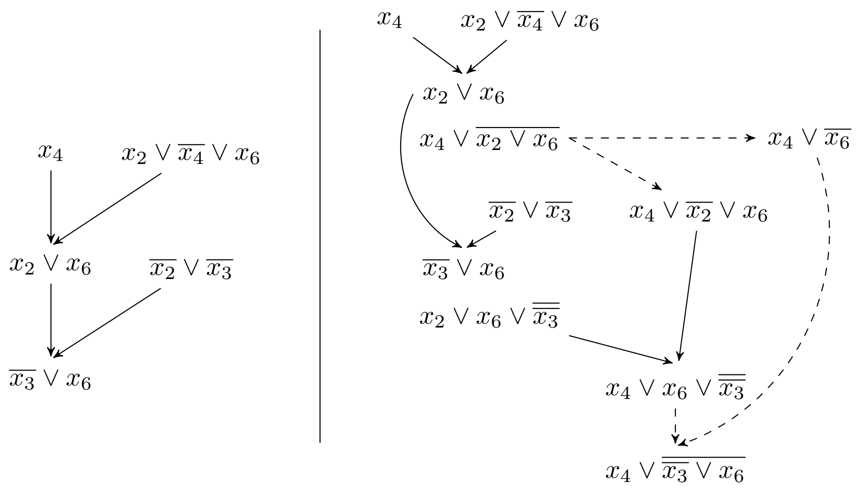

x_4
$x_2 \vee \overline{x_4} \vee x_6$
$x_2 \vee x_6$
$\overline{x_2} \vee \overline{x_3}$
$\overline{x_3} \vee x_6$
x_4
$x_2 \vee \overline{x_4} \vee x_6$
$x_2 \vee x_6$
$x_4 \vee \overline{x_2 \vee x_6}$
$x_4 \vee \overline{x_6}$
$\overline{x_2} \vee \overline{x_3}$
$x_4 \vee \overline{x_2} \vee x_6$
$\overline{x_3} \vee x_6$
$x_2 \vee x_6 \vee \overline{\overline{x_3}}$
$x_4 \vee x_6 \vee \overline{\overline{x_3}}$
$x_4 \vee \overline{\overline{x_3} \vee x_6}$

Figure 3 Dragging the non read-once clause while unfolding a read-once linear section of the proof. Solid lines represent the application of the Max- SAT resolution rule whereas dashed lines represent compaction or expansion. Unused compensation clauses are omitted.

- Suppose that we can generate $C'_{i-1} = C_1 \vee \overline{C_i}$ at the $i^{th} - 1$ Max-SAT resolution step. The i^{th} step on $C_i = l_i \vee A_i$ and $D_i = \overline{l_i} \vee B_i$ w.r.t $var(l_i)$ generates the resolvent $C_{i+1} = A_i \vee B_i$ and the compensation clauses $CC_{1|i} = l_i \vee A_i \vee \overline{B_i}$ and $CC_{2|i} = \overline{l_i} \vee \overline{A_i} \vee B$. The induction step to infer C'_i is represented in Figure 2. Note that similarly to the base case, if $D_i = \overline{l_i}(i > 1)$ is a unit clause, i.e., the i^{th} step corresponds to a deletion of literal l_i from $C_i = l_i \vee A_i$ deducing the resolvent $C_{i+1} = A_i$, the tautological clauses $l_i \vee A_i \vee \overline{A_i}$ can be added to the formula thus replacing $CC_{1|i}$ in Figure 2. However, as showcased in the same figure, the addition of such clauses in case D_1 is unit can be avoided since the initial expansion step on C'_{i-1} suffices to generate $C_1 \vee \overline{A_i} = C_1 \vee \overline{C_{i+1}} = C'_i$.

Finally, by Proposition 11 (case b.), the inference of $C_1 \vee A_i \vee \overline{B_i}$ in Figure 2 requires $O(|B_i|)$ Max-SAT resolution steps and, thus, every step in π is clearly adapted in $O(w(\pi))$ inference steps to generate C and $C_1 \vee \overline{C_m}$. Therefore, we conclude that we can deduce $\phi \vdash_{MaxRes} C \wedge (C_1 \vee \overline{C_m})$ in $O(s(\pi).w(\pi))$ inference steps. ◀

▶ **Example 13.** We consider the read-once linear derivation of clause $\overline{x_3} \vee x_6$ from $\phi = \{x_4,\ x_2 \vee \overline{x_4} \vee x_6,\ \overline{x_2} \vee \overline{x_3}\}$ represented on the left of Figure 3. The Max-SAT resolution proof deducing $\overline{x_3} \vee x_6$ and $x_4 \vee \overline{\overline{x_3} \vee x_6}$ is represented on the right of Figure 3.

Next, we establish our main result on the adaptation of crossing-free resolution derivations. The proof in the following theorem particularly deals with the junction nodes in ensuing derivations, i.e., nodes where the paths starting from the non read-once clauses intersect. More specifically, we want to drag or bring along the non read-once clause through these particular nodes. We provide an illustration of a full adaptation in Example 15.

▶ **Theorem 14.** *Let ϕ be a CNF formula and π be a crossing-free resolution derivation of clause C from ϕ. We can deduce $\phi \vdash_{MaxRes} C$ in $O(s(\pi) \times (s(\pi) + w(\pi))^2)$ inference steps.*

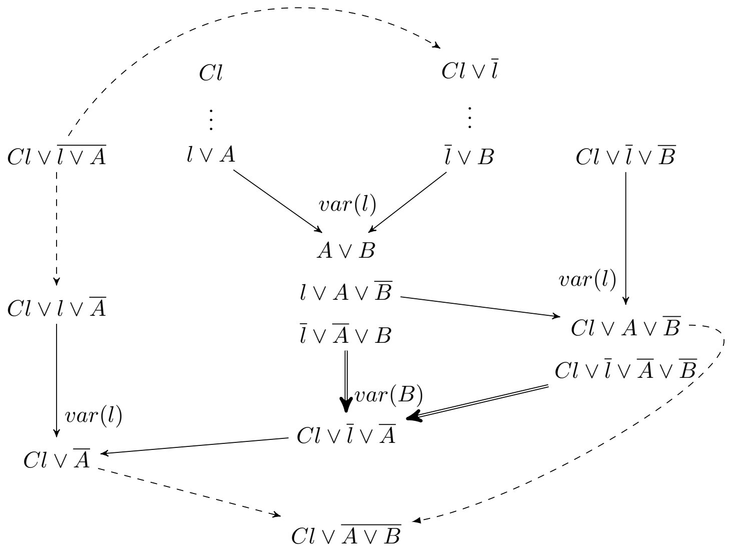

Cl
$Cl \vee \bar{l}$
$Cl \vee \overline{l \vee A}$
$l \vee A$
$\bar{l} \vee B$
$Cl \vee \bar{l} \vee \overline{B}$
$var(l)$
$A \vee B$
$l \vee A \vee \overline{B}$
$var(l)$
$Cl \vee l \vee \overline{A}$
$Cl \vee A \vee \overline{B}$
$\bar{l} \vee \overline{A} \vee B$
$Cl \vee \bar{l} \vee \overline{A} \vee \overline{B}$
$var(B)$
$var(l)$
$Cl \vee \bar{l} \vee \overline{A}$
$Cl \vee \overline{A}$
$Cl \vee \overline{A \vee B}$

Figure 4 Inferring $Cl \vee \overline{A \vee B}$ in a junction node of $ED(Cl)$. Solid lines represent the application of the Max-SAT resolution rule, bold double arcs represent the application of Max-SAT resolution to delete opposed sets of literals and dashed lines represent compaction or expansion. Unused compensation clauses are omitted.

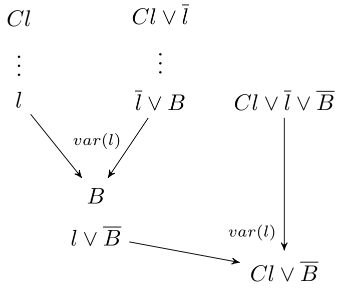


Figure 5 Inferring $Cl \vee \overline{C'}$ in case A is empty in a junction node of $ED(Cl)$. Solid lines represent the application of the Max-SAT resolution rule whereas dashed lines represent compaction and expansion. Unused compensation clauses are omitted.

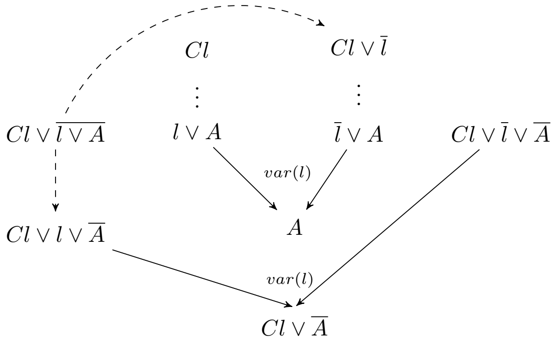

Cl
$Cl \vee \bar{l}$
$Cl \vee \overline{l \vee A}$
$l \vee A$
$\bar{l} \vee A$
$Cl \vee \bar{l} \vee \overline{A}$
$var(l)$
$Cl \vee l \vee \overline{A}$
A
$var(l)$
$Cl \vee \overline{A}$

Figure 6 Inferring $Cl \vee \overline{C'}$ in case $A = B$ in a junction node of $ED(Cl)$. Solid lines represent the application of the Max-SAT resolution rule whereas dashed lines represent compaction and expansion. Unused compensation clauses are omitted.

Proof. Property 10 ensures that each ensuing derivation can be adapted independently. Let Cl be a non read-once clause in π and w.l.o.g we only consider it ensuing derivation $ED(Cl)$. We prove that at each step of $ED(Cl)$ deriving clause C', we can infer C' and $SC \vee \overline{C'}$ where SC is either Cl or its substitute in the path leading to C'. The proof is by induction on the size of the derivation. The base case where the derivation is empty is trivial. Next, using Lemma 12, we can suppose w.l.o.g that C' is derived in a junction node (of paths starting from Cl). Let $l \vee A$ and $\bar{l} \vee B$ be the premises of the resolution step deriving $C' = A \vee B$.

The induction hypothesis ensures that there exists a Max-SAT resolution derivation of $l \vee A$ and $C \vee \overline{l \vee A}$. As showcased in Figure 4, $Cl \vee \bar{l}$ can be used to replace the occurrences of Cl in the derivation of $\bar{l} \vee B$. Note that to avoid using tautological substitutes, we can suppose w.l.o.g that $l \notin Cl$ by interchanging the proofs of $l \vee A$ and $l \vee B$ when necessary thus entailing a different unfolding order of the original proof and the generation of the exact same clause as a substitute in such nodes. Again, similarly to the left side, the induction hypothesis ensures the existence of a Max-SAT resolution derivation of $\bar{l} \vee B$ and $Cl \vee \bar{l} \vee \overline{\bar{l} \vee B}$ and, therefore, $Cl \vee \bar{l} \vee \overline{B}$ by expansion. Clearly, $C' = A \vee B$ can be derived by Max-SAT resolution and we showcase in Figure 4 how $Cl \vee \overline{C'} = C \vee \overline{A \vee B}$ can be inferred using the compensation clauses as well as $Cl \vee \overline{l \vee A}$ and $Cl \vee \bar{l} \vee \overline{B}$.
Note that the following particular cases can occur:

- A or B is empty, in which case a unit clause is used to derive $C' = A \vee B$. We represent in Figure 5 how to derive $Cl \vee \overline{C'}$ in case A is empty. The derivation in case B is empty is symmetric and thus omitted. Notice that in the case both A and B are empty, π is a refutation and there is no need to derive $Cl \vee \overline{C'}$ in the last Max-SAT resolution step. In fact, more generally, this is also not necessary for the last junction node in an ensuing derivation in π.
- $A = B$ in which case the generated compensation clauses are tautological and are not necessary to derive $Cl \vee \overline{C'} = Cl \vee \overline{A}$ as showcased in Figure 6.

Finally, in each junction node we need $O(|B|)$ inference steps to deduce $Cl \vee A \vee \overline{B}$ using case (c) in Proposition 11. Similarly, using expansion on A and pattern (b) in Proposition 11, we need $O(|A| \times |B|)$ inference steps to deduce $Cl \vee \bar{l} \vee \overline{A}$. It is important to note that the width of the proof may evolve while generating substitutes for non read-once clauses as literals may be added in junction nodes. However, the width remains bounded by $w(\pi) + s(\pi)$ and thus each junction node can be adapted in $O((w(\pi) + s(\pi))^2)$ inference steps. Therefore, we conclude that we can deduce $\phi \vdash_{MaxRes} C$ in $O(s(\pi) \times (s(\pi) + w(\pi))^2)$ inference steps. ◀

▶ **Example 15.** We consider the formula $\phi = \{\overline{x_1} \vee x_3 \vee \overline{x_4},\ x_4 \vee x_5,\ x_4 \vee \overline{x_5},\ x_1 \vee \overline{x_4},\ x_2 \vee \overline{x_4} \vee x_6,\ \overline{x_2} \vee \overline{x_3}\}$ and the derivation π of clause x_6 from ϕ represented in Figure 1. We omit the section of the proof (in blue) deriving clause $\overline{x_2} \vee \overline{x_3}$ for simplicity. Note that this omitted part, i.e., the ensuing derivation of the non read-once clause $\overline{x_2} \vee \overline{x_3} \vee \overline{x_7}$ corresponds to a diamond pattern [24]. Such patterns will be studied in Section 5 (refer to Example 23 for the adaptation). The adaptation of proof π is reported in Figure 7. We reuse the adaptation of the linear read-once section in Example 3. The non read-once clause and its substitutes are colored in red and added tautological clauses are represented in green. Note that this is one of the possible adaptations depending on the order chosen for adapting the branches of $ED(x_4)$. Finally, we stress the fact that we could have generated the clause $C = x_4 \vee \overline{x_6}$ after the last Max-SAT resolution step on clauses $\overline{x_3} \vee x_3$ (but we omit this inference since $x_6 = EC(x_4)$ as mentioned in the proof of Theorem 14). Indeed, C can be inferred by an additional Max-SAT resolution step on the compensation clauses obtained in the last step, i.e., clauses $x_3 \vee \overline{x_6}$ and $x_4 \vee \overline{x_3} \vee \overline{x_6}$. In the proof of Theorem 14, this corresponds to the case where B is empty in a junction node of an ensuing derivation.

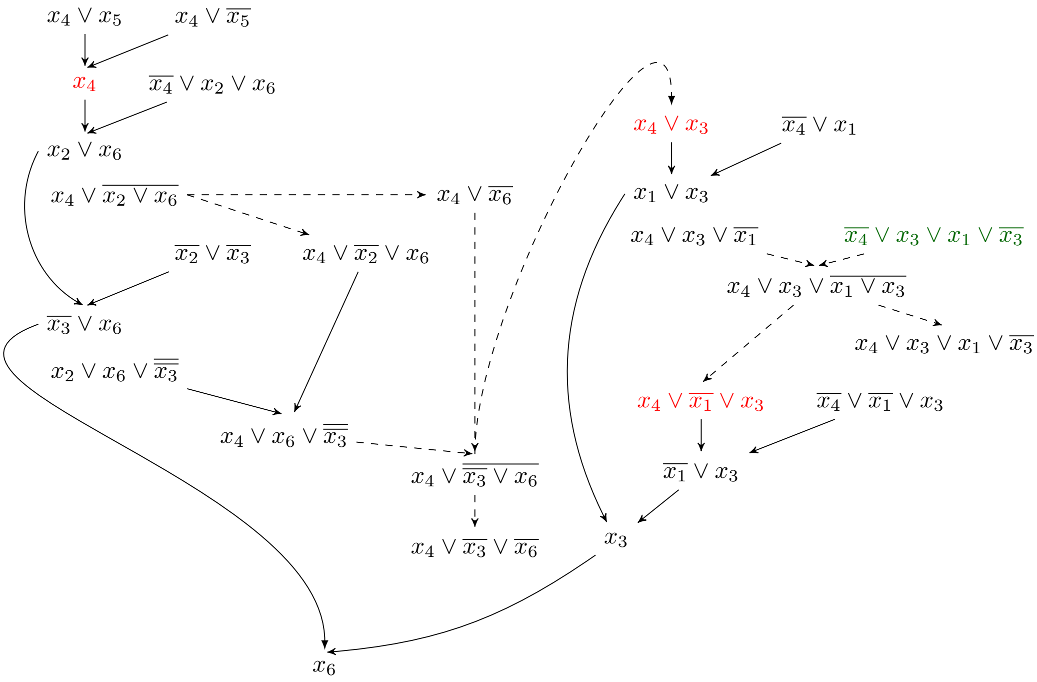

$x_4 \vee x_5$
$x_4 \vee \overline{x_5}$
x_4
$\overline{x_4} \vee x_2 \vee x_6$
$x_2 \vee x_6$
$x_4 \vee \overline{x_2 \vee x_6}$
$x_4 \vee \overline{x_6}$
$\overline{x_2} \vee \overline{x_3}$
$x_4 \vee \overline{x_2} \vee x_6$
$\overline{x_3} \vee x_6$
$x_2 \vee x_6 \vee \overline{\overline{x_3}}$
$x_4 \vee x_6 \vee \overline{\overline{x_3}}$
$x_4 \vee \overline{\overline{x_3} \vee x_6}$
$x_4 \vee \overline{x_3} \vee \overline{x_6}$
x_6
$x_4 \vee x_3$
$\overline{x_4} \vee x_1$
$x_1 \vee x_3$
$x_4 \vee x_3 \vee \overline{x_1}$
$\overline{x_4} \vee x_3 \vee x_1 \vee \overline{x_3}$
$x_4 \vee x_3 \vee \overline{x_1 \vee x_3}$
$x_4 \vee x_3 \vee x_1 \vee \overline{x_3}$
$x_4 \vee \overline{x_1} \vee x_3$
$\overline{x_4} \vee \overline{x_1} \vee x_3$
$\overline{x_1} \vee x_3$
x_3

Figure 7 Adaptation of a crossing-free resolution derivation. Solid lines represent the application of the Max-SAT resolution rule whereas dashed lines represent compaction and expansion. Unused compensation clauses are omitted.

Next, we discuss some minor syntactic subtleties that occur in the adaptation. First, it is important to note that the use of the expansion and compaction rewritings as full fledged rules is relevant for simplification but not necessary. Recall that these two rules are mainly used in order to switch between the different equivalent forms of $\overline{C}$ when it is written in CNF form. Each form corresponds to a different ordering of the literals in C. When applying Max-SAT resolution, a relevant order may be chosen when necessary. However, an application of a compaction followed by an expansion may correspond to a certain rearrangement of the variables in CNF form. This may occur when adapting the read-once linear part of the proof. Indeed, as showcased in Figure 2, a compaction may be followed by an expansion to isolate the clause $C_1 \vee \overline{l_i} \vee A_i$ from the compact form $C_1 \vee \overline{A_i}$. Similarly, as shown in Figure 4, it may be necessary to isolate the clause $Cl \vee \overline{l}$ from the compact form $Cl \vee l \vee \overline{A}$ when dealing with junction nodes.

More specifically, we may need to rearrange a certain literal at the beginning or at the end of the ordering. In Proposition 16, we prove that it is possible to switch the first and last literals in the CNF form of $\overline{C}$ in $O(|C|)$ inference steps. This entails that in the proof of Theorem 14, the compaction and expansion rules can be omitted and replaced with $O(s(\pi) \times (s(\pi) + w(\pi)))$ Max-SAT resolutions. Clearly, this does not impact our result in terms of the size of the resulting adaptation. In Example 17, we provide the full simplified adaptation of the proof in Example 15 without the use of rewriting rules.

▶ **Proposition 16.** *Let n be a natural number and $l_1, ..., l_n$ be n literals. We can deduce $(\overline{l_1}) \wedge (l_1 \vee \overline{l_2}) \wedge ... \wedge (l_1 \vee ... \vee l_{n-1} \vee \overline{l_n}) \vdash_{MaxRes} (\overline{l_n}) \wedge (l_n \vee \overline{l_2}) \wedge ... \wedge (l_n \vee l_2 \vee ... \vee l_{n-1} \vee \overline{l_1})$ in $O(n)$ inference steps.*

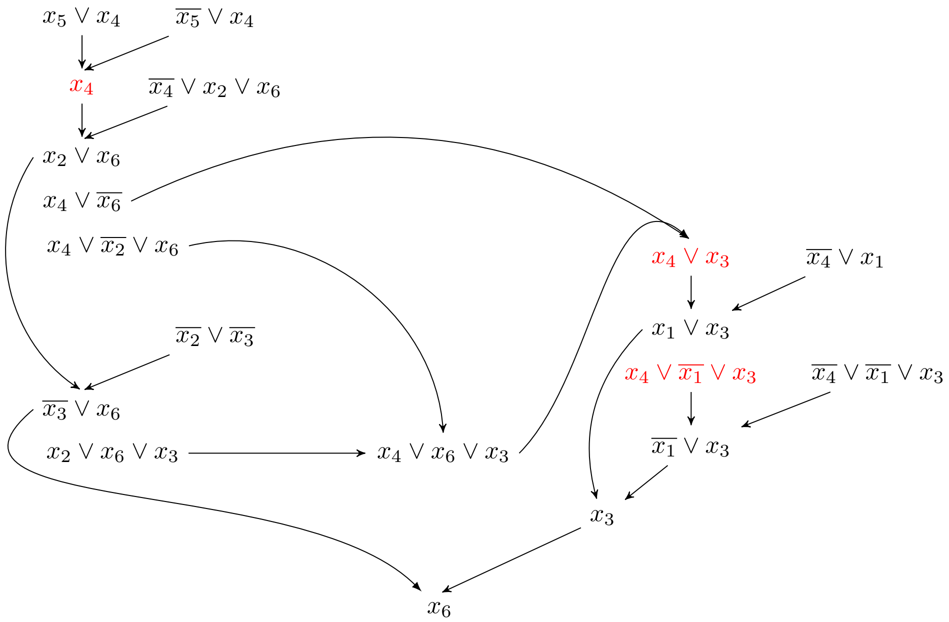

$x_5 \vee x_4$
$\overline{x_5} \vee x_4$
x_4
$\overline{x_4} \vee x_2 \vee x_6$
$x_2 \vee x_6$
$x_4 \vee \overline{x_6}$
$x_4 \vee \overline{x_2} \vee x_6$
$x_4 \vee x_3$
$\overline{x_4} \vee x_1$
$x_1 \vee x_3$
$\overline{x_2} \vee \overline{x_3}$
$x_4 \vee \overline{x_1} \vee x_3$
$\overline{x_4} \vee \overline{x_1} \vee x_3$
$\overline{x_3} \vee x_6$
$x_2 \vee x_6 \vee x_3$
$x_4 \vee x_6 \vee x_3$
$\overline{x_1} \vee x_3$
x_3
x_6

Figure 8 Adaptation of a crossing-free resolution proof to a Max-SAT resolution proof. Unused compensation clauses are omitted.

Proof. By induction on n we have:

- If $n = 1$ the result is trivial.
- For $n > 1$, the application of Max-SAT resolution on clauses $l_1 \vee .... \vee l_{n-2} \vee \overline{l_{n-1}}$ and $l_1 \vee .... \vee l_{n-1} \vee \overline{l_n}$ w.r.t $var(l_{n-1})$ generates the resolvent clause $C = l_1 \vee .... \vee l_{n-2} \vee \overline{l_n}$ and the compensation clause $CC = l_1 \vee .... \vee l_{n-2} \vee l_n \vee \overline{l_{n-1}}$. Furthermore, by induction, we can deduce $(\overline{l_1}) \wedge (l_1 \vee \overline{l_2}) \wedge ... \wedge (l_1 \vee ... \vee l_{n-2} \vee \overline{l_n}) \vdash_{MaxRes} (\overline{l_n}) \wedge (l_n \vee \overline{l_2}) \wedge ... \wedge (l_n \vee l_2 \vee ... \vee l_{n-2} \vee \overline{l_1})$ in $O(n-1)$ inference steps. A single additional Max-SAT resolution step on clauses CC and $l_n \vee l_2 \vee ... \vee l_{n-2} \vee \overline{l_1}$ w.r.t $var(l_1)$ is sufficient to generate the resolvent clause $l_n \vee l_2 \vee ... \vee l_{n-2} \vee \overline{l_{n-1}}$ and the compensation clause $l_n \vee l_2 \vee ... \vee l_{n-1} \vee \overline{l_1}$. Therefore, we deduce the wanted result in $O(n)$ inference steps. ◀

▶ **Example 17.** We consider the same formula ϕ in Example 15. We represent in Figure 8 a Max-SAT resolution proof (without rewriting) of clause x_6 from ϕ. Notice how we use the following rearrangement $\overline{x_6} \wedge (x_6 \vee x_3) \vdash_{MaxRes} x_3 \wedge (x_3 \vee x_6)$ to generate the substitute $x_4 \vee x_3$. Furthermore, the tautological clause $\overline{x_4} \vee x_3 \vee x_1 \vee \overline{x_3}$ colored in green in Figure 7 and the rearrangement in which it is involved are not necessary since the last required substitute for x_1, i.e $x_4 \vee \overline{x_1} \vee x_3$ is naturally generated by the preceding Max-SAT resolution step. Therefore, they can be deleted as is the case for the full adaptation without rewriting in Figure 8.

Next, we discuss the implications of the equality $\overset{*}{=}$ used in the proof of Lemma 12, i.e., $l \vee A \vee \overline{B} \overset{*}{=} l \vee A \vee \overline{A \vee B}$. Recall that this equality is sound for Max-SAT (c.f. Remark 13 in [17]). However, to avoid adding it as a standalone rule and as explained in the proof of Lemma 12, we can consider the addition of tautological clauses. This may also be required in case of unit clauses. It is important to note that the number of tautological clauses added to the formula in an adaptation of a crossing-free resolution derivation π is in $O(s(\pi) \times (w(\pi) + s(\pi)))$. A similar phenomenon was also noted in [8]. In addition, notice how the adaptation may also rely on tautological compensation clauses which are generated by Max-SAT resolution. Such clauses are usually deleted or omitted in the literature [6, 7, 17] but they may carry important information which is necessary to infer substitutes for non read-once clauses.

Finally, we establish our result on crossing-free refutations in the following corollary. We also illustrate in Example 19 an adaptation of a crossing free resolution refutation to a Max-SAT resolution refutation.

▶ **Corollary 18.** *Let ϕ be an unsatisfiable CNF formula and π be a crossing-free resolution refutation of ϕ. We can deduce $\phi \vdash_{MaxRes} \square$ from ϕ in $O(s(\pi)^3)$ inference steps.*

Proof. Trivially entailed from Theorem 14 since $w(\pi) = O(s(\pi))$ for refutations. ◀

▶ **Example 19.** We consider the unsatisfiable CNF formula $\phi = \{x_1, \overline{x_1} \vee x_3, \overline{x_1} \vee x_2, \overline{x_2} \vee \overline{x_3}\}$ and the refutation π of ϕ represented in Figure 9. Clearly, π is crossing-free since there is only one non read-once clause, i.e., x_1. In fact, π also corresponds to the ensuing derivation of x_1 and $\square$ is its ensued clause, i.e., $ED(x_1) = \pi$ and $EC(x_1) = \square$. Two possible adaptations of π are illustrated in Figure 10. The non read-once clause and its substitutes are colored in red. The possible adaptations correspond to different possible orderings of the proof. In the adaptation on the left, we consider that the resolution step on clauses $\overline{x_1} \vee x_1$ and x_1 precedes the one on clauses x_1 and $\overline{x_1} \vee x_2$, and inversely for the adaptation on the right. Note that the adaptation on the left corresponds to the handmade example provided by Bonet et al. in [6, 7] (c.f. Example 1 in [6] or Example 3 in [7]).

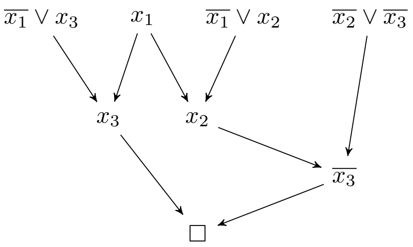

$\overline{x_1} \vee x_3$
x_1
$\overline{x_1} \vee x_2$
$\overline{x_2} \vee \overline{x_3}$
x_3
x_2
$\overline{x_3}$
$\square$

Figure 9 Crossing-free resolution refutation.

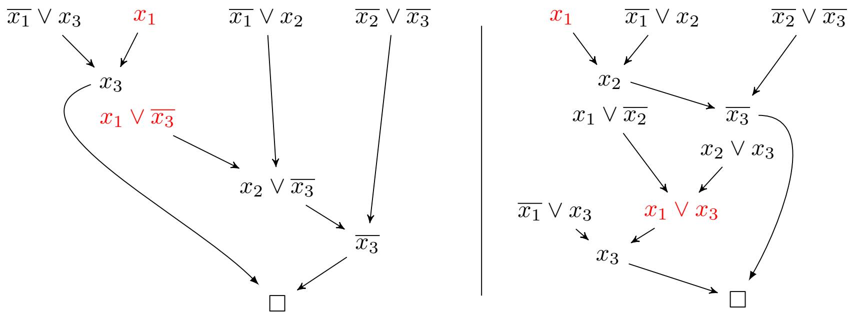

$\overline{x_1} \vee x_3$
x_1
$\overline{x_1} \vee x_2$
$\overline{x_2} \vee \overline{x_3}$
x_3
$x_1 \vee \overline{x_3}$
$x_2 \vee \overline{x_3}$
$\overline{x_3}$
$\square$
x_1
$\overline{x_1} \vee x_2$
$\overline{x_2} \vee \overline{x_3}$
x_2
$\overline{x_3}$
$x_1 \vee \overline{x_2}$
$x_2 \vee x_3$
$\overline{x_1} \vee x_3$
$x_1 \vee x_3$
x_3
$\square$

Figure 10 Two possible adaptations of the crossing-free resolution refutation represented in Figure 9 depending on the ordering of the resolution steps involving the non read-once clause x_1. Unused compensation clauses are omitted.

5 On (k-stacked) Diamond Patterns

In this section, we study particular resolution refutations, called k-stacked diamond patterns, which were introduced and shown exponential for the adaptation (to ResS) in [24]. A diamond pattern (x, y, A) where $x, y \notin A$ is the sequence of resolutions represented in Figure 11. Note that the particular diamond pattern $(x, y, \Box)$ is a resolution refutation. Now, imagine that the topmost clause of $(x, y, \Box)$ is derived through another diamond pattern. We iterate the same reasoning to define a k-stacked diamonds pattern as in Definition 20.

▶ **Definition 20** (k-stacked diamond)**.** *Let $k \geq 1$ be a natural number and let x_i and y_i where $1 \leq i \leq k$ be distinct variables. A k-stacked diamond pattern is formed by k diamond patterns (x_i, y_i, A_i) where $1 \leq i \leq k$ such that $A_1 = \Box$ and $A_i = (x_1 \vee \cdots \vee x_{i-1})$ for $1 < i \leq k$. Each diamond (x_i, y_i, A_i) is stacked on top of $(x_{i-1}, y_{i-1}, A_{i-1})$ such that the last conclusion of the former is the topmost central premise of the latter.*

When $k > 2$, the size of a k-stacked diamond P is $s(P) = 3k$ while the size of the computed refutation in ResS [18], i.e., Max-SAT resolution augmented with the split rule, by the adaptation in [24] is at least 2^{k-1} which is exponential in the size of P. First, notice that k-stacked diamond patterns fall within the crossing-free resolution. Furthermore, these patterns can be adapted to Max-SAT resolution refutations without increasing their size as established in 22. Such an adaptation is illustrated in Example 23.

▶ **Proposition 21.** *Let $k \geq 1$ be a natural number. A k-stacked diamond resolution refutation is crossing-free.*

▶ **Proposition 22.** *Let ϕ be a CNF formula, $k \geq 1$ be a natural number and π be a k-stacked diamond resolution refutation. There exists a Max-SAT resolution refutation π' of ϕ s.t $s(\pi') \leq s(\pi)$.*

Proof. We show how to adapt every diamond pattern without increasing its size In Figure 12. This is entailed by the fact that each diamond is clearly a crossing-free derivation and more specifically an ensuing derivation of a non read-once clause. As such, a k-stacked diamond P can be adapted in at most $s(P)$ Max-SAT resolution steps. ◀

▶ **Example 23.** We consider the ensuing derivation of clause $\overline{x_2} \vee \overline{x_3} \vee x_7$ represented in Figure 1. As mentioned in Example 15, this part of the proof corresponds to a diamond pattern. Its adaptation is illustrated in Figure 13 (the non read-once clause and its substitute are represented in red). The adaptation can be added on top of clause $\overline{x_2} \vee \overline{x_3}$ in Figure 8 to obtain the full adaptation of the initial crossing-free proof represented in Figure 1.

6 Conclusion

In this paper, we introduced a new fragment of resolution, called crossing-free resolution, in which ensuing derivations of non read-once clauses are disjoint. We showed that crossing-free

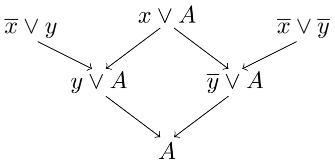


Figure 11 Diamond pattern (x, y, A).

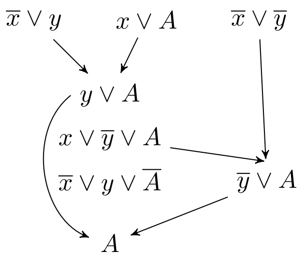

$\overline{x} \lor y$
$x \lor A$
$\overline{x} \lor \overline{y}$
$y \lor A$
$x \lor \overline{y} \lor A$
$\overline{x} \lor y \lor \overline{A}$
$\overline{y} \lor A$
A

Figure 12 Adaptation of a diamond pattern (x, y, A).

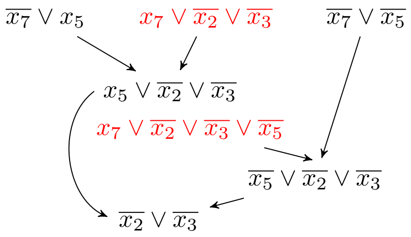

$\overline{x_7} \lor x_5$
$x_7 \lor \overline{x_2} \lor \overline{x_3}$
$\overline{x_7} \lor \overline{x_5}$
$x_5 \lor \overline{x_2} \lor \overline{x_3}$
$x_7 \lor \overline{x_2} \lor \overline{x_3} \lor \overline{x_5}$
$\overline{x_5} \lor \overline{x_2} \lor \overline{x_3}$
$\overline{x_2} \lor \overline{x_3}$

Figure 13 Adaptation of the diamond pattern in the resolution proof represented in Figure 1 (colored in blue) corresponding to the ensuing derivation of clause $x_7 \lor \overline{x_2} \lor \overline{x_3}$. Unused compensation clauses are omitted.

resolution derivations and in particular crossing-free refutations can be adapted to Max-SAT resolution proofs without substantially increasing their size. To the best of our knowledge, this is the first non trivial fragment, i.e., different from read-once resolution, whose adaptation is shown possible using only Max-SAT resolution with a reasonable guarantee on the size of the adapted proofs. The idea behind the adaptation is to naturally infer substitutes for non read-once clauses by dragging them along while unfolding the initial resolution proof and by relying on compensation clauses produced by Max-SAT resolution. Furthermore, we show that diamond patterns, which were shown exponential for the adaptation in [24], fall within the crossing-free resolution fragment and can be adapted into Max-SAT resolution proofs without increasing their size.

Our results contribute to the difficult open problem of adapting resolution proofs to Max-SAT resolution proofs without increasing their size [6, 7] and, therefore, helps to bridge the gap between resolution for SAT and Max-SAT. Furthermore, unlike SAT solvers, Max-SAT solvers are still not able to output certificates in the form of Max-SAT equivalent proofs mainly due to the variety of solving paradigms and due to the theoretical gap between SAT and Max-SAT resolution. Our work can be useful in this regard and particularly in improving the efficiency of independent proof builders for the Max-SAT problem [25]. Finally, as future work, it would be interesting to characterize a larger intersection between SAT and Max-SAT resolution by proving that an adaptation of an extended refinement of resolution (ideally unrestricted resolution) without a substantial increase in the size of the proofs is possible, even through augmenting Max-SAT resolution by other inference rules.

References

1 André Abramé and Djamal Habet. Ahmaxsat: Description and Evaluation of a Branch and Bound Max-SAT Solver. *J. Satisf. Boolean Model. Comput.*, 9(1):89–128, 2014. doi:10.3233/sat190104.

2 Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. SAT-based MaxSAT algorithms. *Artificial Intelligence*, 196:77–105, 2013. doi:10.1016/j.artint.2013.01.002.

3 Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins, editors. *MaxSAT Evaluation 2021: Solver and Benchmark Descriptions.* Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, Finland, 2021. URL: https://maxsat-evaluations.github.io/2021/.

4 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. *Maximum Satisfiability*, pages 929–991. Frontiers in Artificial Intelligence and Applications. IOS PRESS, Netherlands, 2 edition, 2021. doi:10.3233/FAIA201008.

5 Maria Luisa Bonet and Jordi Levy. Equivalence Between Systems Stronger Than Resolution. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 166–181. Springer, 2020. doi:10.1007/978-3-030-51825-7_13.

6 Maria Luisa Bonet, Jordi Levy, and Felip Manyà. A Complete Calculus for Max-SAT. 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 240–251. Springer, 2006. doi:10.1007/11814948_24.

7 Maria Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for Max-SAT. *Artif. Intell.*, 171(8-9):606–618, 2007. doi:10.1016/j.artint.2007.03.001.

8 Joshua Buresh-Oppenheim and Toniann Pitassi. The Complexity of Resolution Refinements. *J. Symb. Log.*, 72(4):1336–1352, 2007. doi:10.2178/jsl/1203350790.

9 Sam Buss and Jan Johannsen. On Linear Resolution. *J. Satisf. Boolean Model. Comput.*, 10(1):23–35, 2016. doi:10.3233/sat190112.

10 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.

11 Yuval Filmus, Meena Mahajan, Gaurav Sood, and Marc Vinyals. MaxSAT Resolution and Subcube Sums. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 295–311. Springer, 2020. doi:10.1007/978-3-030-51825-7_21.

12 Federico Heras and João Marques-Silva. Read-Once Resolution for Unsatisfiability-Based Max-SAT Algorithms. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 572–577. IJCAI/AAAI, 2011. doi:10.5591/978-1-57735-516-8/IJCAI11-103.

13 Kazuo Iwama and Eiji Miyano. Intractability of Read-Once Resolution. In *Proceedings of the Tenth Annual Structure in Complexity Theory Conference, Minneapolis, Minnesota, USA, June 19-22, 1995*, pages 29–36. IEEE Computer Society, 1995. doi:10.1109/SCT.1995.514725.

14 Adrian Kügel. Improved Exact Solver for the Weighted MAX-SAT Problem. In Daniel Le Berre, editor, *POS-10. Pragmatics of SAT, Edinburgh, UK, July 10, 2010*, volume 8 of *EPiC Series in Computing*, pages 15–27. EasyChair, 2010. doi:10.29007/38lm.

15 Sukhamay Kundu. Tree resolution and generalized semantic tree. In Zbigniew W. Ras and Maria Zemankova, editors, *Proceedings of the ACM SIGART International Symposium on Methodologies for Intelligent Systems, ISMIS 1986, Knoxville, Tennessee, USA, October 22-24, 1986*, pages 270–278. ACM, 1986. doi:10.1145/12808.12838.

16 Javier Larrosa and Federico Heras. Resolution in Max-SAT and its relation to local consistency in weighted CSPs. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 193–198. Professional Book Center, 2005. URL: http://ijcai.org/Proceedings/05/Papers/0360.pdf.

17 Javier Larrosa, Federico Heras, and Simon de Givry. A logical approach to efficient Max-SAT solving. *Artif. Intell.*, 172(2-3):204–233, 2008. doi:10.1016/j.artint.2007.05.006.

18 Javier Larrosa and Emma Rollon. Towards a Better Understanding of (Partial Weighted) MaxSAT Proof Systems. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2020. doi:10.1007/978-3-030-51825-7_16.

19 Chu Min Li, Felip Manyà, and Jordi Planes. New Inference Rules for Max-SAT. *J. Artif. Intell. Res.*, 30:321–359, 2007. doi:10.1613/jair.2215.

20 Chu Min Li, Felip Manyà, and Joan Ramon Soler. A Clause Tableau Calculus for MaxSAT. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 766–772. IJCAI/AAAI Press, 2016. URL: http://www.ijcai.org/Abstract/16/114.

21 Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Combining Clause Learning and Branch and Bound for MaxSAT. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 38:1–38:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CP.2021.38.

22 D. W. Loveland. A linear format for resolution. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Symposium on Automatic Demonstration*, pages 147–162, Berlin, Heidelberg, 1970. Springer Berlin Heidelberg. doi:10.1007/BFb0060630.

23 Nina Narodytska and Fahiem Bacchus. Maximum Satisfiability Using Core-Guided MaxSAT Resolution. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 2717–2723. AAAI Press, 2014. URL: https://ojs.aaai.org/index.php/AAAI/article/view/9124.

24 Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. Towards Bridging the Gap Between SAT and Max-SAT Refutations. In *32nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI*, pages 137–144. IEEE, 2020. doi:10.1109/ICTAI50040.2020.00032.

25 Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. A Proof Builder for Max-SAT. 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 488–498. Springer, 2021. doi:10.1007/978-3-030-80223-3_33.

26 Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. Computing Max-SAT Refutations using SAT Oracles. In *33rd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2021, Washington, DC, USA, November 1-3, 2021*, pages 404–411. IEEE, 2021. doi:10.1109/ICTAI52525.2021.00066.

27 Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. Inferring Clauses and Formulas in Max-SAT. In *33rd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2021, Washington, DC, USA, November 1-3, 2021*, pages 632–639. IEEE, 2021. doi:10.1109/ICTAI52525.2021.00101.

28 John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965. doi:10.1145/321250.321253.

Isomorphisms Between STRIPS Problems and Sub-Problems

Martin C. Cooper ✉
IRIT, University of Toulouse, France

Arnaud Lequen ✉
IRIT, University of Toulouse, France

Frédéric Maris ✉
IRIT, University of Toulouse, France

Abstract

Determining whether two STRIPS planning instances are isomorphic is the simplest form of comparison between planning instances. It is also a particular case of the problem concerned with finding an isomorphism between a planning instance P and a sub-instance of another instance P'. One application of such an isomorphism is to efficiently produce a compiled form containing all solutions to P from a compiled form containing all solutions to P'. In this paper, we study the complexity of both problems. We show that the former is GI-complete, and can thus be solved, in theory, in quasi-polynomial time. While we prove the latter to be NP-complete, we propose an algorithm to build an isomorphism, when possible. We report extensive experimental trials on benchmark problems which demonstrate conclusively that applying constraint propagation in preprocessing can greatly improve the efficiency of a SAT solver.

2012 ACM Subject Classification Computing methodologies → Planning for deterministic actions; Mathematics of computing → Matchings and factors

Keywords and phrases planning, isomorphism, complexity, constraint propagation

Digital Object Identifier 10.4230/LIPIcs.CP.2022.13

Supplementary Material The material to reproduce the experiments can be found here:
Software (Source Code): `https://github.com/arnaudlequen/PDDLIsomorphismFinder`

Funding This work was supported by the AI Interdisciplinary Institute ANITI, funded by the French program "Investing for the Future – PIA3" under Grant agreement no. ANR-19-PI3A-0004.

Acknowledgements The authors would like to thank the anonymous reviewers, whose insightful comments helped improve this paper.

1 Introduction

Models used for STRIPS [7] planning encode sizeable state-spaces that can rarely be represented explicitly, but that have a clear and somewhat regular structure. Parts of this structure can be, however, common to multiple planning instances, although this similarity is often far from immediate to identify by looking at the STRIPS representation. Indeed, finding whether or not an instance P is a sub-instance of another problem P' requires to map every fluent and every operator of P to their counterpart in P', while respecting a morphism property. This requires the exploration of the exponential search space of mappings from P to P'. Finding such a mapping, however, allows us to carry over significant pieces of information from one problem to the other. In particular, any solution-plan for P can then be translated into a plan for P' efficiently.

A classical technique in constraint programming is to store all solutions to a CSP or SAT instance in a compact compiled form [1]. This is performed off-line. A compilation map indicates which operations and transformations can be performed in polynomial time during the on-line stage [6]. STRIPS fixed-horizon planning can be coded as a SAT instance using the classical SATPLAN encoding [9]. So, for a given instance, all plans can be stored in a compiled form, at least in theory. In practice, the compiled form will often be too large to be stored. Types of planning problems which are nevertheless amenable to compilation are those where the number of plans is small or, at the other extreme, there are few constraints on the order of operators. If we have a compiled form C' representing all solution-plans to an instance P' and we encounter a similar problem P, it is natural to ask whether we can synthesize a plan for P from C'. If P is isomorphic to a subproblem of P', then it suffices to apply a sequence of conditioning operations to C' to obtain a compiled form C representing all solutions to P. This is our main motivation for studying isomorphisms between subproblems. A trivial but important special case occurs when C' is empty, i.e. P' has no solution. In this case, an isomorphism from P' to a subproblem of P is a proof that P' has no solution.

In this paper, we first focus on problem SI, which is concerned with finding an isomorphism between two STRIPS instances of identical size. As we show that the problem is GI-complete, we prove that a quasi-polynomial time algorithm exists [2]. We then consider problem SSI, which is concerned with finding an isomorphism between a STRIPS instance and a subinstance of another STRIPS instance. We call such a mapping a *subinstance isomorphism*. After showing that this problem is NP-complete, we propose an algorithm that finds a subinstance isomorphism if one exists, or that detects that none exists. This algorithm is based on constraint propagation techniques, that allow us to prune impossible associations between elements of P and P', as well as on a reduction to SAT.

So far we have assumed that the two planning instances P and P' have the same initial states and goals (modulo the isomorphism). Even when this is not the case, an isomorphism from P to a subinstance of P' can still be of use. For example, if π is a solution-plan for P, then its image in P' can be converted to a single new operator which could be added to P' to facilitate its resolution. Such an operator would have the image of the initial state I of P for precondition, and the image of the result of the application of π to I for effect, thus abstracting away the application of the sequence of operators π. We therefore also consider this weaker notion of subinstance isomorphism, that we call *homogeneous subinstance isomorphism*, and the corresponding computational problem SSI-H.

Previous work investigated the complexity of various problems related to finding solution-plans for STRIPS planning instances [4], or focused on the complexity of solving instances from specific domains [8]. More scarcely, problems focused on altering planning models have been studied from a complexity theory point of view, such as the problem concerned with adapting a planning model so that some user-specified plans become feasible [10].

The paper is organized as follows. In Section 2, we introduce general notations, concepts and constructions that we use throughout this paper. In Section 3 and Section 4, we present our complexity results, for SI and SSI respectively. In Section 5, we present the outline of our algorithm for SSI. Section 6 is dedicated to the experimental evaluation and discussion.

2 Preliminaries

2.1 Automated Planning

A STRIPS planning problem is a tuple $P = \langle F, I, O, G\rangle$ such that F is a set of *fluents* (propositional variables whose values can change over time), I and G are sets of literals of F, called the *initial state* and *goal*, and O is a set of *operators*. Operators are of the form $o = \langle \mathsf{pre}(o), \mathsf{eff}(o)\rangle$. $\mathsf{pre}(o)$ and $\mathsf{eff}(o)$ are, respectively, the *precondition* and *effect* of o, which are sets of literals of F. We will denote $\mathsf{pre}^+(o) = \{f \in F \mid f \in \mathsf{pre}(o)\}$ the positive fluents of $\mathsf{pre}(o)$, and $\mathsf{pre}^-(o) = \{f \in F \mid \neg f \in \mathsf{pre}(o)\}$ the negative fluents. Similarly, $\mathsf{eff}^+(o) = \{f \in F \mid f \in \mathsf{eff}(o)\}$ and $\mathsf{eff}^-(o) = \{f \in F \mid \neg f \in \mathsf{eff}(o)\}$. By a slight abuse of notation, we will denote $\mathsf{pre} : O \to 2^F \cup 2^{\neg F}$ the function $o \mapsto \mathsf{pre}(o)$, and use similar notations for $\mathsf{pre}^+, \mathsf{pre}^-, \mathsf{eff}$, eff^+, and eff^-. In the rest of this paper, we will note $\mathcal{C} = \{\mathsf{pre}^+, \mathsf{pre}^-, \mathsf{eff}^+, \mathsf{eff}^-\}$. We will also use the notation that, for any set S of literals of F, $\neg S = \{\neg l \mid l \in S\}$.

A state s is an assignment of truth values to all fluents in F. For notational convenience, we associate s with the set of literals of F which are true in s. Given an instance $P = \langle F, I, O, G\rangle$, a *solution-plan* is a sequence of operators $o_1, \ldots, o_k$ from O such that the sequence of states $s_0, \ldots, s_k$ defined by $s_0 = I$ and $s_i = (s_{i-1} \setminus \mathsf{eff}^-(o_i)) \cup \mathsf{eff}^+(o_i)$ $(i = 1, \ldots, k)$ satisfies $\mathsf{pre}^+(o_i) \subseteq s_{i-1}$, $\mathsf{pre}^-(o_i) \cap s_{i-1} = \varnothing$ $(i = 1, \ldots, k)$ and $G \subseteq s_k$. A *plan* is defined similarly but without the conditions concerning I and G.

2.2 Complexity Class GI

This section introduces the complexity class GI, for which SI is later shown to be complete. GI is built around the Graph Isomorphism problem, which consists in finding a bijection $u : V \to V'$ between the vertices of two graphs $\mathcal{G}(V, E)$ and $\mathcal{G}'(V', E')$, such that the images of vertices linked by an edge in $\mathcal{G}$ are also linked by an edge in $\mathcal{G}'$ (and vice-versa). Formally, we require that the following condition holds:

$$\{x, y\} \in E \text{ iff } \{u(x), u(y)\} \in E' \tag{1}$$

▶ **Definition 1.** *The complexity class GI is the class of problems with a polynomial-time Turing reduction to the Graph Isomorphism problem.*

Complexity class GI contains numerous problems concerned with the existence of an isomorphism between two non-trivial structures encoded explicitly. Such problems are often complete for the class: finding an isomorphism between colored graphs, hypergraphs, automata, etc. are GI-complete problems [15]. In particular, we later use the following result:

▶ **Proposition 2** ([15], Ch. 4, Sec. 15)**.** *The* Oriented Graph Isomorphism problem *is GI-complete.*

As with the Graph Isomorphism problem, an isomorphism between oriented graphs $\mathcal{G}(V, E)$ and $\mathcal{G}'(V', E')$ is a bijection $u : V \to V'$ such that $(x, y) \in E$ iff $((u(x), u(y)) \in E'$.

In this paper, we consider another category of structures, called Finite Model, defined below. Finite models are also such that the related isomorphism existence problem is GI-complete.

▶ **Definition 3.** *A Finite Model is a tuple $M = \langle V, R_1, \ldots, R_n\rangle$ where V is a finite non-empty set and each R_i is a relation on elements of V with a finite number of arguments.*

Let $M = \langle V, R_1, \ldots, R_n\rangle$ and $M' = \langle V', R_1', \ldots, R_n'\rangle$ be two finite models. An isomorphism between M and M' is a bijection $u : V \to V'$ such that, for any $i \in \{1, \ldots, n\}$, for any set of elements $v_1, \ldots, v_m$ with m the arity of R_i, $R_i(v_1, \ldots, v_m)$ iff $R_i'(u(v_1), \ldots, u(v_m))$.

▶ **Proposition 4** ([15], Ch. 4, Sec. 15)**.** *The* Finite Model Isomorphism problem *is GI-complete.*

Class GI is believed to be an intermediate class between P and NP: the Graph Isomorphism problem can indeed be solved in quasi-polynomial time [2]. Although the problem is thought not to be NP-complete, no polynomial time algorithm is known.

2.3 Graph encodings into STRIPS

In this section, we present two ways to encode a graph $\mathcal{G} = (V, E)$ into a planning problem $P = \langle F, I, O, G\rangle$. These constructions are needed at various points in the rest of this paper, and only differ in that they take into account, or not, the orientation of the edges of $\mathcal{G}$. The intuition behind these constructions is that they model an agent that would move on the graph, resting on vertices and moving along edges. An agent being on vertex v would thus be denoted by the state $\{v\}$, where all fluents other than v are false.

In order to make the construction and resulting proofs simpler to read, for any pair $(v_s, v_t) \in F^2$, we will denote $\mathsf{move}(v_s, v_t)$ the operator that represents a movement from vertex v_s to vertex v_t. Keeping in mind that $F = V$, we have, more formally:

$$\mathsf{move}(v_s, v_t) = \langle \{v_s\}, \{v_t\} \cup \neg(V \setminus \{v_t\})\rangle$$

Where $\{v_s\}$ is the precondition of the operator, and $\{v_t\} \cup \neg(V \setminus \{v_t\})$ its effects. In the following construction, the vertices (resp. edges) of $\mathcal{G}$ are in bijection with the fluents (resp. operators) of P. In particular, we do not allow multi-edges. Other alternative constructions for move could have been used, as long as the encoding of each edge is unique. The one we propose is sufficient for our theoretical use, even though they encode trivial planning tasks.

▶ **Construction 5.** *Let $\mathcal{G} = (V, E)$ be an oriented graph. Let us build the planning problem $P_{\mathcal{G}} = \langle F, I, O, G\rangle$, where:*

$$F = V$$
$$O = \{\mathit{move}(v_s, v_t) \mid (v_s, v_t) \in E\}$$
$$G = I = \varnothing$$

In the case of non-oriented graphs, the construction is essentially the same, except that moves are possible in both directions. This gives us the following definition:

▶ **Construction 6.** *Let $\mathcal{G} = (V, E)$ be a non-oriented graph. Let us build the planning problem $P_{\mathcal{G}} = \langle F, I, O, G\rangle$, where F, I and G are defined as in Construction 5, but where:*

$$O = \{\mathit{move}(v_s, v_t), \mathit{move}(v_t, v_s) \mid \{v_s, v_t\} \in E\}$$

3 STRIPS Isomorphism Problem

This section is concerned with the problem of finding an isomorphism between two STRIPS planning problems. After introducing the notion of isomorphism between STRIPS instances that we use throughout this paper, we formally introduce problem SI, and settle its complexity.

▶ **Definition 7** (Isomorphism between STRIPS instances)**.** *Let $P = \langle F, I, O, G\rangle$ and $P' = \langle F', I', O', G'\rangle$ be two STRIPS instances. An* isomorphism *from P to P' is a pair (υ, ν) of bijections $\upsilon : F \to F'$ and $\nu : O \to O'$ that respect the following three conditions:*

$$\forall o \in O,\ \nu(o) = \langle \upsilon(\mathit{pre}(o)), \upsilon(\mathit{eff}(o))\rangle \tag{2}$$
$$\upsilon(I) = I' \tag{3}$$
$$\upsilon(G) = G' \tag{4}$$

Where, by a slight abuse of notation, for any two sets F_1 and F_2 of fluents of F,

$$\upsilon(F_1 \cup \neg F_2) = \upsilon(F_1) \cup \neg\upsilon(F_2)$$

An immediate property of this definition is that it carries over all plans: any sequence $o_1, \ldots, o_n$ of operators of O is a plan for P if, and only if, the corresponding plan $\nu(o_1), \ldots, \nu(o_n)$ is a plan for P'. This homomorphism property is enforced by equation (2). Similarly, all solution-plans carry over, as enforced by the additional conditions defined in equations (3) and (4). We now introduce the problem SI formally, and analyze its complexity:

▶ **Problem 8.** *STRIPS Isomorphism problem SI*

Input: *Two STRIPS instances P and P'*

Output: *An isomorphism (υ, ν) between P and P', if one exists*

▶ **Proposition 9.** *SI is GI-complete*

The rest of this section is dedicated to the proof of this result. We first show the GI-hardness of the problem, and then that it belongs to GI.

▶ **Lemma 10.** *SI is GI-hard*

Proof. The proof consists in a reduction from the Oriented Graph Isomorphism problem to SI. Let $(\mathcal{G}, \mathcal{G}')$ be an instance of the Oriented Graph Isomorphism problem, where $\mathcal{G} = (V, E)$ and $\mathcal{G}' = (V', E')$. The proof relies on Construction 5, which gives us in polynomial time the STRIPS planning problems $P_\mathcal{G}$ and $P_{\mathcal{G}'}$.

We show that there exists an isomorphism $u : V \to V'$ between $\mathcal{G}$ and $\mathcal{G}'$ iff there exists an isomorphism (υ, ν) between $P_\mathcal{G}$ and $P_{\mathcal{G}'}$. The main idea consists in, first, identifying mappings u and υ, and second, showing that the morphism condition between the edges of graphs $\mathcal{G}$ and $\mathcal{G}'$ is enforced by the morphism condition on the operators of STRIPS instances $P_\mathcal{G}$ and $P_{\mathcal{G}'}$, and vice-versa.

($\Rightarrow$) Suppose that there exists a graph isomorphism $u : V \to V'$ between $\mathcal{G}$ and $\mathcal{G}'$, and let us show that there exists an isomorphism between $P_\mathcal{G}$ and $P_{\mathcal{G}'}$. We define the transformation ν on elements of O by $\nu(\langle \mathsf{pre}(o), \mathsf{eff}(o)\rangle) = \langle u(\mathsf{pre}(o)), u(\mathsf{eff}(o))\rangle$. We will show that $\nu : O \to O'$ is well-defined and that the pair (u, ν) forms an isomorphism between $P_\mathcal{G}$ and $P_{\mathcal{G}'}$. For any $o \in O$, by construction, there exists a unique pair $(v_1, v_2) \in V^2$ such that $o = \mathsf{move}(v_1, v_2)$. Thus, we have that

$$\begin{aligned} o \in O \;& \textit{iff } (v_1, v_2) \in E \\ & \textit{iff } (u(v_1), u(v_2)) \in E' \\ & \textit{iff } \mathsf{move}(u(v_1), u(v_2)) \in O' \\ & \textit{iff } \nu(\mathsf{move}(v_1, v_2)) \in O' \\ & \textit{iff } \nu(o) \in O' \end{aligned}$$

The arguments from one line to the other stem from the construction of the various objects we use. Thus, we have shown that $P_\mathcal{G}$ and $P_{\mathcal{G}'}$ are isomorphic.

($\Leftarrow$) Suppose that $P_\mathcal{G}$ and $P_{\mathcal{G}'}$ are isomorphic, and that there exists an isomorphism (υ, ν) between them. We will show that there exists an isomorphism between $\mathcal{G}$ and $\mathcal{G}'$. By hypothesis, we have $\upsilon : F \to F'$ (or $\upsilon : V \to V'$) and $\nu : O \to O'$ two bijections.

In the following, we will denote by g and g' the bijections $g : E \to O$ and $g' : E' \to O'$, that exist by the construction (e.g. $g((v_1, v_2)) = \mathsf{move}(v_1, v_2)$).

Let us show that the function υ is a graph isomorphism between $\mathcal{G}$ and $\mathcal{G}'$. We have that, for any $e = (v_1, v_2) \in E$, $g(e) = \mathsf{move}(v_1, v_2) \in O$. So $\nu \circ g(e) = \nu(\mathsf{move}(v_1, v_2))$, and as such, $\nu \circ g(e) = \mathsf{move}(\upsilon(v_1), \upsilon(v_2)) \in O'$. Then, $g'^{-1} \circ \nu \circ g(e) = (\upsilon(v_1), \upsilon(v_2))$, but also $g'^{-1} \circ \nu \circ g(e) \in E'$. As a consequence, $(\upsilon(v_1), \upsilon(v_2)) \in E'$.

With similar arguments, as g, g' and ν are bijections, the converse can be shown. As a consequence, ν is a graph isomorphism between $\mathcal{G}$ and $\mathcal{G}'$. ◀

▶ **Lemma 11.** *SI is in GI.*

Proof. The proof follows a reduction from SI to the Finite Model isomorphism problem, as defined in Definition 3. It is based on the following construction: for any planning problem $P = \langle F, I, O, G\rangle$, we build the finite model M_P, such that:

$$M_P = \langle V, \mathcal{R}_F, \mathcal{R}_I, \mathcal{R}_O, \mathcal{R}_G, \mathcal{R}_{\mathsf{pre}^+}, \mathcal{R}_{\mathsf{pre}^-}, \mathcal{R}_{\mathsf{eff}^+}, \mathcal{R}_{\mathsf{eff}^-}\rangle$$
$$V = F \sqcup O$$
$$\text{For } X \in \{F, I, O, G\},\ \mathcal{R}_X = X$$
$$\text{For each } \mathcal{S} \in \mathcal{C},\ \mathcal{R}_\mathcal{S} = \big\{(o, f) \in V^2 \mid o \in O \textit{ and } f \in \mathcal{S}(o)\big\}$$

where $\mathcal{C} = \{\mathsf{pre}^+, \mathsf{pre}^-, \mathsf{eff}^+, \mathsf{eff}^-\}$. We will show that any two STRIPS planning problems P and P' are isomorphic iff M_P and $M_{P'}$ are isomorphic.

Let us denote $M_P = \langle V, \mathcal{R}_F, \ldots, \mathcal{R}_{\mathsf{eff}^-}\rangle$ and $M_{P'} = \langle V', \mathcal{R}'_F, \ldots, \mathcal{R}'_{\mathsf{eff}^-}\rangle$

($\Rightarrow$) Suppose that there exists an isomorphism (υ, ν) between P and P'. Define the mapping $g : V \to V'$ such that, for $x \in V$,

$$g(x) = \begin{cases} \upsilon(x) & \text{if } x \in F \\ \nu(x) & \text{if } x \in O \end{cases} \tag{5}$$

g is immediately a bijection, by hypothesis on (υ, ν). In addition, for $X \in \{F, I, O, G\}$, $\mathcal{R}_X(v)$ iff $\mathcal{R}'_X(g(v))$, by hypothesis on (υ, ν).

Let $o \in O$, $p \in F$. We have that, for any $\mathcal{S} \in \mathcal{C} = \{\mathsf{pre}^+, \mathsf{pre}^-, \mathsf{eff}^+, \mathsf{eff}^-\}$,

$$\begin{aligned} \mathcal{R}_\mathcal{S}(o,p) &\textit{ iff } o \in O \textit{ and } p \in \mathcal{S}(o) & (6)\\ &\textit{ iff } \nu(o) \in O' \textit{ and } \upsilon(p) \in \mathcal{S}(\nu(o)) & (7)\\ &\textit{ iff } \mathcal{R}'_\mathcal{S}(\nu(o), \upsilon(p)) \\ &\textit{ iff } \mathcal{R}'_\mathcal{S}(g(o), g(p)) \end{aligned}$$

The passage from (6) to (7) is by definition of the isomorphism. The other equivalences follow mostly by definition. This proves that M_P and $M_{P'}$ are isomorphic.

($\Leftarrow$) Suppose that M_P and $M_{P'}$ are isomorphic, and that $g : V \to V'$ is an isomorphism between the two models. Let us define $\upsilon = g_{|F}$ (resp. $\nu = g_{|O}$) the restriction of g on the subdomain F (resp. O). Clearly, we have that $\upsilon : F \to F'$, as otherwise there would exist an element $v \in V$ such that $\mathcal{R}_F(v)$ but *without* $\mathcal{R}'_F(g(v))$, violating the isomorphism hypothesis on g. Similarly, we have $\nu : O \to O'$. We have, as above, for any $o \in O$,

$$\begin{aligned} & o = \langle \mathsf{pre}(o), \mathsf{eff}(o)\rangle \\ &\textit{iff } \forall p \in F, \forall \mathcal{S} \in \mathcal{C}, p \in \mathcal{S}(o) \Leftrightarrow \mathcal{R}_S(o,p) & (8)\\ &\textit{iff } \forall p \in F, \forall \mathcal{S} \in \mathcal{C}, p \in \mathcal{S}(o) \Leftrightarrow \mathcal{R}'_\mathcal{S}(g(o), g(p)) & (9)\\ &\textit{iff } \forall p \in F, \forall \mathcal{S} \in \mathcal{C}, g(p) \in g(\mathcal{S}(o)) \Leftrightarrow \mathcal{R}'_\mathcal{S}(g(o), g(p)) & (10)\\ &\textit{iff } \forall p' \in F', \forall \mathcal{S} \in \mathcal{C}, p' \in g(\mathcal{S}(o)) \Leftrightarrow \mathcal{R}'_\mathcal{S}(g(o), p') & (11)\\ &\textit{iff } g(o) = \langle g(\mathsf{pre}(o)), g(\mathsf{eff}(o))\rangle & (12)\\ &\textit{iff } \nu(o) = \langle \upsilon(\mathsf{pre}(o)), \upsilon(\mathsf{eff}(o))\rangle \end{aligned}$$

The relations between the first line and (8), as well as between (11) and (12) hold by construction of M_P and $M_{P'}$. The equivalence between (8) and (9) comes from the hypothesis that g is an isomorphism. Between (9) and (10), we use that g is a bijection. For the equivalence between (10) and (11), we use that g is surjective over F'.

This finally proves that (υ, ν) is a homomorphism, and thus an isomorphism by choice of its domain and codomain. ◀

The results still hold if we do not enforce conditions (3) and (4), that the initial and goal states of P and P' are in bijection. Indeed, the hardness proof relies on a reduction from the Graph Isomorphism problem, with graphs that do not have initial or goal nodes, which renders trivial the initial and goal states of the construction. Conversely, the proof that SI belongs to class GI can include, or not, the relations $\mathcal{R}_I$ and $\mathcal{R}_G$ that take into account the information concerning initial and goal states, and still remain correct for the version of SI without conditions on the initial and goal states. This means that the hardness of SI comes from matching the inner structure of the state-space, and that additional properties on some states (like being initial states or goal states) do not impact significantly the complexity of the problem. This is consistent with our intuition of class GI: it is known that finding a color-preserving isomorphism between colored graphs (i.e., an isomorphism that conserves a given property on nodes) is also a problem that is complete for this class [15].

4 The STRIPS Subinstance isomorphism problem

Let us now introduce problems SSI-H and SSI, which are concerned with finding (different kinds of) isomorphisms between a planning instance P and some subinstance of another STRIPS instance P'. In this section, we settle the complexity of both problems, and show that they are NP-complete. We use this result in order to propose, in the next section, an algorithm for SSI and SSI-H. This algorithm is based on a reduction to SAT, assisted by a preprocessing phase that relies on constraint propagation.

We begin by introducing the notion of *homogeneous subinstance isomorphism*, which is concerned with finding an isomorphism between P and a subinstance of P', but does not conserve the initial state and goal. It maps the whole state-space of problem P to a part of the state-space of problem P', regardless of the initial state and goal of either problem.

▶ **Definition 12** (Homogeneous subinstance isomorphism). *Consider two STRIPS instances* $P = \langle F, I, O, G\rangle$ *and* $P' = \langle F', I', O', G'\rangle$. *A* homogeneous subinstance isomorphism *from* P *to* P' *is a pair* (υ, ν) *of* injective *mappings* $\upsilon : F \to F'$ *and* $\nu : O \to O'$ *that respect condition (2) of Definition 7.*

▶ **Problem 13.** *STRIPS Homogeneous Subinstance Isomorphism SSI-H*

Input *Two STRIPS instances* P *and* P'

Output *A* homogeneous subinstance isomorphism (υ, ν) *between* P *and* P'*, if one exists*

A homogeneous subinstance isomorphism between P and P' is useful, for instance, in the case where we managed to compile all plans for P', and wish to extract a plan for P. The following more precise notion of isomorphism between P and a subinstance of P' takes into account the information provided by the initial state and goal. This allows us to carry over only *solution*-plans from one problem to the other.

▶ **Definition 14** (Subinstance isomorphism). *A* subinstance isomorphism *from* P *to* P' *is a homogeneous subinstance isomorphism that respects conditions (3) and (4) of Definition 7.*

▶ **Problem 15.** *STRIPS Subsintance Isomorphism SSI*

Input *Two STRIPS instances* P *and* P'

Output *A subinstance isomorphism* (υ, ν) *between* P *and* P'*, if one exists*

The main difference between SI and SSI is that, in SSI, we relax the condition on the bijectivity of υ and ν, to account for the difference in size between P and P'. Their injectivity is still required in order to prevent fluents (or operators) being merged together by the mapping. All other conditions remain the same.

The main result of this section is presented below. The proof is based on a reduction from the Subgraph Matching problem, which is known to be NP-complete [5]. As such, we introduce that problem before stating our result. Essentially, it consists in finding a mapping g, that defines an isomorphism between $\mathcal{G}$ and the subgraph $(g(V), E' \cap g(V) \times g(V))$ of $\mathcal{G}'$.

▶ **Problem 16.** *Subgraph Matching problem*

Input *Two non-oriented graphs $\mathcal{G}(V, E)$ and $\mathcal{G}'(V', E')$*

Output *An injective mapping $g : V \to V'$ such that, for any $v_1, v_2 \in V$, $\{v_1, v_2\} \in E$ iff $\{g(v_1), g(v_2)\} \in E'$.*

▶ **Proposition 17.** *SSI is NP-complete*

Proof. In order to prove that SSI is in NP, it suffices to resort to the certificate-based definition of the class NP, and observe that the mappings υ and ν constitute a polynomial size certificate that can be checked in polynomial time.

The proof that SSI is NP-hard consists in a reduction from the Subgraph Matching problem, which is straightforward with the construction that we proposed earlier.

Let $(\mathcal{G}, \mathcal{G}')$ be an instance of the Subgraph Matching problem, and let us follow Construction 6 to build planning problems $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$. We show that there exists a subgraph matching g between $\mathcal{G}$ and $\mathcal{G}'$ iff there exists a subinstance isomorphism of $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$.

($\Rightarrow$) Suppose that there exists a subgraph matching $g : V \to V'$ between $\mathcal{G}$ and $\mathcal{G}'$. Then by construction, as $F = V$ and $F' = V'$, g is also an injective mapping between F and F'. In addition, let us define the mapping $\nu : O \to O'$ such that $\nu : \mathsf{move}(v_1, v_2) \mapsto \mathsf{move}(g(v_1), g(v_2))$. ν is well-defined, as $\{v_1, v_2\} \in E$ iff $\{g(v_1), g(v_2)\} \in E'$, so $\mathsf{move}(v_1, v_2) \in O$ iff $\mathsf{move}(g(v_1), g(v_2)) \in O'$. In addition, as g is injective, so is ν. As a consequence, (g, ν) is a subinstance isomorphism between $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$.

($\Leftarrow$) Suppose that there exists a subinstance isomorphism (υ, ν) between $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$. As above, $\upsilon : V \to V'$ is an injective mapping. In addition, we have that

$$\begin{aligned} & (v_1, v_2) \in E \\ \text{iff } & \mathsf{move}(v_1, v_2) \in O \\ \text{iff } & \nu\left(\mathsf{move}(v_1, v_2)\right) \in O' \\ \text{iff } & \mathsf{move}(\upsilon\left(v_1\right), \upsilon\left(v_2\right)) \in O' \\ \text{iff } & \left(\upsilon\left(v_1\right), \upsilon\left(v_2\right)\right) \in E' \end{aligned}$$

As a consequence, υ is a subgraph matching between $\mathcal{G}$ and $\mathcal{G}'$. ◀

In addition, it is clear that SSI-H is in NP. As the above proof of NP-hardness of SSI is independent of the initial and goal states, it also applies to the problem SSI-H.

▶ **Corollary 18.** *SSI-H is NP-complete*

5 An algorithm for SSI

In this section, we present an algorithm for problem SSI, for which the pseudo-code is presented in Algorithm 1. This algorithm is based on a compilation of the problem into a propositional formula, which is then passed to a SAT solver. It is completed by a preprocessing step, based on constraint propagation, that allows us to prune impossible mappings early on.

Algorithm 1 to find a subinstance isomorphism.

```
Input: Two STRIPS instances $P$ and $P'$
Output: A subinstance isomorphism between $P$ and $P'$ if one exists
 1: Initialize_domains($F, O$)
    /* Prune impossible associations */
 2: $Q := F \cup O$
 3: while $Q \neq \varnothing$ do
 4:     $v := Q$.Pop()
 5:     $r :=$ Revise($v$)
 6:     if $r$ then
 7:         if $\mathcal{D}(v) = \varnothing$ then return UNSAT
 8:         else $Q$.Add($\{v' \mid v'$ related to $v\}$)
    /* Search phase through a SAT solver */
 9: $\varphi :=$ Encode_to_SAT($P, P', \mathcal{D}$)
10: return Interpret(Solver.Find_model($\varphi$))
```

Given two STRIPS instances P and P', the algorithm outputs, when possible, a subinstance isomorphism (υ, ν). Algorithm 1 consists in two main phases. The first phase, that spans lines 2 to 8, consists in pruning as many associations between fluents (resp. operators) of problem P and fluents (resp. operators) of problem P' that are impossible, because of some syntactical inconsistencies (described below) that are then propagated. The second phase, that starts at line 9, consists in a search phase, by means of an encoding of the problem into a CNF formula, that is then passed to a SAT solver.

5.1 Pruning invalid associations

By *association* between fluents, we mean a pair $(f, f') \in F \times F'$ such that f' is a candidate for the value of $\upsilon(f)$. Similarly, we call an *association* between operators a pair $(o, o') \in O \times O'$ such that o' is a candidate for the value of $\nu(o)$. Detecting early on associations that can not be part of a valid subinstance isomorphism reduces the size of the search space.

In order to prune as many inconsistent associations as possible, we use a technique similar to constraint propagation, as commonly found in the constraint programming literature. The general idea is to maintain, for each fluent $f \in F$ of P, a *domain* $\mathcal{D}(f) \subseteq F'$ of fluents of P', that consists of the plausible candidates for the value of $\upsilon(f)$. Similarly, each operator $o \in O$ is assigned a domain $\mathcal{D}(o) \subseteq O'$, containing the plausible candidates for $\nu(o)$. In the following, we call fluents and operators *variables*. The aim of the procedure presented below is to trim the domains of the variables, thus alleviating the load left to the SAT solver.

The first step is to initialize the domains. For each fluent $f \in F$, we set $\mathcal{D}(f) = F'$. The initial assignment of the domains of operators $o \in O$, however, is based on *operator profiles*. For each operator $o \in O \cup O'$, we define the vector $\mathsf{profile}(o) \in \mathbb{N}^6$, called the *profile* of o. This vector numerically abstracts some characteristics of the operator, so that an operator $o \in O$ cannot be associated to operator $o' \in O'$ if $\mathsf{profile}(o) \neq \mathsf{profile}(o')$. In practice, $\mathsf{profile}(o)$ consists of the number of positive and negative fluents in the precondition and effect of o, as well as its number of *strict-add* and *strict-delete* fluents. A fluent f is said to be *strict-add* for operator o if $f \in \mathsf{pre}^-(o) \wedge f \in \mathsf{eff}^+(o)$, and *strict-delete* if $f \in \mathsf{pre}^+(o) \wedge f \in \mathsf{eff}^-(o)$. Then, we initialize the domain of each $o \in O$ such that

$$\mathcal{D}(o) = \{o' \in O' \mid \mathsf{profile}(o') = \mathsf{profile}(o)\}$$

The second step is to propagate the additional constraints posed by these newly-found restrictions of the domains. The technique we propose is based on the concept of arc consistency, which is ubiquitous in the field of constraint programming. The idea consists in eliminating, from the domains of fluents (resp. operators), the candidate fluents (resp. operators) that have no support in the domain of some operator (resp. fluent).

More specifically, let us consider a fluent $f \in F$. When an operator $o \in O$ is such that f appears, negated or not, in its precondition or effect, then we say that *o depends* on f. Let us denote $d(f)$ the set of operators that depend on f. When $f' \in F'$, we define $d(f')$ in a similar fashion. Now suppose that $\upsilon(f) = f'$. As a consequence of equation (2) of Definition 7, each operator of $d(f)$ must have its image by ν in $d(f')$. Otherwise, f would appear in $\mathsf{pre}(o)$ or $\mathsf{eff}(o)$, but $\upsilon(f)$ would not appear in $\upsilon(\mathsf{pre}(o))$ nor $\upsilon(\mathsf{eff}(o))$. Thus, if for some operator $o \in d(f)$ no candidate operator for its image is in $d(f')$ (*i.e.*, $\mathcal{D}(o) \cap d(f') = \varnothing$), then it means that f' can not be chosen as the image of f.

In the following, we refine the argument of last paragraph by identifying $\mathsf{pre}^+(o)$ with $\mathsf{pre}^+(o'), \ldots, \mathsf{eff}^-(o)$ with $\mathsf{eff}^-(o')$. We thus have the following constraint for $\mathcal{D}(f)$, where $\mathcal{C} = \{\mathsf{pre}^+, \mathsf{pre}^-, \mathsf{eff}^+, \mathsf{eff}^-\}$:

$$\mathcal{D}(f) \subseteq \left\{ f' \;\middle|\; \begin{array}{l} \forall o \in O, \forall \mathcal{S} \in \mathcal{C} \text{ s.t. } f \in \mathcal{S}(o), \\ \exists o' \in \mathcal{D}(o) \text{ s.t. } f' \in \mathcal{S}(o') \end{array} \right\} \tag{13}$$

A similar case can be made for operators. Let $o \in O$ be any operator, and consider a candidate operator $o' \in O'$. In order for the morphism property to hold, in the case where $\nu(o) = o'$, for every fluent f of $\mathsf{pre}^+(o)$, for instance, there must exist in $\mathsf{pre}^+(o')$ a fluent that belongs to $\mathcal{D}(f)$. More generally and more formally, we have the following:

$$\mathcal{D}(o) \subseteq \{o' \mid \forall \mathcal{S} \in \mathcal{C}, \forall f \in \mathcal{S}(o), \exists f' \in \mathcal{D}(f) \cap \mathcal{S}(o')\} \tag{14}$$

Algorithmically, we enforce these constraints using an adaptation of AC3 [12, 14]. The algorithm revolves around the revision of the coherence of the variables' domains. Revising a variable v boils down to checking that all elements of its domain still comply with the necessary condition evoked earlier, which is either equation (13) if v is a fluent, or equation (14) if v is an operator. The main loop, depicted in Algorithm 1, then consists in revising all fluents and operators iteratively, by maintaining a queue Q of variables to revise (line 1). The algorithm begins by revising once each variable. If, during the revision of a variable v, the domain of v is altered by the procedure, then all variables that are related to v are added to the set of variables to revise later on (lines 5 to 9). We say that v' is related to v if v is a fluent and $v' \in d(v)$, or conversely. If the domain of a variable is empty, then no isomorphism exists, and the procedure ends prematurely (line 6 and 7). Otherwise, the loop ends when there is no variable left to revise.

This procedure is often not sufficient to conclude, but greatly alleviates the pressure on the search phase, which we present in the following section.

5.2 Encoding into a SAT instance

In this section, we build the propositional formula φ evoked earlier, from the models of which an isomorphism can be extracted. φ is built on the set of variables $\mathit{Var}(\varphi)$, such that:

$$\mathit{Var}(\varphi) = \left\{ f_i^j \;\middle|\; i \in F, j \in F' \right\} \cup \{ o_r^s \mid r \in O, s \in O' \}$$

The propositional variable f_i^j represents the association of fluent $i \in F$ to fluent $j \in F'$. Likewise, o_r^s represents the association of $r \in O$ to $s \in O'$.

In the rest of this section, we show how to build formula φ, which encodes the SSI problem input to Algorithm 1. φ consists in the conjunction of the formulas presented below.

The formula presented in (15) enforces that each fluent has an image which is unique. Similarly, by swapping f_i^j variables for o_i^j and adapting the domains of i and j, we enforce that each operator has an image by ν.

$$\bigwedge_{i \in F} \left(\bigvee_{j \in \mathcal{D}(i)} f_i^j \;\wedge \bigwedge_{\substack{j,k \in \mathcal{D}(i) \\ j \neq k}} (\neg f_i^j \vee \neg f_i^k) \right) \tag{15}$$

We now need to ensure that υ and ν are injective. For fluents, this is done through (16). A similar formula is used to ensure the injectivity of ν on operators.

$$\bigwedge_{i \in F'} \bigwedge_{\substack{j,k \in F \\ j \neq k}} \neg f_j^i \vee \neg f_k^i \tag{16}$$

The morphism property is enforced by formulas (17) and (18), for each $\mathcal{S} \in \mathcal{C}$, where $\mathcal{C} = \{\mathsf{pre}^+, \mathsf{pre}^-, \mathsf{eff}^+, \mathsf{eff}^-\}$. More precisely, (17) ensures that, for any $\mathcal{S} \in \mathcal{C}$ and for any operator $o \in O$, we have $\upsilon(\mathcal{S}(o)) \subseteq \mathcal{S}(\nu(o))$. Conversely, (18) ensures that $\mathcal{S}(\nu(o)) \subseteq \upsilon(\mathcal{S}(o))$.

$$\bigwedge_{\substack{r \in O \\ s \in O'}} \left(o_r^s \longrightarrow \bigwedge_{i \in \mathcal{S}(r)} \bigvee_{j \in \mathcal{S}(s)} f_i^j \right) \tag{17}$$

$$\bigwedge_{\substack{r \in O \\ s \in O'}} \left(o_r^s \longrightarrow \bigwedge_{j \in \mathcal{S}(s)} \bigvee_{i \in \mathcal{S}(r)} f_i^j \right) \tag{18}$$

Finally, we need to conserve the initial and the goal state (i.e., respect equations (3) and (4)). Let us denote I^+ (resp. I^-) the set of fluents appearing positively (resp. negatively) in I, and use similar notation for G, I' and G'. For every $\mathcal{T} \in \{I^+, I^-, G^+, G^-\}$, and for the corresponding $\mathcal{T}' \in \{I'^+, I'^-, G'^+, G'^-\}$, we then add the following formulas:

$$\bigwedge_{i \in \mathcal{T}} \bigvee_{j \in \mathcal{T}'} f_i^j \quad \wedge \quad \bigwedge_{j \in \mathcal{T}'} \bigvee_{i \in \mathcal{T}} f_i^j \tag{19}$$

The formulas presented in (15), (16), and (19) are immediately in CNF, and the size of their conjunction is in $\mathcal{O}(|F| \cdot |F'|^2 + |O| \cdot |O'|^2)$ assuming $|F| \leq |F'|$ and $|O| \leq |O'|$. In addition, the formulas presented in (17) and (18) can be readily converted into CNF by duplicating the implication in each clause, and then have a size $\mathcal{O}(|O| \cdot |O'| \cdot |F| \cdot |F'|)$.

The preprocessing step presented in Section 5.1 allows us to simplify φ. Indeed, if it is known that fluent $i \in F$ (resp. $r \in O$) cannot be mapped to fluent $j \in F'$ (resp. $s \in O'$), then f_i^j (resp. o_r^s) is necessarily false in any model of φ. As a consequence, as all formulas are in CNF, every positive occurrence of f_i^j is removed in the clauses of φ, while clauses where f_i^j appears negatively are simplified.

In order to adapt the algorithm for SSI-H, it suffices to remove the set of formulas presented in (19). The others formulas and the rest of the algorithm remains the same.

Table 1 Number of instances of SSI-H and SSI on which our implementation of our method terminates within 600 seconds. For each problem, the first pair of columns shows the number of STRIPS matching instances solved with and without the constraint propagation-based preprocessing, respectively. The last column shows the average percentage of clauses that have been eliminated from the propositional encoding, thanks to the pruning step.

Domain	SSI-H			SSI		
	CP	NoCP	Av. Simp.	CP	NoCP	Av. Simp.
blocks	172	96	76.1%	166	93	76.2%
gripper	210	189	74.9%	90	84	75.1%
hanoi	74	75	0.2%	85	82	0.2%
rovers	19	6	97.4%	16	6	97.3%
satellite	34	22	79.1%	38	23	78.4%
sokoban	204	0	98.6%	205	4	98.6%
tsp	376	374	0.7%	265	266	1.0%

6 Experimental evaluation

We implemented Algorithm 1 in Python 3.10, and used it to solve SSI and SSI-H. In order to parse planning instances in PDDL and convert them into a STRIPS representation, we used the parser of TouISTPlan [3]. The SAT solver we used was Maple LCM [11], winner of the main track at SAT 2017. The code and sets of benchmarks are available online.

Experiments were run on a machine running Rocky Linux 8.5, powered by an Intel Xeon E5-2667 v3 processor, using at most 8GB of RAM and 4 threads per test. Our set of benchmarks is based on eight sets found in previous International Planning Competitions, namely Blocks, Gripper, Hanoi, Rovers, Satellite, Sokoban, TSP and Visitall. For each of these domains, we created what we call *STRIPS matching instances*, which are pairs of instances of the same domain. We did this for each possible pair of planning instances of each considered domain. A STRIPS matching instance is an instance of both SSI and SSI-H. We thus evaluated our algorithm adapted for both problems on the same set of benchmarks.

The goal of the experiments is twofold. First, the aim is to demonstrate that, despite the theoretical hardness of the problem, it is possible to find a (homogeneous) subinstance isomorphism in reasonable time for problems of non-trivial size. Second, the goal is to show the efficiency of the pruning technique presented in Section 5.1, i.e., to prove that the additional cost of the preprocessing is outbalanced by the speed-up it provides during search.

The coverage of our implementation on our set of benchmarks is shown in Table 1 for both SSI and SSI-H. The table shows the absolute and relative numbers of instances of SSI (resp. SSI-H) on which our implementation terminates within the time and memory cutoffs. Note that we tested our algorithm on a handful of other domains, but we only report those for which at least one instance was solved. Domains where even the smallest problem timeouts include Visitall, Barman and Woodworking, for instance.

The first point we notice is that problems SSI-H and SSI are often closely comparable in terms of hardness, except for some particular domains. These include domains TSP and Gripper, for which 40% and 133% more instances are solved when requiring no condition on the initial state and goal. For both domains, this is due to the additional constraints in SSI. Indeed, because of these constraints all pairs of non-identical TSP planning instances (or Gripper planning instances) constitute negative SSI instances, which turn out to be harder for the SAT solver to detect than positive ones.

A crucial observation is that the preprocessing step almost never holds back the algorithm: almost all instances of our test sets that can be solved without preprocessing are also solved when the preprocessing step is performed. Furthermore, in many sets of benchmarks, the preprocessing greatly improves the overall performance of our implementation, so much so that some previously infeasible domains are now within the range of our algorithm. Such extreme cases include Sokoban, for which our algorithm is powerless without the pruning step: all 204 instances solved by our implementation are outside the range of the preprocessing-less version of the algorithm. In most cases, however, we observe a significant increase in the coverage of the algorithm, that remains nonetheless within the same order of magnitude. For example, for domain Satellite in the case of SSI, 34 instances are solved when constraint propagation is enabled, whereas only 22 can be settled without it.

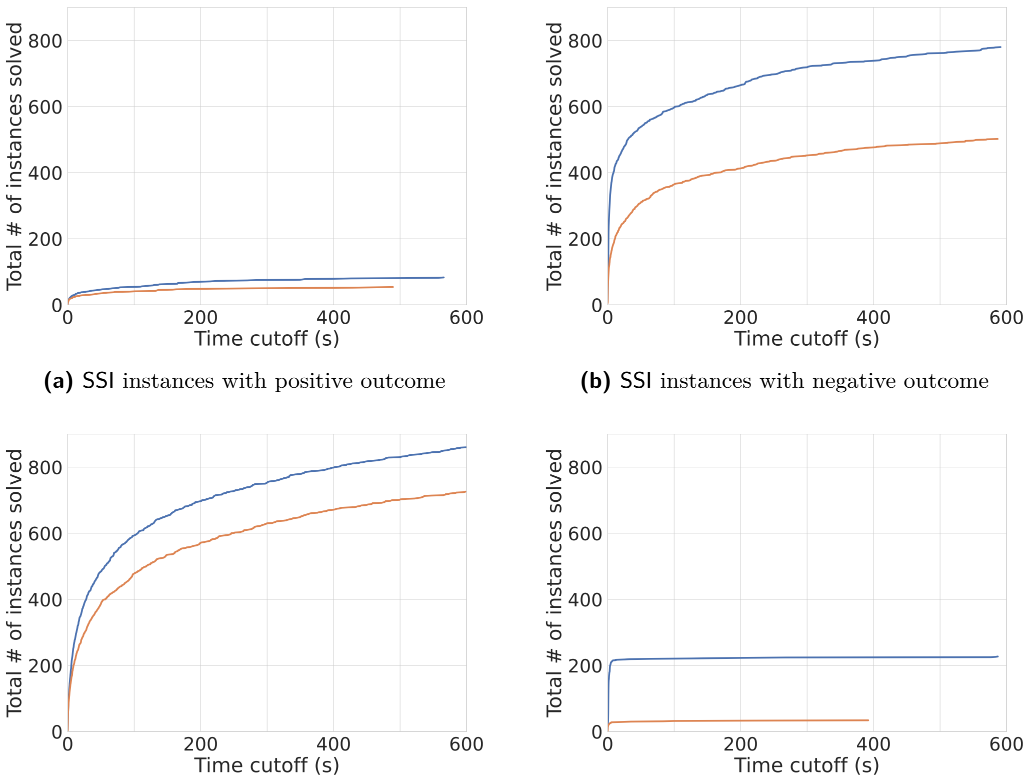
More specifically, in most cases, the preprocessing step leads to a reduction of the size of the propositional encoding. This is shown by the columns labeled "Av. Simp." in Table 1, which represent the average proportion of clauses that are simplified as a consequence of the pruning step. The highest percentages of simplified clauses are found in domains that contain little to no symmetries. For example, in Rovers, fluents represents entities that often have different types, and that are affected in different ways by operators. For instance, operators of the form `navigate(rover, x, y)` have a unique profile, and are not numerous. Consequently, their respective domains remain small, which is something our algorithm makes the most of.

On the contrary, for domains that contain lots of symmetries, the pruning step does not remove a significant number of associations. This is the case in Hanoi, where all operators have the same profile: except for the information provided by the initial and goal state, all disks are interchangeable, which does not allow our preprocessing to draw any conclusive result. The only bits of information that can guide the search are encoded in the initial state, which we believe partly explains the slightly greater coverage of SSI over SSI-H.

In some instances of our set of benchmarks, pruning suffices to find that no (homogeneous) subinstance isomorphism exists: as the majority of associations between fluents or between operators are ruled out, the domains of some variables become empty. As a direct consequence, our algorithm is most effective in the case where no (homogeneous) subinstance isomorphism exists. In many of these cases, an empty domain is found for a variable, which allows the algorithm to return UNSAT prematurely, and skip the search phase altogether. This is why the pruning step allows us to significantly increase our coverage on STRIPS matching instances that are negative, as shown in Figure 1, while our performance on positive instances is more modest, although significant.

In Table 2, we also show that the additional time required by the constraint propagation phase is negligible compared to the rest of the algorithm. In fact, be it in domains where it prunes out lots of associations or in domains where its efficiency is limited, constraint propagation rarely takes more than a handful of seconds. As a consequence, some instances that would otherwise require a substantial amount of time are now solved almost immediately. In addition, as shown in Figure 1b, solving any 500 negative SSI instances requires 10 minutes when pruning is not enabled, while it requires less than a minute when pruning is enabled.

In Table 3, we present a few results on the absolute sizes of the problems that we solved during our experiments, within the time and memory limits. For a STRIPS planning problem $P = \langle F, I, O, G \rangle$, we denote $|P| = |F| + |O|$. As an SSI instance has two main dimensions, represented by the respective sizes of the planning instances that constitute it, we present two different ways of measuring it size6. In the first set of columns of Table 3, the sum of both planning instances is considered, and we report the size of the SSI instance that maximizes that sum. With this metric, P' is often disproportionately bigger than P. This



(a) SSI instances with positive outcome

(b) SSI instances with negative outcome

(c) SSI-H instances with positive outcome

(d) SSI-H instances with negative outcome

Figure 1 Number of SSI (top) and SSI-H (bottom) instances that can be solved by our implementation, as a function of the time cutoff. Blue/orange curves correspond respectively to with/without pruning (constraint propagation preprocessing).

imbalance can be explained by the fact that the encoding into a propositional formula is of time and size $\mathcal{O}(|O| \cdot |O'| \cdot |F| \cdot |F'|)$, as mentioned previously. Thus, in the second set of columns, we consider instead the lexicographic order on pairs $(|P|, |P'|)$, and report the biggest problem with respect to that metric.

7 Conclusion

In this article, we introduced the problem SI, which is concerned with finding an isomorphism between two planning problems, and showed that it is GI-complete. Afterwards, we introduced the notion of subinstance isomorphism, as well as the associated problem SSI. In addition to proving the NP-completeness of the problem, we proposed an algorithm for it, based on constraint propagation techniques and a reduction to SAT

The experimental evaluation of said algorithm shows that traditional constraint propagation in a preprocessing step can greatly improve the efficiency of SAT solvers. However, even though it is not costly to perform, not all planning domains benefited equally from this preprocessing.

On a more general note, various methods have been proposed to automatically reformulate general models, with the aim of rendering easier the task delegated to the solver [13]. It remains an interesting open question to identify which characteristics of problems in NP make them amenable to this hybrid CP-SAT approach.

Table 2 Average time, in seconds, spent in each of the main three steps of the algorithm: pruning (CP), compilation to SAT, and solving, respectively. The last column summarizes the average total running time of the algorithm. We only report instances that were successfully solved (either positively or negatively), and results for SSI-H and SSI are thus non-comparable.

Domain	SSI-H				SSI			
	CP	Comp.	Solving	Total time	CP	Comp.	Solving	Total time
blocks	0.5	93.3	76.8	170.3	0.4	83.3	94.6	178.1
gripper	0.2	23.5	9.8	33.4	0.1	11.2	35.2	46.5
hanoi	0.3	43.9	78.0	122.0	0.3	70.9	47.8	118.9
rovers	1.8	168.9	2.2	171.4	1.7	180.7	2.7	183.5
satellite	0.4	116.4	48.8	165.4	0.4	85.0	10.3	95.4
sokoban	1.7	222.7	2.3	225.2	1.7	220.6	1.4	222.3
tsp	0.2	50.7	46.7	97.5	0.1	14.6	26.6	41.2

Table 3 Sizes of the biggest instances that can be solved by our implementation within the time and memory limits, for both SSI-H and SSI. In the first set of columns, we consider the sum of the sizes of the planning instances that constitute the STRIPS matching instance. In the second set, we consider the size of P, the smallest planning instance among the pair that constitutes the instance.

Domain	SSI-H					SSI				
	Maximum sum			Max $\lvert P\rvert$		Maximum sum			Max $\lvert P\rvert$	
	$\lvert P\rvert$	$\lvert P'\rvert$	Sum	$\lvert P\rvert$	$\lvert P'\rvert$	$\lvert P\rvert$	$\lvert P'\rvert$	Sum	$\lvert P\rvert$	$\lvert P'\rvert$
blocks	57	4642	4699	534	534	57	4642	4699	534	534
gripper	510	510	1020	510	510	510	510	1020	510	510
hanoi	13	3328	3341	391	391	13	6953	6966	391	513
rovers	276	2667	2943	920	920	276	2667	2943	920	920
satellite	147	2066	2213	608	920	147	2610	2757	608	920
sokoban	2212	2286	4498	2212	2286	2212	2286	4498	2212	2286
tsp	182	930	1112	462	462	90	930	1020	380	380

References

1 Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic csps application to configuration. *Artif. Intell.*, 135(1-2):199–234, 2002. doi:10.1016/S0004-3702(01)00162-X.

2 László Babai. Group, graphs, algorithms: the graph isomorphism problem. In *Proceedings of the International Congress of Mathematicians*, pages 3319–3336. World Scientific, 2018.

3 Djamila Baroudi, Maël Valais, and Frédéric Maris. Touistplan. URL: https://github.com/touist/touistplan.

4 Tom Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

5 Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *3rd Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971. doi:10.1145/800157.805047.

6 Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002. doi:10.1613/jair.989.

7 Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971. doi:10.1016/0004-3702(71)90010-5.

8 Malte Helmert. Complexity results for standard benchmark domains in planning. *Artificial Intelligence*, 143(2):219–262, 2003. doi:10.1016/S0004-3702(02)00364-8.

9 Henry A. Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, *ECAI 92*, pages 359–363. John Wiley and Sons, 1992.

10 Songtuan Lin and Pascal Bercher. Change the world - how hard can that be? On the computational complexity of fixing planning models. In Zhi-Hua Zhou, editor, *IJCAI-21*, pages 4152–4159, August 2021. doi:10.24963/ijcai.2021/571.

11 Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for cdcl sat solvers. In *IJCAI-17*, pages 703–711, 2017. doi:10.24963/ijcai.2017/98.

12 Alan K Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118, 1977.

13 Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in savile row. *Artif. Intell.*, 251:35–61, 2017. doi:10.1016/j.artint.2017.07.001.

14 Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

15 Viktor N Zemlyachenko, Nickolay M Korneenko, and Regina I Tyshkevich. Graph isomorphism problem. *Journal of Soviet Mathematics*, 29(4):1426–1481, 1985.

Solving the Constrained Single-Row Facility Layout Problem with Decision Diagrams

Vianney Coppé
UCLouvain, Louvain-la-Neuve, Belgium

Xavier Gillard
UCLouvain, Louvain-la-Neuve, Belgium

Pierre Schaus
UCLouvain, Louvain-la-Neuve, Belgium

Abstract

The Single-Row Facility Layout Problem is an NP-hard problem dealing with the ordering of departments with given lengths and pairwise traffic intensities in a facility. In this context, one seeks to minimize the sum of the distances between department pairs, weighted by the corresponding traffic intensities. Practical applications of this problem include the arrangement of rooms on a corridor in hospitals or offices, airplanes and gates in an airport or machines in a manufacture. This paper presents two novel exact models for the Constrained Single-Row Facility Layout Problem, a recent variant of the problem including positioning, ordering and adjacency constraints. On the one hand, the state-of-the-art mixed-integer programming model for the unconstrained problem is extended to incorporate the constraints. On the other hand, a decision diagram-based approach is described, based on an existing dynamic programming model for the unconstrained problem. Computational experiments show that both models outperform the only mixed-integer programming model in the literature, to the best of our knowledge. While the two models have execution times of the same order of magnitude, the decision diagram-based approach handles positioning constraints much better but the mixed-integer programming model has the advantage for ordering constraints.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization

Keywords and phrases Discrete Optimization, Mixed-Integer Programming, Decision Diagrams, Constrained Single-Row Facility Layout Problem

Digital Object Identifier 10.4230/LIPIcs.CP.2022.14

Supplementary Material *Software (Source Code)*: `https://github.com/vcoppe/csrflp-dd`
Software (Source Code): `https://github.com/vcoppe/csrflp-mip`

1 Introduction

The *Single-Row Facility Layout Problem* (SRFLP) is an ordering problem considering a set of departments in a facility, with given lengths and pairwise traffic intensities. Its goal is to find a linear ordering of the departments minimizing the weighted sum of the distances between department pairs. The SRFLP is applied in different fields to arrange items such as rooms on a corridor in hospitals or offices [52], airplanes and gates in an airport [53], machines in a manufacture [30], books on a shelf and files in disk cylinders [50]. When all facilities have equal lengths and the traffic intensities are binary, the problem is known as the *Minimum Linear Arrangement Problem* (MinLA). It is a well-known graph layout problem which has been proved to be NP-hard [20] and consequently, so is the SRFLP.

Due to its difficulty in being solved by exact methods, many heuristic techniques have been designed to find good quality solutions to the SRFLP problem [19, 28, 29, 41] and more recently [18, 23, 40, 47, 51]. The first attempt to solve the SRFLP optimally was a branch-and-bound algorithm with interesting lower bounds [52]. Later, the DP approach

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 14; pp. 14:1–14:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

presented in [39] was applied to the SRFLP in [50]. More recent techniques include non-linear programming [31], linear *mixed-integer programming* (MIP) [2, 3, 46], branch-and-cut [4, 5] and semidefinite programming [7, 8, 9, 35, 36].

In [37], *positioning*, *ordering* and *relation* constraints were suggested for the SRFLP to model real-life situations. The resulting problem is called the *Constrained Single-Row Facility Layout Problem* (cSRFLP). They also proposed a permutation-based genetic algorithm to solve this new problem and reported very good results, with objective values deviating by only a few percents from the best known solutions to the unconstrained problem for instances with up to 100 departments. In [44], the first MIP model solving the cSRFLP is introduced and a constrained improved fireworks algorithm is described. The latter is shown to find solutions of better quality than the genetic algorithm of [37].

This paper begins with a formal definition of the SRFLP in Section 2 and of the constraints that constitute the cSRFLP. We then present two novel exact models to solve the problem. In Section 3, we model the constraints of the cSRFLP on top of the state-of-the-art MIP model for the SRFLP [4]. Likewise, Section 4 recalls the *dynamic programming* (DP) model for the SRFLP from [50] and shows how the new constraints can be integrated. This DP model will be used as the basis of a decision diagram-based approach described in detail in Sections 5 and 6.

A *decision diagram* (DD) is a data structure used to encode sets in a compressed form through a graphical representation. They first appeared as binary decision diagrams for the representation of Boolean functions and were successfully used for circuit design and formal verification [1, 15, 34, 43]. Among the wide variety of domains in which the DDs were applied through the years [45, 56], the compactness which they provide was exploited in constraint programming [32, 48, 55] and optimization [10, 25, 26, 27, 42]. Recently, a complete framework for discrete optimization with decision diagrams was introduced in [13]. It relies on a DP model of the problem, which can represent the solution space in a compact form. In spite of their compactness, DDs encoding hard optimization problems may not fit in memory. The exact optimization method is therefore built upon *relaxed* and *restricted* DDs. These approximate DDs were introduced in [6, 11, 14] for their ability to provide tight lower and upper bounds [16, 17, 33, 54]. An adapted branch-and-bound algorithm based exclusively on DDs was presented in [13].

In Section 7, the results of our computational experiments are presented. They show that our two new models outperform the MIP model from [44] in terms of solving time. Other than that, there is no clear winner between the DD approach and the new MIP model. The former seems to handle positioning constraints better while the latter is particularly efficient for relation constraints. However, the ability to parallelize the DD approach is unmatched by the MIP solver. The paper concludes with a summary of our contributions and directions for future work.

2 Problem Definition

This section is organized as follows, a formal definition of the SRFLP is given in Section 2.1 which is then completed in Section 2.2 with the constraints that constitute the cSRFLP.

2.1 SRFLP

The SRFLP is a linear ordering problem considering a set $N = \{1, 2, \ldots, n\}$ of departments in a facility. Each department i has a given length l_i and is connected to all other departments by a traffic intensity c_{ij}. Both the lengths and the traffic intensities are positive integers. It is imposed that $c_{ij} = c_{ji}$ but it is not a modeling restriction since a trip in any direction covers the same distance, the traffic intensities can thus concentrate both directions [52].

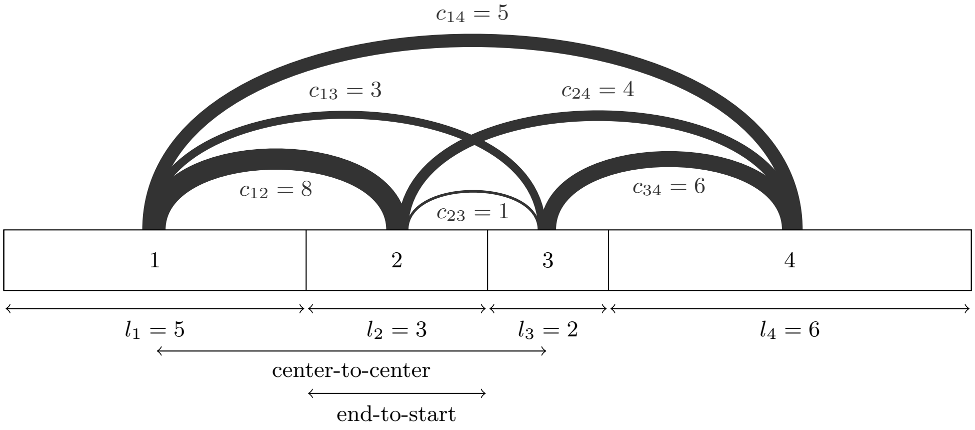


Figure 1 An instance of the SRFLP with 4 departments ordered optimally. The lengths of the departments are noted below them and the pairwise traffic intensities are given on the edges connecting pairs of departments. Center-to-center and end-to-start distances between departments one and three are shown.

A solution to the SRFLP is an ordering of the departments on a line, defined by the bijection $\pi : N \to \{1, 2, \ldots, n\}$. If d^π_{ij} is the center-to-center distance between departments i and j for ordering π, the cost function to minimize is formulated as follows:

$$SRFLP(\pi) = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ i<j}}^{n} c_{ij} d^\pi_{ij}. \tag{1}$$

It is a measure of the total distance traveled by components or products within the facility. Using center-to-center distances implies that we must deal with half department lengths. However, one can notice that wherever a department i is placed with respect to a department j, the center-to-center distance between i and j will be at least $\frac{l_i+l_j}{2}$. This leads to a reformulation that simplifies the coming formulas:

$$SRFLP(\pi) = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ i<j}}^{n} c_{ij} \tilde{d}^\pi_{ij} + K \qquad \text{with} \qquad K = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ i<j}}^{n} c_{ij} \frac{l_i + l_j}{2} \tag{2}$$

where $\tilde{d}^\pi_{ij}$ is the end-to-start distance (see Figure 1) separating departments i and j and K is a constant accounting for all contributions of half department lengths [52].

▶ **Example 1.** Let us illustrate the computation of the objective function on the facility given in Figure 1. We first compute the value of the constant K:

$$\begin{aligned} K &= c_{12}\frac{l_1+l_2}{2} + c_{13}\frac{l_1+l_3}{2} + c_{14}\frac{l_1+l_4}{2} + c_{23}\frac{l_2+l_3}{2} + c_{24}\frac{l_2+l_4}{2} + c_{34}\frac{l_3+l_4}{2} \\ &= 8\frac{5+3}{2} + 3\frac{5+2}{2} + 5\frac{5+6}{2} + 1\frac{3+2}{2} + 4\frac{3+6}{2} + 6\frac{2+6}{2} \\ &= 8 \cdot 4 + 3 \cdot 3.5 + 5 \cdot 5.5 + 1 \cdot 2.5 + 4 \cdot 4.5 + 6 \cdot 4 = 114.5 \end{aligned}$$

and then the cost of the ordering $\pi(i) = i, \forall i \in N$ as shown in Figure 1:

$$\begin{aligned} SRFLP(\pi) &= c_{12}\tilde{d}^\pi_{12} + c_{13}\tilde{d}^\pi_{13} + c_{14}\tilde{d}^\pi_{14} + c_{23}\tilde{d}^\pi_{23} + c_{24}\tilde{d}^\pi_{24} + c_{34}\tilde{d}^\pi_{34} + K \\ &= c_{12} \cdot 0 + c_{13} l_2 + c_{14}(l_2 + l_3) + c_{23} \cdot 0 + c_{24} l_3 + c_{34} \cdot 0 + K \\ &= 8 \cdot 0 + 3 \cdot 3 + 5 \cdot (3+2) + 1 \cdot 0 + 4 \cdot 2 + 6 \cdot 0 + 114.5 = 156.5. \end{aligned}$$

2.2 cSRFLP

The cSRFLP is obtained by adding three types of constraints to the SRFLP:

- *Positioning constraints:* A department is forced to be located at a specific position within the ordering. These constraints are described by a function $position : N \to N \cup \{0\}$ which maps positions to their corresponding department or to 0 if there is no constraint on the position. To simplify the coming equations, we also define the function $department : N \to N \cup \{0\}$ which is the inverse mapping, between departments and positions.
- *Ordering constraints:* These constraints impose that some department must come before another one in the ordering. Formally, the function $predecessors : N \to 2^N$ gives the set of *predecessors* of each department, i.e. all departments that must be placed on the left of the given department.
- *Relation constraints:* Similarly to ordering constraints, relation constraints impose a relative ordering between a pair of departments. In this case, however, the two departments are required to be adjacent in the ordering. The function $previous : N \to N \cup \{0\}$ maps departments to the department that must be placed right before, or to 0 if there is no such constraint.

3 Mixed-Integer Programming Model

In this section, we integrate the constraints of the cSRFLP to the MIP model for the SRFLP, used within the branch-and-cut framework of [4]. This model uses betweenness variables ζ_{ijk} which describe the relative ordering of departments $i, j, k \in N$ in an ordering π:

$$\zeta^{\pi}_{ijk} = \begin{cases} 1, & \text{if } \pi(i) < \pi(k) < \pi(j) \text{ or } \pi(j) < \pi(k) < \pi(i) \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

Using those variables, the objective function can be formulated as follows:

$$SRFLP(\zeta) = \sum_{i \in N} \sum_{\substack{j \in N \\ i<j}} c_{ij} \sum_{k \in N} \zeta_{ijk} l_k + K \tag{4}$$

and is to be minimized under the following constraints:

$$\zeta_{ijk} = \zeta_{jik} \qquad \forall \{i,j,k \mid i<j\} \subseteq N \tag{5}$$

$$\zeta_{ijk} + \zeta_{ikj} + \zeta_{jki} = 1 \qquad \forall \{i,j,k\} \subseteq N \tag{6}$$

$$\zeta_{ijd} + \zeta_{jkd} - \zeta_{ikd} \geq 0 \qquad \forall \{i,j,k,d\} \subseteq N \tag{7}$$

$$\zeta_{ijd} + \zeta_{jkd} + \zeta_{ikd} \leq 2 \qquad \forall \{i,j,k,d\} \subseteq N. \tag{8}$$

Equation (5) follows from the definition of the betweenness variables in Equation (3). Equation (6) states that only one department among i, j, k lies between the two others. Finally, Equations (7) and (8) express the fact that when a department d is placed between departments i and k, then the department d must either lie between departments (a) i and j or (b) j and k, but not both (a) and (b).

We now present how the constraints of the cSRFLP can be integrated in the model. A solution to the original model specifies a relative ordering of the departments. Yet, it does not impose one extremity to the left of the arrangement. As we will need this information in the constraints presented in Section 2.2, we solve this issue by adding two dummy departments L and R. For the cSRFLP, the set of departments is thus defined as $N = \{1, \dots, n\} \cup \{L, R\}$ and departments L and R also obey Equations (3)–(8). We set $l_L = l_R = 0$ and $c_{Li} = c_{iL} = c_{Ri} = c_{iR} = 0, \forall i \in N$ so that the dummy departments have no impact on the objective function. Department L and R are respectively forced on the left and right side of the arrangement by adding the constraints:

$$\zeta_{LRi} = 1 \qquad \forall i \in N \setminus \{L, R\} \tag{9}$$

$$\zeta_{ijL} = 0 \qquad \forall i, j \in N \tag{10}$$

$$\zeta_{ijR} = 0 \qquad \forall i, j \in N. \tag{11}$$

Equation (9) imposes that all other departments are placed between departments L and R. Inversely, Equations (10) and (11) ensure that departments L and R are not placed between any two departments.

We can now write the additional constraints of the model for the cSRFLP:

$$\sum_{k=1}^{n} \zeta_{Lik} = j - 1 \qquad \forall i \in N, position(i) = j \neq 0 \tag{12}$$

$$\sum_{k=1}^{n} \zeta_{iRk} = n - j \qquad \forall i \in N, position(i) = j \neq 0 \tag{13}$$

$$\zeta_{Lij} = 0 \qquad \forall i, j \in N, i \in predecessors(j) \vee i = previous(j) \tag{14}$$

$$\zeta_{Lji} = 1 \qquad \forall i, j \in N, i \in predecessors(j) \vee i = previous(j) \tag{15}$$

$$\zeta_{iRj} = 1 \qquad \forall i, j \in N, i \in predecessors(j) \vee i = previous(j) \tag{16}$$

$$\zeta_{jRi} = 0 \qquad \forall i, j \in N, i \in predecessors(j) \vee i = previous(j) \tag{17}$$

$$\zeta_{ijk} = 0 \qquad \forall i, j \in N, i = previous(j), k \in N \setminus \{i, j\}. \tag{18}$$

Equations (12) and (13) ensure that $j-1$ departments are located on the left of department i and $n-j$ on the right, given that i must be placed at the j-th position. Equations (14)–(17) impose that i is placed between L and j and that j is placed between i and R, when either i is a predecessor of j or i must be placed right before j. Finally, Equation (18) is added for relation constraints to avoid having any departments placed between the two departments involved in the constraint.

4 Dynamic Programming Model

Dynamic programming is a different technique to tackle this problem. Section 4.1 presents an efficient DP model introduced in [50]. We then show in Section 4.2 how the constraints can be incorporated in this model. As a whole, this formulation will be the starting point for our DD-based approach.

4.1 SRFLP

Let us first reformulate the cost function:

$$SRFLP\left(\pi\right) = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ i<j}}^{n} c_{ij} \tilde{d}_{ij}^{\pi} + K = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ \pi(i)<\pi(j)}}^{n} c_{ij} \tilde{d}_{ij}^{\pi} + K \tag{19}$$

$$= \sum_{i=1}^{n} \sum_{\substack{j=1 \\ \pi(i)<\pi(j)}}^{n} c_{ij} \sum_{\substack{k=1 \\ \pi(i)<\pi(k)<\pi(j)}}^{n} l_k + K \tag{20}$$

$$= \sum_{k=1}^{n} l_k \sum_{\substack{i=1 \\ \pi(i)<\pi(k)}}^{n} \sum_{\substack{j=1 \\ \pi(k)<\pi(j)}}^{n} c_{ij} + K. \tag{21}$$

In Equation (19), we use the bijection π to sum over unique pairs of positions instead of unique pairs of departments. We then develop the end-to-start distances $\tilde{d}_{ij}^{\pi}$ in Equation (20), which are equal to the sum of the lengths of departments between i and j in the ordering π. Finally, we reorder the summations in Equation (21). This allows reading the cost function differently: for each department k, we add its length l_k to the distance between pairs of departments (i, j) lying on opposite sides of k and multiply it by the corresponding traffic intensity c_{ij}.

The idea of the DP model is to place the departments one by one on the line from left to right. From Equation (21), it is clear that the individual cost of placing department k at position $\pi(k)$ only depends on the side on which all other departments are located with respect to k. If the state of the DP model is the subset of departments which remain to be placed – called *free* departments from now on, as opposed to *fixed* departments – we can compute this individual cost and recursively find the optimal ordering of each subset of N. Formally, the components of the DP model are:

- The *control variables* $x_j \in D_j$ with $j \in \{0, \ldots, n-1\}$. Variable x_j represents the department placed at position $j+1$ on the line. All variables have the same domain $D_j = N$ since departments can appear anywhere in the ordering.
- The *state space* S which contains all subsets of N. It includes a *root state* $\hat{r} = N$, a *terminal state* $\hat{t} = \emptyset$ and an *infeasible state* $\hat{0}$. The state space is partitioned into the sets $S_0, \ldots, S_n$ where S_j contains all states with j variables assigned.
- The set of *transition functions* $t_j : S_j \times D_j \to S_{j+1}$ for $j = 0, \ldots, n-1$ which rule the transition between the states of consecutive stages:

$$t_j\left(s^j, x_j\right) = \begin{cases} s^j \setminus \{x_j\}, & \text{if } x_j \in s^j \\ \hat{0}, & \text{otherwise.} \end{cases} \tag{22}$$

- The set of *transition cost functions* $h_j : S_j \times D_j \to \mathbb{R}$ for $j = 0, \ldots, n-1$ which associate a value to each transition:

$$h_j\left(s^j, x_j\right) = \begin{cases} l_{x_j} \sum_{i \in \overline{s}^j} \sum_{k \in s^j \setminus \{x_j\}} c_{ik}, & \text{if } x_j \in s^j \\ 0, & \text{otherwise.} \end{cases} \tag{23}$$

  This formula immediately follows from Equation (21) since $\overline{s}^j$ – the complement of s^j – contains fixed departments placed before position j and $s^j \setminus \{x_j\}$ contains free departments, which will be placed after position j.
- The *root value* $v_r = K$ from Equation (2).

To solve an instance of the SRFLP using this DP model, one needs to apply the following recurrence:

$$\begin{aligned} &\min\ \hat{f}(x) = v_r + \sum_{j=0}^{n-1} h_j\left(s^j, x_j\right) \\ &\text{subject to } s^{j+1} = t_j\left(s^j, x_j\right), \text{ for all } x_j \in D_j, j = 0, \ldots, n-1 \\ &\qquad s^j \in S_j, j = 0, \ldots, n. \end{aligned} \tag{24}$$

Speeding up the computation of transition costs. We also store in the states an array containing the *cut values* of each free department: the sum of all traffic intensities from the fixed departments and each free department. It allows to reduce the computational complexity of the transition costs from $\mathcal{O}\left(n^2\right)$ to $\mathcal{O}(n)$ and will also be useful when designing a lower bound in Section 6.2. For a state s^j and each department $i \in N$, we define:

$$s^j_{cut}[i] = \begin{cases} \sum_{j \in \overline{s}^j} c_{ij}, & \text{if } i \in s^j \\ 0, & \text{otherwise} \end{cases} \tag{25}$$

which can be updated in $\mathcal{O}(n)$ during a transition $t_j\left(s^j, x_j\right)$:

$$s^{j+1}_{cut}[i] = \begin{cases} s^j_{cut}[i] + c_{ix_j}, & \text{if } i \in s^j \setminus \{x_j\} \\ 0, & \text{otherwise} \end{cases} \tag{26}$$

and the transition costs become:

$$h_j\left(s^j, x_j\right) = \begin{cases} l_{x_j} \sum_{i \in s^j \setminus \{x_j\}} s^j_{cut}[i], & \text{if } x_j \in s^j \\ 0, & \text{otherwise.} \end{cases} \tag{27}$$

▶ **Example 2.** Considering the instance shown on Figure 1, we compute the cut values for the state $s = \{3, 4\}$. We have that $s_{cut}[1] = s_{cut}[2] = 0$ since departments 1 and 2 are already placed. For the free departments, we apply Equation (25) and obtain: $s_{cut}[3] = c_{13} + c_{23} = 3 + 1 = 4$ and $s_{cut}[4] = c_{14} + c_{24} = 5 + 4 = 9$.

4.2 cSRFLP

Adding constraints to the DP model is done through the predicates $valid_j : S_j \times D_j \to \{true, false\}$ for $j = 0, \ldots, n-1$. They are used in the transition functions to filter out infeasible solutions:

$$t_j\left(s^j, x_j\right) = \begin{cases} s^j \setminus \{x_j\}, & \text{if } x_j \in s^j \wedge valid_j(s^j, x_j) \\ \hat{0}, & \text{otherwise.} \end{cases} \tag{28}$$

For clarity, we split the predicates $valid_j$ into several conditions, corresponding each to a specific constraint:

$$valid_j(s^j, x_j) = p_j(s^j, x_j) \wedge o_j(s^j, x_j) \wedge r_j(s^j, x_j) \tag{29}$$

with p_j, o_j and r_j concerning respectively positioning, ordering and relation constraints:

$$p_j(s^j, x_j) = (position(x_j) = 0 \wedge department(j+1) = 0) \vee position(x_j) = j + 1 \tag{30}$$

$$o_j(s^j, x_j) = predecessors(x_j) \subseteq \overline{s}^j \tag{31}$$

$$r_j(s^j, x_j) = (previous(x_j) = 0 \wedge \nexists k \in s^j : previous(k) \in \overline{s}^j) \vee previous(x_j) \in \overline{s}^j. \tag{32}$$

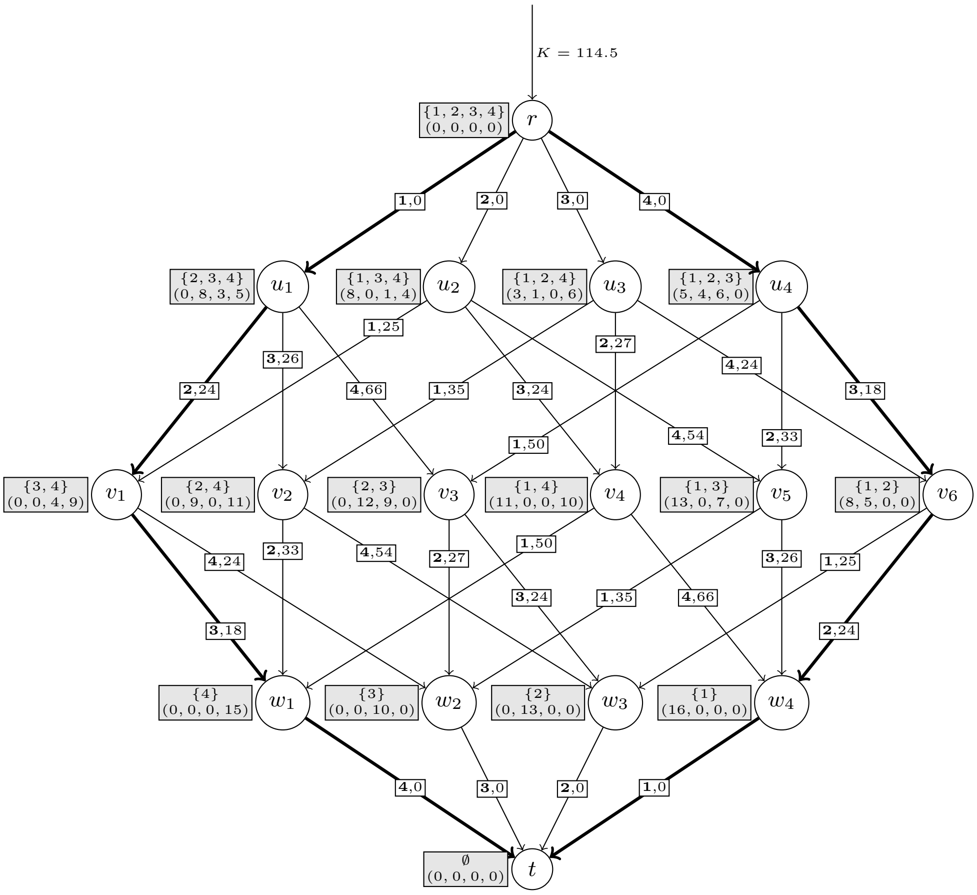


Figure 2 The exact DD associated with the instance shown on Figure 1. Arcs are annotated with their label (in bold) and cost. Next to each node, a gray box contains the corresponding state: the set of free departments and the cut values. Arcs in bold are part of an optimal solution.

In Equation (30), p_j checks that either department x_j and position $j + 1$ are both unconstrained, or that department x_j is constrained to be at position $j + 1$. As explained previously, the j-th transition decides which department is placed at position $j + 1$. For ordering constraints, Equation (31) verifies that all predecessors of department x_j have already been placed. The predicates r_j for relation constraints are slightly more complicated. Either x_j has no relation constraint, then it can only be placed if no other free department has a relation constraint with a fixed department, or x_j has a relation constraint and $previous(x_j)$ must be a fixed department.

5 Decision Diagram Representation

This section explains how a DP model can be used to derive DDs. A *weighted decision diagram* is a graphical structure which encodes a set of solutions to a discrete optimization problem $\mathcal{P}$. Formally, it is represented by a layered directed acyclic graph $B = (U, A, d, v, \sigma)$ where U is the set of nodes, A is the set of arcs. The set of nodes is partitioned into layers $L_0, \ldots, L_n$. In particular, layers L_0 and L_n contain only one node, respectively the root node r and the terminal node t. Each node is mapped to a state by the function σ. An arc

$a \in A$ connects a node in a layer L_j to a node in the next layer L_{j+1}. Its label $d(a) \in D_j$ represents the assignment of value $d(a)$ to variable x_j and $v(a)$ denotes its length. As a result, each path $p = \left(a^{(0)}, \dots, a^{(n-1)}\right)$ from r to t is a complete assignment of the variables, with $x_j = d\left(a^{(j)}\right)$, and has a total length of $v(p) = v_r + \sum_{j=0}^{n-1} v\left(a^{(j)}\right)$. The set of all $r - t$ paths of B encodes the set of possible assignments $\mathrm{Sol}(B)$. In an *exact decision diagram*, the length of each $r - t$ path is equal to the objective function value of the corresponding assignment and $\mathrm{Sol}(B) = \mathrm{Sol}(\mathcal{P})$. Thus, the resolution of discrete optimization problems is reduced to a shortest-path problem on a directed acyclic graph $f(x^*) = v^*(B)$.

The *size* $|B|$ of a decision diagram is the number of nodes it contains in all layers. Its *width* is given by $\max_j |L_j|$, where $|L_j|$ is the *width* of layer j. Arcs leaving a same node always have different labels, so every node $u \in L_j$ has a maximum out-degree of $|D_j|$. A *binary decision diagram* encodes binary variables only, as opposed to *multi-valued decision diagrams* in the general case [38].

Using a DP model, an exact DD can be built layer by layer starting with the first layer L_0 containing the root node r associated to the root state $\hat{r}$. From a layer L_j, we then fill L_{j+1} with all nodes corresponding to *distinct* feasible states which can be reached from any state in L_j. For each of these transitions, we add an arc from the node in L_j to the one in L_{j+1} and its length is given by the transition cost.

▶ **Example 3.** The exact DD for the instance shown in Figure 1 is illustrated in Figure 2. The size of this DD is 16 and its width is 6. On the left side of the DD, the path in bold is an optimal solution. It corresponds to the ordering displayed in Figure 1 and its length is equal to $114.5 + 0 + 24 + 18 + 0 = 156.5$ as computed in Example 1.

6 Branch-and-Bound

Although DP formulations tend to represent problems in a compact manner, it is usually intractable to generate exact DDs for combinatorial problems as the size of the state space can grow exponentially with the number of variables. An adapted branch-and-bound algorithm exploiting DDs (B&B-DD) was presented in [13] with the potential to solve larger instances to optimality. The algorithm successively explores subproblems corresponding to nodes in the exact DD of the problem. As in classical branch-and-bound, two ingredients are used: a primal upper bound heuristic to discover good feasible solutions, and a lower bound procedure allowing to prune the nodes with a lower bound larger than the best so far solution. The major idea of B&B-DD is to limit the width of the DDs to obtain these two ingredients. The primal heuristic is obtained by discarding nodes of the DD to respect the width limit while the lower bound procedure consists in discovering the best path in a relaxed DD obtained by merging nodes. The nodes of B&B-DD are expanded using a classical best-first-search.

Preliminary experiments convinced us to slightly deviate from the generic B&B-DD framework and specialize it for the SRFLP in order to be competitive with state-of-the-art approaches. We use a problem specific lower bound rather than the state-merging procedure, as well as a breadth-first search allowing to better exploit the recursive structure of the problem. The lower bound and the custom search are detailed in the next sections.

6.1 Primal Upper Bound Heuristic

As explained in Section 6, we rely on restricted DDs to generate good feasible solutions starting from a given node of the exact DD. To obtain a restricted DD, it is sufficient to remove nodes of a layer when its width exceeds a given maximum width. The nodes and

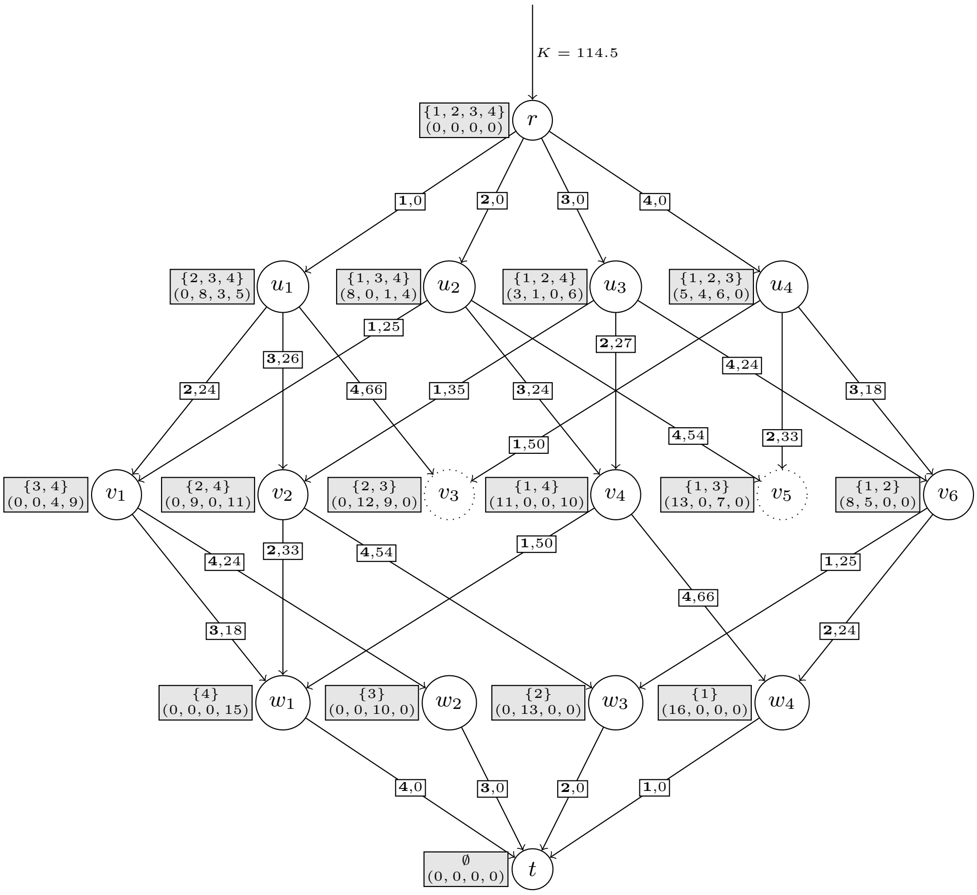


Figure 3 A restricted DD associated with the instance shown on Figure 1 and built with a maximum width of 4. Nodes in dotted circles have been removed from the layer.

arcs which remain in the DD are not modified and thus correspond to feasible solutions. A heuristic is used to select nodes to remove from a layer and attempts to identify nodes leading to the poorer quality solutions. Restricted DDs also allow to retrieve the set of subproblems which need to be explored next. In this paper, this set is computed as the set of direct successors of the initial node of the restricted DD.

▶ **Example 4.** Figure 3 shows a restricted DD built for the instance displayed in Figure 1 with a maximum width of 4. The third layer exceeded the maximum width so the nodes v_3 and v_5 have been removed.

6.2 Lower Bound

In [13], a relaxed DD is used to compute a single lower bound at a given node. Based on this lower bound, we decide whether to enqueue or prune the open subproblems. Recently, [22] suggested that we could attach a different lower bound to each node to be added to the branch-and-bound queue. In our approach, this lower bound is based on a heuristic *rough lower bound* (RLB) which can be computed swiftly for any node. As described in [22], the RLB can also be used to skip nodes during the compilation of restricted DDs.

In order to derive the RLB from a node u, the next theorem shows that the cost to optimally complete the partial solution of node u can be decomposed in two terms: one solely involving the free departments and the other one involving the cost between free and fixed departments.

▶ **Theorem 5.** *Given a node u and its state $\sigma(u) = s$, let $\pi^*|_u$ be the best ordering one can obtain when crossing node u. For conciseness, we set $\pi = \pi^*|_u$. We have the equivalence:*

$$SRFLP(\pi) - v^*(u) = \underbrace{\sum_{i \in s} \sum_{\substack{j \in s \\ \pi(i)<\pi(j)}} c_{ij} \sum_{\substack{k \in s \\ \pi(i)<\pi(k)<\pi(j)}} l_k}_{free\ departments\ layout\ cost} + \underbrace{\sum_{j \in s} s_{cut}[j] \sum_{\substack{k \in s \\ \pi(k)<\pi(j)}} l_k}_{cost\ w.r.t.\ fixed\ departments} . \tag{33}$$

Those two terms of Equation (33) cannot be evaluated exactly in a cheap way as this would be as difficult as solving the original problem. Nonetheless one can compute an efficient lower bound for each term independently. For a node u, the value of the RLB is given by:

$$RLB(u) = \begin{cases} \infty, & \text{if } \sigma(u) = \hat{0} \\ LB_{edge}(u) + LB_{cut}(u), & \text{otherwise,} \end{cases} \tag{34}$$

where $LB_{edge}(u)$ is a lower bound on the free departments layout cost and $LB_{cut}(u)$ is a lower bound on the cost induced by the cut values of free departments.

6.2.1 Free departments layout cost

The first lower bound LB_{edge} is an under-approximation of the internal layout cost of free departments. Given a subset of departments, we compute a lower bound on the cost of its optimal layout by multiplying each pairwise traffic intensity by an optimistic distance. If we must place n departments on a line, $n-k$ pairs of departments will have $k-1$ departments between them (see Figure 1). In order to under-approximate the layout cost, we greedily multiply the highest traffic intensities by the smallest distance possible. Since we cannot assume any particular ordering of the free departments, the distances between pairs of free departments are unknown. Still, we can compute lower bounds on those distances if we sort the free departments by increasing length and assume that a separation of k departments will be formed by the k shortest departments. This lower bound can be seen as a generalization of the Edges method [49] designed for the MinLA.

In practice, a list containing all pairwise traffic intensities in decreasing weight order is precomputed, as stated by the precondition of Algorithm 1. The same is done for the department lengths. We then only need to traverse those lists and multiply each traffic intensity value by the adequate cumulative length. The complexity of the algorithm is $\mathcal{O}\left(n^2\right)$ since there are $\frac{n(n-1)}{2}$ pairs in total.

▶ **Example 6.** Let us illustrate the computation of this lower bound on the root node of the DD in Figure 2. We first create the list of traffic intensities sorted decreasingly: $edge = [c_{12} = 8, c_{34} = 6, c_{14} = 5, c_{24} = 4, c_{13} = 3, c_{23} = 1]$ and the list of free department lengths sorted increasingly: $length = [l_3 = 2, l_2 = 3, l_1 = 5, l_4 = 6]$. There are 3 pairs of departments with 0 departments in between, 2 pairs with 1 department in between and 1 pair with 2 departments in between.

$$\begin{aligned} LB_{edge}(r) &= 0 \cdot c_{12} + 0 \cdot c_{34} + 0 \cdot c_{14} + l_3 c_{24} + l_3 c_{13} + (l_3 + l_2) c_{23} \\ &= 0 \cdot 8 + 0 \cdot 6 + 0 \cdot 5 + 2 \cdot 4 + 2 \cdot 3 + (2+3) \cdot 1 = 19 \end{aligned}$$

Algorithm 1 Computation of $LB_{edge}(u)$.

```
Require: $edge = sorted_{\geq}(\{\langle c : c_{ij}, dep1 : i, dep2 : j\rangle \mid 1 \leq i < j \leq n\})$
    and $length = sorted_{\leq}(\{\langle l : l_i, dep : i\rangle \mid 1 \leq i \leq n\})$
$s \leftarrow \sigma(u), lb \leftarrow 0, cumul\_l \leftarrow 0, i \leftarrow 1, j \leftarrow 1$
for $k \leftarrow 1$ to $|s| - 1$ do
    for $l \leftarrow 1$ to $k$ do
        while $edge[i].dep1 \notin s \vee edge[i].dep2 \notin s$ do
            $i \leftarrow i + 1$
        $lb \leftarrow lb + cumul\_l \cdot edge[i].c$
        $i \leftarrow i + 1$
    while $length[j].dep \notin s$ do
        $j \leftarrow j + 1$
    $cumul\_l \leftarrow cumul\_l + length[j].l$
    $j \leftarrow j + 1$
return $lb$
```

6.2.2 Cost with respect to fixed departments

The second term of the RLB is related to the cut values of free departments and a lower bound is given by the *first-generation bound* described in [52]. Given a department i placed first on the line, the minimum total cost with respect to i is defined as:

$$MTC(i) = \min_{\pi} \sum_{\substack{j=1 \\ i \neq j}}^{n} c_{ij} \sum_{\substack{k=1 \\ \pi(k) < \pi(j)}}^{n} l_k \tag{35}$$

and Lemma 7 tells us how to find the optimal arrangement π.

▶ **Lemma 7.** *Suppose that department i is placed in first position on the line. For every other department j compute the cost-to-length ratio $r_j = \frac{c_{ij}}{l_j}$. The optimal arrangement, which yields $MTC(i)$ is obtained by ordering the departments according to decreasing values of this ratio r_j, the department with the greatest r_j being adjacent to i.*

This lower bound can also be used when several departments are placed in the leftmost positions on the line. We only need to consider all fixed departments as a single department connected to free departments with traffic intensities given by the respective cut values, exactly as in the second term of Equation (33). As the free departments need to be sorted by decreasing cut-to-length ratios, the time complexity of this lower bound is $\mathcal{O}(n \log(n))$.

▶ **Example 8.** We compute the lower bound for the node u_1 of the DD shown in Figure 2, with $\sigma(u_1) = s$. The departments are first sorted as follows:

$$order = \left[\frac{s_{cut}[2]}{l_2} = \frac{8}{3}, \frac{s_{cut}[3]}{l_3} = \frac{3}{2}, \frac{s_{cut}[4]}{l_4} = \frac{5}{6}\right].$$

We then compute the lower bound as the total cost with respect to all fixed departments:

$$\begin{aligned} LB_{cut}(u_1) &= 0 \cdot s_{cut}[2] + l_2 s_{cut}[3] + (l_2 + l_3) s_{cut}[4] \\ &= 0 \cdot 8 + 3 \cdot 3 + (3 + 2) \cdot 5 = 34. \end{aligned}$$

6.2.3 Refining the lower bound

In Section 6.1, we mentioned that for a node $u^{j-1} \in L_{j-1}$, its direct successors are added to the branch-and-bound queue. During the compilation of a restricted DD, not only we generate these successors in layer L_j but we also create all nodes which we can reach in layer L_{j+1}. As a result, we can compute a tighter lower bound for each node of L_j by taking advantage of the RLB values of its successors:

$$LB\left(u^j\right) = v^*(u^j) + \min_{x_j \in D_j} \left(h_j\left(\sigma\left(u^j\right), x_j\right) + RLB\left(t_j\left(\sigma\left(u^j\right), x_j\right)\right)\right). \tag{36}$$

▶ **Example 9.** Given the restricted DD shown in Figure 3, the local lower bound of node u_1 is computed as follows: $LB(u_1) = 0 + \min(24 + RLB(v_1), 26 + RLB(v_2), 66 + RLB(v_3))$.

6.3 A Breadth-First Branch-and-Bound

In the DP model of the SRFLP, a state s^j at level j is the successor of exactly j different states. More generally, it can be reached by as many as j! different paths since any permutation of the departments could be a valid solution. The classical branch-and-bound algorithm always explores the most promising node first with a best-first-search strategy i.e. the one with the lowest lower bound or lowest shortest-path length. In the context of B&B-DD, nodes with a same state could be enqueued and explored multiple times during the algorithm. Preliminary experiments showed that this was often the case for the cSRFLP. This can be avoided by only exploring a complete layer before considering the next one. Therefore we suggest exploring the most promising node of what we call the *lowest active layer* (LAL) – the layer containing nodes of the queue with the least variables assigned. By doing so, all ancestors of the chosen node must have already been explored. It also ensures that at most one node associated with any state of the model will be inserted in the queue. The only adjustment to make is to maintain an additional data structure keeping track of all nodes in the queue. In that data structure, exactly like in any layer of a DD, we identify nodes by their state and keep in memory the path with shortest length to each state. We then only add one node to the queue for each state, and otherwise update the shortest-path leading to it. Our strategy is thus equivalent to a breadth-first-search in the exact DD but enhanced by pruning mechanisms.

This whole procedure is described by Algorithm 2. The index of the LAL is denoted l and increases throughout the execution of the algorithm. The branch-and-bound queue is split between Q_l and Q_{l+1} which respectively contain open nodes of layers l and $l+1$. For each layer L_j, M_j is a map containing the node with the shortest path to each state of the level j. It is used in lines 17-26 to avoid adding multiple nodes in the branch-and-bound queue for the same state. The loop of line 8 can be parallelized, which is a key asset of B&B-DD [12, 21]. Each thread is responsible for developing a different restricted DD at line 13 and synchronization happens when queues, maps and the incumbent solution need to be updated.

7 Computational Experiments

In this section, we draw a comparison between the existing techniques to solve the cSRFLP to optimality. Namely, the MIP model from [44], the MIP model introduced in [4] and extended in Section 3 and the DD-based approach presented throughout the rest of the paper. In the following, they are respectively referred to as Liu, Amaral and DD. The MIP models were implemented and evaluated using Gurobi version 9.1.2 [24]. Concerning the DD approach, it was implemented in C++ and the code was largely based on DDO [21], a Rust library for DD-based discrete optimization. The heuristics selected are the following:

- *Maximum width:* We use fixed-width DDs for all experiments. To that end, we experimentally determined that a narrow maximum width of 3 leads to the best performance. It may seem very small but as explained in Section 6, the lower bound of a node is exclusively based on RLB values of its child nodes. As a result, the quality of the lower bounds does not depend on the maximum width of the DDs. Moreover, we observed that we were able to find very good solutions early in the search anyway.

Algorithm 2 The breadth-first branch-and-bound algorithm. *select_node* is a heuristic used to select the most promising node of the queue.

```
$v(r) \leftarrow v_r$ // root node value
$Q_0 \leftarrow \{r\}$ // queue for layer 0
$M_0 \leftarrow \{\sigma(r) : r\}$ // map for layer 0
$UB \leftarrow \infty$
for $l = 0$ to $n - 1$ do // $l$ is the lowest active layer
    $Q_{l+1} \leftarrow \emptyset$ // queue for layer $l + 1$
    $M_{l+1} \leftarrow \emptyset$ // map for layer $l + 1$
    while $Q_l \neq \emptyset$ do
        $u \leftarrow select\_node(Q_l)$
        $Q_l \leftarrow Q_l \setminus \{u\}$
        if $LB(u) \geq UB$ then
            continue
        $\overline{B} \leftarrow Restricted(u)$
        for all $u' \in L_n$ of $\overline{B}$ do // update best solution
            if $v(u') < UB$ then
                $UB \leftarrow v(u')$
        for all $u' \in L_{l+1}$ of $\overline{B}$ do // enqueue successors of $u$ and update $M_{l+1}$
            if $LB(u') < UB$ then
                if $M_{l+1}.contains(\sigma(u'))$ then // this state is already in the queue and map
                    if $v(u') < v(M_{l+1}[\sigma(u')])$ then // update only if the value is improved
                        $Q_{l+1} \leftarrow Q_{l+1} \setminus \{M_{l+1}[\sigma(u')]\}$
                        $Q_{l+1} \leftarrow Q_{l+1} \cup \{u'\}$
                        $M_{l+1}[\sigma(u')] \leftarrow u'$
                else // this state is not in the queue and map
                    $M_{l+1}[\sigma(u')] \leftarrow u'$
                    $Q_{l+1} \leftarrow Q_{l+1} \cup \{u'\}$
return $UB$
```

- *Variable ordering:* The vertices must be placed from left to right on the line in the DP model so it is imposed for this formulation of the problem.
- *Search node selection:* Nodes with the smallest lower bound in the branch-and-bound queue are explored first in the branch-and-bound.
- *Node selection for restriction:* When the size of a layer exceeds the maximum width of the DD, we delete the nodes with the largest RLB values.

The instances used in the experiments are classical SRFLP instances taken from [2, 3, 8, 31, 52] with up to 25 departments. We then created admissible sets of constraints for each problem size:

- constraint sets with $2, 4, 6, 8$ and 10 positioning, ordering or relation constraints.
- constraint sets with $0, 2, 4, 6, 8$ and 10 constraints of each type.

For each of these scenarios, 5 different random sets of constraints were generated, except for the case with no constraints. Note that an instance with n departments can not have more than n positioning constraints, and that similar limits exist for the other types of constraints, we thus have up to 101 sets of constraints for each problem size. A link to the source code along with all the benchmark instances is given in the supplementary material. All experiments were performed on a machine with two Intel Xeon E5-2640 (2.6GHz) processors.

The three algorithms were executed on all combinations of instances and constraints with a time limit of 5 hours for each. The first row of Figure 4 shows the cumulative number of instances solved by each algorithm over time while the second row shows the mean ratio between the runtimes of each instance and its corresponding unconstrained instance, with respect to the number of constraints of each given type. Our first observation is that the two models presented in this paper clearly outperform the one from [44], which fails to solve most of the instances under the time limit regardless of the type of constraints

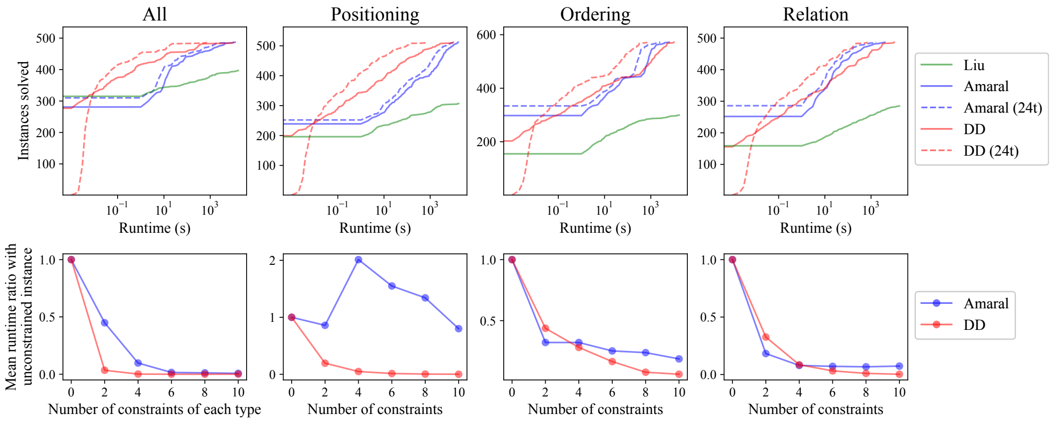


Figure 4 Number of instances solved by each algorithm for the different types of constraints, and mean ratio between the runtime of each constrained instance and the runtime of the corresponding unconstrained instance, with respect to the number of constraints of each type.

applied. Next, even if Amaral and DD both succeed in solving all instances, they have different behaviors depending on the type of scenario. From the graphs of the second row, we notice that the more constraints we add, the faster the DD approach gets. The constraints in the DD formulation are indeed handled very efficiently because all infeasible solutions are automatically pruned in the transition functions, which results in a smaller DP graph to explore. The same cannot be said about Amaral, since instances with between 4 and 8 positioning constraints take more time to solve than their corresponding unconstrained instance on average. This is probably because positioning constraints are modeled with a sum of n variables on the left side of an equality. On the contrary, ordering and relation constraints are modeled very naturally in Amaral because it uses relative ordering variables. Adding these types of constraints thus tightens the model and reduces the execution time. It allows Amaral to solve hard instances with ordering and relation constraints slightly faster than DD. However, DD is the first to solve all instances when using 24 threads and seems to benefit the most from parallelization.

8 Conclusion

In this paper, two novel exact models for the cSRFLP have been presented: an extension of the MIP model from [4] for the SRFLP and a DD-based approach starting from the DP model of [50]. The computational experiments have shown that they greatly improve on the performance of the only MIP model introduced in the literature to the best of our knowledge. Both models have their benefits, the DD approach incorporates the three types of constraints very efficiently, especially positioning constraints, and parallelizes better. On the other hand, the MIP model integrates ordering and relation constraints very well and can be easily implemented with any MIP solver. The DD approach can surely be improved in the future, for instance by taking the constraints into account within the lower bounds. It would also be interesting to combine the strengths of our two approaches.

References

1 Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(06):509–516, 1978.

2 André R. S. Amaral. On the exact solution of a facility layout problem. *European Journal of Operational Research*, 173(2):508–518, 2006.

3 André R. S. Amaral. An exact approach to the one-dimensional facility layout problem. *Operations Research*, 56(4):1026–1033, 2008.

4 André R. S. Amaral. A new lower bound for the single row facility layout problem. *Discrete Applied Mathematics*, 157(1):183–190, 2009.

5 André R. S. Amaral and Adam N. Letchford. A polyhedral approach to the single row facility layout problem. *Mathematical programming*, 141(1-2):453–477, 2013.

6 Henrik R. Andersen, Tarik Hadžić, John N. Hooker, and Peter Tiedemann. A constraint store based on multivalued decision diagrams. In *International Conference on Principles and Practice of Constraint Programming*, pages 118–132. Springer, 2007.

7 Miguel F. Anjos, Andrew Kennings, and Anthony Vannelli. A semidefinite optimization approach for the single-row layout problem with unequal dimensions. *Discrete Optimization*, 2(2):113–122, 2005.

8 Miguel F. Anjos and Anthony Vannelli. Computing globally optimal solutions for single-row layout problems using semidefinite programming and cutting planes. *INFORMS Journal on Computing*, 20(4):611–617, 2008.

9 Miguel F. Anjos and Ginger Yen. Provably near-optimal solutions for very large single-row facility layout problems. *Optimization Methods & Software*, 24(4-5):805–817, 2009.

10 Bernd Becker, Markus Behle, Friedrich Eisenbrand, and Ralf Wimmer. Bdds in a branch and cut framework. In *International Workshop on Experimental and Efficient Algorithms*, pages 452–463. Springer, 2005.

11 David Bergman, Andre A. Cire, Willem-Jan van Hoeve, and John N. Hooker. Optimization bounds from binary decision diagrams. *INFORMS Journal on Computing*, 26(2):253–268, 2014.

12 David Bergman, Andre A. Cire, Ashish Sabharwal, Horst Samulowitz, Vijay Saraswat, and Willem-Jan van Hoeve. Parallel combinatorial optimization with decision diagrams. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 351–367. Springer, 2014.

13 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.

14 David Bergman, Andre A. Cire, Willem-Jan van Hoeve, and Tallys Yunes. Bdd-based heuristics for binary optimization. *Journal of Heuristics*, 20(2):211–234, 2014.

15 Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

16 Margarita P. Castro, Andre A. Cire, and J. Christopher Beck. An mdd-based lagrangian approach to the multicommodity pickup-and-delivery tsp. *INFORMS Journal on Computing*, 32(2):263–278, 2020.

17 Andre A. Cire and Willem-Jan van Hoeve. Multivalued decision diagrams for sequencing problems. *Operations Research*, 61(6):1411–1428, 2013.

18 Dilip Datta, André R. S. Amaral, and José R. Figueira. Single row facility layout problem using a permutation-based genetic algorithm. *European Journal of Operational Research*, 213(2):388–394, 2011.

19 Zvi Drezner. A heuristic procedure for the layout of a large number of facilities. *Management Science*, 33(7):907–915, 1987.

20 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Books in mathematical series. W. H. Freeman, 1979.

21 Xavier Gillard, Pierre Schaus, and Vianney Coppé. Ddo, a generic and efficient framework for mdd-based optimization. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI-20)*, pages 5243–5245, 2020.

22 Xavier Gillard, Pierre Schaus, Vianney Coppé, and André A. Cire. Improving the filtering of branch-and-bound mdd solver. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 2021.

23 Jian Guan and Geng Lin. Hybridizing variable neighborhood search with ant colony optimization for solving the single row facility layout problem. *European Journal of Operational Research*, 248(3):899–909, 2016.

24 LLC Gurobi Optimization. Gurobi optimizer reference manual, 2022. URL: `https://www.gurobi.com`.

25 Gary D. Hachtel and Fabio Somenzi. A symbolic algorithms for maximum flow in 0-1 networks. *Formal Methods in System Design*, 10(2):207–219, 1997.

26 Tarik Hadžić and John N. Hooker. Postoptimality analysis for integer programming using binary decision diagrams. Technical report, Carnegie Mellon University, 2006.

27 Tarik Hadžić and John N. Hooker. Cost-bounded binary decision diagrams for 0-1 programming. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 84–98. Springer, 2007.

28 Kenneth M. Hall. An r-dimensional quadratic placement algorithm. *Management science*, 17(3):219–229, 1970.

29 Sunderesh S. Heragu and Attahiru Sule Alfa. Experimental analysis of simulated annealing based algorithms for the layout problem. *European Journal of Operational Research*, 57(2):190–202, 1992.

30 Sunderesh S. Heragu and Andrew Kusiak. Machine layout problem in flexible manufacturing systems. *Operations research*, 36(2):258–268, 1988.

31 Sunderesh S. Heragu and Andrew Kusiak. Efficient models for the facility layout problem. *European Journal of Operational Research*, 53(1):1–13, 1991.

32 Samid Hoda, Willem-Jan van Hoeve, and John N. Hooker. A systematic approach to mdd-based constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, pages 266–280. Springer, 2010.

33 John N. Hooker. Improved job sequencing bounds from decision diagrams. In *International Conference on Principles and Practice of Constraint Programming*, pages 268–283. Springer, 2019.

34 Alan J. Hu. *Techniques for efficient formal verification using binary decision diagrams.* PhD thesis, Stanford University, Department of Computer Science, 1995.

35 Philipp Hungerländer and Franz Rendl. A computational study and survey of methods for the single-row facility layout problem. *Computational Optimization and Applications*, 55(1):1–20, 2013.

36 Philipp Hungerländer and Franz Rendl. Semidefinite relaxations of ordering problems. *Mathematical Programming*, 140(1):77–97, 2013.

37 Zahnupriya Kalita and Dilip Datta. A constrained single-row facility layout problem. *The international journal of advanced manufacturing technology*, 98(5):2173–2184, 2018.

38 Timothy Kam. Multi-valued decision diagrams: Theory and applications. *Multiple-Valued Logic*, 4(1):9–62, 1998.

39 Richard M. Karp and Michael Held. Finite-state processes and dynamic programming. *SIAM Journal on Applied Mathematics*, 15(3):693–718, 1967.

40 Ravi Kothari and Diptesh Ghosh. An efficient genetic algorithm for single row facility layout. *Optimization Letters*, 8(2):679–690, 2014.

41 K. Ravi Kumar, George C. Hadjinicola, and Ting-li Lin. A heuristic procedure for the single-row facility layout problem. *European Journal of Operational Research*, 87(1):65–73, 1995.

42 Yung-Te Lai, Massoud Pedram, and Sarma B. K. Vrudhula. EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):959–975, 1994.

43 C.-Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, 1959.

44 Silu Liu, Zeqiang Zhang, Chao Guan, Lixia Zhu, Min Zhang, and Peng Guo. An improved fireworks algorithm for the constrained single-row facility layout problem. *International Journal of Production Research*, 59(8):2309–2327, 2021.

45 Elsa Loekito, James Bailey, and Jian Pei. A binary decision diagram based approach for mining frequent subsequences. *Knowledge and Information Systems*, 24(2):235–268, 2010.

46 Robert Love and Jsun Wong. On solving a one-dimensional space allocation problem with integer programming. *INFOR: Information Systems and Operational Research*, 14(2):139–143, 1976.

47 Gintaras Palubeckis. Single row facility layout using multi-start simulated annealing. *Computers & Industrial Engineering*, 103:1–16, 2017.

48 Guillaume Perez and Jean-Charles Régin. Efficient operations on mdds for building constraint programming models. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI-15)*, pages 374–380, 2015.

49 Jordi Petit. Experiments on the minimum linear arrangement problem. *Journal of Experimental Algorithmics*, 8, 2003.

50 Jean-Claude Picard and Maurice Queyranne. On the one-dimensional space allocation problem. *Operations Research*, 29(2):371–391, 1981.

51 Hamed Samarghandi and Kourosh Eshghi. An efficient tabu algorithm for the single row facility layout problem. *European Journal of Operational Research*, 205(1):98–105, 2010.

52 Donald M. Simmons. One-dimensional space allocation: an ordering algorithm. *Operations Research*, 17(5):812–826, 1969.

53 J. K. Suryanarayanan, Bruce L. Golden, and Qi Wang. A new heuristic for the linear placement problem. *Computers & Operations Research*, 18(3):255–262, 1991.

54 Willem-Jan van Hoeve. Graph coloring lower bounds from decision diagrams. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 405–418. Springer, 2020.

55 Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Compact-mdd: Efficiently filtering (s) mdd constraints with reversible sparse bit-sets. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*, pages 1383–1389, 2018.

56 Ingo Wegener. Branching programs and binary decision diagrams: Theory and applications. *Discrete Applied Mathematics*, 2000.

Constraint Acquisition Based on Solution Counting

Christopher Coulombe ✉
Université Laval, Québec, Canada

Claude-Guy Quimper ✉ 🏠
Université Laval, Québec, Canada

Abstract

We propose CABSC, a system that performs Constraint Acquisition Based on Solution Counting. In order to learn a Constraint Satisfaction Problem (CSP), the user provides positive examples and a Meta-CSP, i.e. a model of a combinatorial problem whose solution is a CSP. This Meta-CSP allows listing the potential constraints that can be part of the CSP the user wants to learn. It also allows stating the parameters of the constraints, such as the coefficients of a linear equation, and imposing constraints over these parameters. The CABSC reads the Meta-CSP using an augmented version of the language MiniZinc and returns the CSP that accepts the fewest solutions among the CSPs accepting all positive examples. This is done using a branch and bound where the bounding mechanism makes use of a model counter. Experiments show that CABSC is successful at learning constraints and their parameters from positive examples.

2012 ACM Subject Classification Computing methodologies → Modeling methodologies

Keywords and phrases Constraint acquisition, CSP, Model counting, Solution counting

Digital Object Identifier 10.4230/LIPIcs.CP.2022.15

1 Introduction

Constraint solvers are used to solve complex combinatorial problems. They require an expert to model the problem using the constraints available in the solver. The model creation is a crucial step, but is often time-consuming. One way to save time to the expert is to suggest a model based on sample solutions. For instance, a hospital that wants to automatize the creation of their work schedules for its staff might provide to the experts previous schedules. Assisted with software, the expert wants to discover what constraint generated the examples. While some of these constraints are already known and even written on legal documents, there are as important constraints that are not written but are part of the work culture. These are the constraints for which a constraint acquisition software becomes handy.

When two constraints are candidates for a model, the one that was the most likely used to generate the sample solutions is the most restrictive one [14]. Different approaches exist to decide which constraint is the most restrictive. There are mainly statistical approaches [13, 14] and approaches based on a ranking system [6] (that includes many other criteria). Current methods analyze the constraint in isolation. However, adding to a model a constraint that accepts many solutions can reduce more the solution space than adding a constraint that accepts few solutions. It all depends on the interaction between the constraints in the model. We propose the first approach that takes into account this interaction. It uses a model counter to make sure that the constraints suggested to the expert are those that are the most likely to explain the observed sample solutions given the constraints that were already identified to be part of the model.

In this paper, we propose CABSC, an algorithm for Constraint Acquisition Based on Solution Counting. CABSC uses examples of solutions to evaluate which constraints to keep from a chosen set of candidates. The selection process is based on solution counting using model counters, an approach which differs from the current methods detailed in Section 2. The definitions for our approach are given in the Section 3, followed by a practical explanation in Section 4. Experiments are explained in Section 5 and discussed in Section 6.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 15; pp. 15:1–15:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 General Background

Constraint acquisition is an intricate problem that can be solved in a few ways. A first idea called passive learning requires examples of solutions and/or non-solutions. A system chooses which constraint represents best the examples from a preselection of constraints. The preselected pool of constraints from which the model is built is called a bias. Other methods use active learning and generate examples of solution and ask an expert to classify the examples given. From a bias, the system choses the best set of constraints according to the answer provided.

Passive learning systems exploit the idea that the underlying structure of the given examples gives information about the model to learn. Beldiceanu and Simonis [6] created a Model Seeker that learns constraints from a catalog given positive and negative examples. The constraints of the catalog that accepts the positive examples and reject the negative examples are sorted with the more likely constraints having a higher rank. The sorting system is based on a ranking value that is a function of multiple parameters, including the number of solutions satisfying the constraint [5]. A constraint accepting fewer solutions is more likely to be the constraint that generated the examples as there is a lesser chance that the examples are a product of a coincidence. To work, this method needs to make the hypothesis that the constraints learned are independent of each other. That hypothesis is not what transpires in real applications and may result in errors. Two constraints with a small but near identical set of solutions would be picked over two constraints accepting more solutions if picked individually but very few solutions when combined. This is counterproductive as the idea is often to complete an already existing model or to learn multiple constraints at the same time.

Picard-Cantin et al. [13] approached the problem with a statistical approach with the idea that the constraint that best explains the examples is the most improbable one. Equation (1) was therefore used by Picard-Cantin et al. [14] to calculate the probability of the constraints where $G_{\text{C}}(P)$ is the probability that a random assignment satisfies the constraint C with the parameters P. The parameters can be, for instance, the coefficients of a linear equation. $S_{\text{C}}(P)$ is the solution set that satisfies the constraint C with the parameters P. The probability is calculated for a constraint over n variables. $prob(e)$ is the probability to observe an assignment e of n variables and $prob(e_i)$ is the probability to observe an assignment of a single variable.

$$G_{\text{C}}(P) = \sum_{\boldsymbol{e} \in S_{\text{C}}(P)} \text{Prob}(\boldsymbol{e}) = \sum_{\boldsymbol{e} \in S_{\text{C}}(P)} \prod_{i=1}^{n} \text{Prob}(e_i) \tag{1}$$

A hypothesis of independence between the variables of the constraints is applied in the equation. Whenever a variable is in the scopes of multiple constraints, the hypothesis of independence between the variables becomes an approximation. In all cases, the preferred constraints are the ones with a small number of solutions but the independence hypothesis can lead to an erroneous ranking of the constraints. Moreover, this system was not designed for learning multiple constraints and requires solution counting algorithms specialized for each constraint.

Another approach was suggested by Bessiere et al. [8] which consists of creating a model from partial queries, a form of active learning, with an algorithm called QuAcq. The system creates an example and asks an expert whether the presented values are valid. The system adapts the learned constraints depending on the provided answer. Recently, QuAcq was improved with a new version called QuAcq2 [7]. In some cases, QuAcq and QuAcq2 can require a number of queries too high to be efficiently answered by a person. The number of queries can go as high as $n^2 \log(n)$ where n is the number of variables of the problem [7]. Multiple authors tackled this problem such as Daoudi et al. [10], Addi et al. [2], Addi et al. [1], Arcangioli and Lazaar [3], Tsouros et al. [20] and Tsouros et al. [19], but up to thousands of queries can still be needed.

3 The CABSC approach

The CABSC approach (Constraint Acquisition Based on Solutions Counting) we introduce fulfills three goals:

1. To lift the hypothesis of independence between variables;
2. To allow learning multiple constraints;
3. To work with any set of constraints for which filtering algorithms exist, rather than solution counting algorithms.

CABSC models the process of learning constraints as a Meta-CSP. As will be described in Section 3.1, a Meta-CSP is a combinatorial problem whose solution is a CSP. In our case, the solution is the CSP we learn from the examples. When modeling the Meta-CSP, we list the mandatory constraints, i.e. the constraints that we know belong to the model, and also the possible constraints, those that could belong to the model. The variables of the Meta-CSP encode the possible activation of a constraint and also the parameters of the constraints, such as the coefficients of a linear constraint. Solving the Meta-CSP provides the learned model. To do so, we use a branch and bound to decide which constraint to keep and identify the values of the parameters. Our approach uses constraint programming to model a Meta-CSP and to define a family of CSPs from which we can learn. We therefore do not aim to learn any CSP but the optimal CSP among a set programmed through constraint programming. This approach is inspired from regression where one defines a family of functions (e.g. linear functions) and aims at finding the function from this family that best fits the data. Here, we aim at finding the CSP from a family of CSPs defined by the Meta-CSP that best explains the examples.

As there are multiple candidate constraints that could belong to the learned model, we follow Beldiceanu and Simonis [6] and Picard-Cantin et al. [13] by selecting the constraints that minimize the number of solutions. However, instead of analyzing the constraints individually like Beldiceanu and Simonis [6] and Picard-Cantin et al. [13], our system reasons globally on all constraints which allows us to consider multiple different constraints at once.

In order to lift the hypothesis that variables and constraints in a CSP are independent, we directly count the solutions of a model using a model counter. The solution to our Meta-CSP is therefore a CSP whose constraints are satisfied by all observed examples and is as restrictive as possible, i.e. it minimizes the number of solutions.

Our approach has two main differences from existing methods. The first difference is that constraint programming, through the declaration of a Meta-CSP, is used to define a family of CSPs from which we can learn. A second difference from most existing methods is that we use a criterion with a global view on the model to learn by considering the constraints to learn as a whole instead of individually.

3.1 Definition of a Meta-CSP

Following [16], a CSP $\mathcal{P}$ is a triple $\mathcal{P} = \langle X, \text{dom}, C \rangle$ where X is a n-tuple of variables $X = \langle X_1, X_2, \ldots, X_n \rangle$, dom is a function that maps a variable in $X_i \in X$ to a set of values, called domain, that can be assigned to the variable X_i, C is a t-tuple of constraints

$C = \langle C_1, C_2, \ldots, C_t \rangle$. A constraint C_j is a pair $\langle R_j, S_j \rangle$ where $S_j \subseteq X$ is the scope of the constraint and R_j is a relation on the variables in S_j. In other words, R_j is a subset of the Cartesian product of the domains of the variables in S_j. A solution to the CSP $\mathcal{P}$ is an assignment to the variables $X = v_1, \ldots, X_n = v_n$ such that $v_j \in \mathrm{dom}(X_j)$ $\forall 1 \leqslant j \leqslant n$ and each C_j is satisfied in that the tuple $\langle v_1, \ldots, v_n \rangle$ projected onto S_j is a tuple in R_j.

We extend the definition of a CSP to a Meta-CSP. The solution of a Meta-CSP is a CSP. In our case, it is the CSP we want to learn. A Meta-CSP is a tuple $M = \langle X, P, \alpha, \mathrm{dom}, E, C \rangle$ where $X = \langle X_1, \ldots, X_n \rangle$ are the decision variables, $P = \langle P_1, \ldots, P_q \rangle$ are the parameter variables, $\alpha = \langle \alpha_1, \alpha_2, \ldots \rangle$ are the activation variables, dom is a function that maps a variable in $X \cup P \cup \alpha$ to a set of values that can be assigned to the variable, E is the example matrix of dimensions $m \times n$, and $C = \{C_1, \ldots, C_t\}$ is a set of constraints. A row $e_i = \langle e_{i,1}, \ldots, e_{i,n} \rangle$ of matrix E satisfies $e_{i,j} \in \mathrm{dom}(x_j)$ and is a solution to the CSP we want to learn. The examples of the matrix must satisfy the constraints that we want to learn.

A constraint C_j is a quadruple $\langle R_j, S_j, P_j, \alpha_j \rangle$ where $S_j \subseteq X$ is the scope of the constraint, $P_j \subseteq P \cup \alpha$ its parameters set and $\alpha_j \in \alpha$ its activation variable. For instance, for a linear constraint, the parameters P_j are the coefficients that need to be learned. To each constraint C_j is associated the activation variable α_j with domain $\mathrm{dom}(\alpha_j) \subseteq \{\bot, \top\}$. Deciding whether α_j is true ($\top$) is equivalent to deciding whether the constraint appears in the learned model. One can force a constraint to appear in the learned model by setting $\mathrm{dom}(\alpha_j) = \{\top\}$ in the definition of the Meta-CSP. The relation R_j is a set of the assignments accepted by the constraint along with the parameters given to the constraint: $R_j \subseteq \bigtimes_{x \in S_j} x \times \bigtimes_{p \in P_j}$.

A solution to the Meta-CSP is an assignment to the parameter variables $P_1 = p_1, \ldots, P_q = p_q$ and an assignment to the activation variables $\alpha_1 = r_1, \ldots, \alpha_t = r_t$ such that $r_j \in \mathrm{dom}(\alpha_j)$ for all constraints C_j, $p_k \in \mathrm{dom}(P_k)$ for all $1 \leqslant k \leqslant q$. Finally, the examples must satisfy the activated constraint, i.e. $\forall 1 \leqslant j \leqslant t,\ \alpha_j \implies \forall i\ \langle e_{i,1}, \ldots, e_{i,n}, p_1, \ldots, p_q \rangle \in R_j$.

4 Framework

4.1 The Language

We augmented the MiniZinc language [12] to model a Meta-CSP. The declaration of constraints in a Meta-CSP differs from the one in a CSP in two ways. First, the constraints had to be rewritten in MiniZinc to include the Boolean activation variable. This avoids writing explicitly, for each constraint, the underlying constraints needed for such variables. Second, when declaring the scope of a constraint, the indices of the decision variables in X need to be stored in the constraint. Indeed, the constraint's filtering algorithm needs a map of the decision variables in its scope to the columns of the matrix of examples E. Therefore, constraints used for the Meta-CSP have different specifications from what is possible within MiniZinc, which is why the language had to be augmented. The MiniZinc language was also modified to better communicate with the solver we developed, i.e. imports and heuristics were adapted to give a better control. Even though the modifications to MiniZinc do not change its fundamental structure, the way to write a Meta-CSP is made significantly easier.

Listing 1 provides a code snippet written in the augmented MiniZinc language. A set of two-dimensional points are given as solutions of an unknown CSP problem. We know that the x and y coordinates of these points are nonnegative. We do not know whether these points are subject to a linear inequality or an elliptic inequality. This Meta-CSP will tell us.

Listing 1 Code snippet of the augemented MiniZinc.

```
1  set: domain = 1..10;
2  array: x = [1]; %Points are (x,y)
3  array: y = [2];
4  array: x_y = [1..2];
5  var domain: a;
6  var domain: b;
7  var domain: c;
8  var 0..1: activation1;
9  var 0..1: activation2;
10
11 constraint Linear(x, [1], ">=", 0, true); % x >= 0
12 constraint Linear(y, [1], ">=", 0, true); % y >= 0
13 constraint Linear(x_y, [a,b], "<=", c, activation1);  % a*x + b*y <= c
14 constraint Ellipse(x_y, [a,b], "<=", c, activation2); % a*x^2 + b*y^2 <= c
15 constraint Xor(activation1, activation2, true);
```

The decision variables x and y are declared on lines 2 and 3. As their values are known for each example, they are not declared as variables using the keyword *var* but rather as constants corresponding to the column numbers in the example matrix E.

Line 11 declares the first constraint of the problem. It is interpreted as follows: It is a linear constraint whose scope is the decision variable x, whose coefficient vector is [1], whose comparison operator is $\geqslant$, and whose right-hand side is 0. It can be interpreted as $[1]^T x \geqslant 0$. The activation variable is set to *true*, which means that this constraint is known to belong to the CSP. Line 12 imposes $y \geqslant 0$ with a similar constraint. Line 13 encodes the first constraint that we want to learn. It is a linear constraint over the variables x and y whose coefficients and right-hand side are unknown and are represented by the parameter variables a, b, and c. Finally, it is unknown whether this constraint belongs to the CSP. The activation variable `activation1` will be set to 1 if it belongs and 0 otherwise. Line 14 encodes the second constraint that we want to learn. It is an elliptic constraint centered at the origin where parameter variables a, b, and c are reused. The activation variable `activation2` is used for this constraint. Line 15 shows an example of a constraint over two activation variables meaning that exactly one constraint among the linear and the elliptic constraint can be activated. This is an example of how one can define the bias (i.e. the family of CSPs from which the CSP is learned) and exploit the full richness of CP to model the learning process.

A constraint can be satisfied by all examples even if the solver chooses not to learn it by setting its activation variable to $\bot$, unlike a reified constraint which would be set to $\bot$ only if the examples are not satisfied.

Figure 1 is a graphical representation of the problem encoded in Listing 1. The curves represent both candidate constraints: the linear candidate and the elliptic candidate. The dots are the sample solutions that are provided.

We are looking for the CSP that is the most likely the one that generated the points provided in the example matrix E, i.e. the CSP that accepts the fewest solutions among all CSPs that accepts all solutions in E. We see in Figure 1 that the Elliptic constraint accepts 9 solutions while the linear constraint accepts 10 solutions. Therefore, our approach learns that an elliptic inequality fits best the examples with parameters $a = 1, b = 2, c = 10,$ `activation1` $= \bot$ and `activation2` $= \top$, which confirms the visual intuition.

4.2 The Solver

We created a custom solver called CabscSolver that reads the Meta-CSP written in the augmented MiniZinc language and the example matrix E. This solver finds the CSP that accepts the fewest solutions among all CSPs that accept all examples. CabscSolver uses a branch and bound to solve the problem. The branching variables are the activation

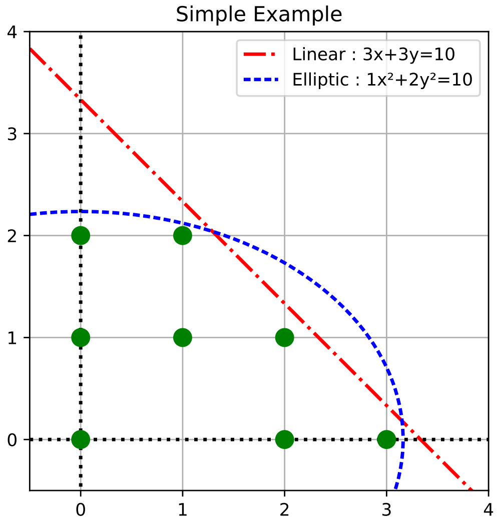


Figure 1 Simple example.

and parameter variables $\alpha \cup P$. After branching, constraint propagation is triggered. Let $C(\vec{x}, \vec{p}, \alpha)$ be a constraint where $\vec{x}$ is the vector of decision variables, $\vec{p}$ is the vector of parameter variables, and α is the activation variable. Only the domains of $\vec{p}$ and α need to be filtered as the values of the decision variables are provided by the examples. To filter the constraint, one needs to filter the expression $\alpha \implies \bigwedge_{i=1}^{m} C(e_i|\vec{x}, \vec{p}, \top)$ where $e_i|\vec{x}$ is the projection of the i^{th} example over the decision variables in the scope of the constraint. The filtering can take place only when the value of the activation variable α is known. Indeed, if α is false ($\perp$), the constraint is satisfied and no filtering is required. If α is true ($\top$), a conjunction of constraints needs to be filtered. Each component of the conjunction can be filtered independently, but a more sophisticated algorithm might process the examples in batch to gain in efficiency. The choice is specific to each constraint. In the example of Listing 1, if variable `activation1` is set to $\top$ during the search process, the linear constraint filters values 1 and 2 from the domain of c as the point $(x, y) = (3, 0)$ prevents the linear constraint to be satisfied when $c \leqslant 2$.

In order to make the branch and bound effective at minimizing the number of solutions accepted by the CSP we want to learn, one needs to compute a lower bound on this number of solutions. This computation is carried in two phases. In the first phase, we detect if a situation occurs where it is possible to deduce which CSP accepts the fewest solutions, regardless whether this CSP accepts the examples or not. If such a CSP can be deduced, the second phase launches a model counter to compute the number of solutions for this CSP.

Some constraints have monotonic parameters with respect to the number of solutions they accept [14]. For instance, consider the linear constraint $c^T x \leqslant b$ where the parameters c and b are a vector of nonnegative coefficients and a nonnegative right-hand side. The vector x contains the decision variables. It is clear that the number of solutions accepted by this constraint decreases as the values in c increases and b decreases. In order to obtain the most restrictive constraint, one needs to fix the parameters c to their greatest values in their domains and b to its smallest value. If all parameter variables with more than one value in their domain are monotonic and all constraints agree to set these variables to the same values (either largest or smallest) in order to minimize the number of solutions, then we can proceed to the second phase and compute a lower bound on the number of solutions. Otherwise, we use the number of examples as the trivial lower bound as this is the minimum number of solutions the CSP can accept. Since parameter variables can be subject to constraints, it is possible that fixing the value of the parameter variables leads to inconsistencies. In such a case, the CSP used to calculate the lower bound has no solution. Even if that CSP has no solution, multiple CSPs can exist further in the search tree. We therefore still use the number of examples as a lower bound on those nodes.

In the second phase, the parameter variables are set to their most restrictive value and activation variables that are not set to *false* are forced to be *true* in order to have the maximum number of activated constraints. This results in a CSP $\mathcal{A}$ for which the number of solutions needs to be determined. There exists a few model counters in the literature such as the exact probabilistic model counter GANAK [17] or the approximate model counter ApproxMC4 [9, 18]. Both of these counters can only approximate the number of solutions of a model written as a CNF file. CabscSolver encodes the constraints of $\mathcal{A}$ into a pseudo-Boolean language that is translated to a CNF using the MiniSat+ module NaPS [11]. This CNF is given to the model counter which calculates the number of solutions of the model. This number is used as a lower bound on the number of solutions of the learned CSP for the current node of the branch and bound.

Executing the model counter is the most time-consuming operation in the whole search process. Since the parameter variables are often fixed to the same values (due to their monotonicity), it is worth implementing a cache system. Therefore, before calling the model counter, the system checks whether the generated model was previously counted, and if so, returns the number of solutions previously found.

The resulting algorithm is summarized in Figure 2. The next branching is defined by the best-first-search heuristics, i.e. the open node with the smallest lower bound is expanded. When the lower bound of a node is greater than the number of solutions of the incumbent CSP, this node is closed.

5 Experiments

5.1 Implementation

We implemented CabscSolver in Python[1]. While this interpreted language leads to a slow execution, in practice, most of the computation time is spent in the model counters. We use GANAK [17] and ApproxMC4 [9, 18] as model counters that are both efficiently implemented in C/C++.

GANAK is a probabilistic exact model counter [17]. Using the parameter δ, GANAK guarantees with a probability of at least $1 - \delta$ that the value provided is an exact count. The approximate model counter ApproxMC4 [9, 18] was also integrated to our solver to count the number of solutions since some calculations are much faster with this counter. Let F be the real number of solutions of a model. ApproxMC4 gives an approximation of F with a configurable confidence. More specifically, it returns a count that is guaranteed to be within $[\frac{F}{(1+\epsilon)}, F \cdot (1 + \epsilon)]$ with a probability of at least $1 - \delta$, where ϵ and δ are the configurable parameters. The chosen values for the parameters ϵ and δ are discussed in Section 5.3.

[1] The code and the benchmarks will be available on the authors' web sites.

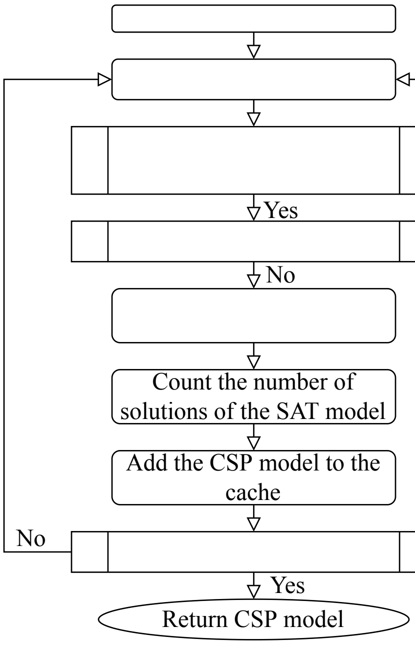

No
Yes
Yes
No
Count the number of solutions of the SAT model
Add the CSP model to the cache
No
Yes
Return CSP model

Figure 2 Flow chart for the CABSC approach.

To read the Meta-CSP models, using the parsing toolkit Lark, we implemented, from scratch, a parser that interprets a subset of the MiniZinc language [12] to which we add the necessary augmentations. MiniZinc was not changed in any way other than the required augmentations. This allows us to efficiently communicate the Meta-CSP models to the solver.

5.2 Instances

We try to learn the constraints inspired from nurse scheduling problems. The problem consists of creating a schedule that respects a set of predetermined rules. In these schedules, the increments used are days, meaning that we are only preoccupied on a daily basis whether the nurses work or not. Let $\eta \in \{2, 3, 4\}$ and $d \in \{7, 14, 21, 28\}$ be the number of nurses and days in a schedule (with $\eta d \leqslant 56$). All instances have a matrix of decision variables $[[X_{(1,1)}, \ldots, X_{(1,d)}], \ldots, [X_{(\eta,1)}, \ldots, X_{(\eta,d)}]]$. Each variable of the matrix represents a day of work for a nurse with its domain being $\{0, 1, 2\}$. $X_{i,j}$ takes the value 0 if the nurse i does not work on day j. If the nurse i does work during day j, $X_{i,j}$ takes the value 1 or 2, depending on whether the nurse works in room 1 or 2.

In the first benchmark, denoted **Sequence**, we want to learn one of these two constraints on the rows of the matrix.

$$\textsc{Sequence}([X_{i,1}, \ldots, X_{i,d}], l, u, k, V) \qquad \forall 1 \leqslant i \leqslant \eta \tag{2}$$

$$\textsc{Among}(t_1, t_2, [X_{i,7w+1}, \ldots, X_{i,7(w+1)}], V) \qquad \forall 1 \leqslant i \leqslant \eta,\ \forall 0 \leqslant w < \frac{d}{7} \tag{3}$$

Constraint (2) is the Sequence constraint [4] that is satisfied when at least l and at most u variables in a window $X_{i,j}, \ldots, X_{i,j+k-1}$ of k consecutive variables are assigned to a value in the set V. This constraint is used to spread out the workload of the nurses over the days without underload nor overload. The parameters l, u, and k are unknown and need to be learned. Their domains are given by $\mathrm{dom}(l) = \mathrm{dom}(u) = \mathrm{dom}(k) = [0, 7]$ and are subject to $l \leqslant u < k$. The set V is known and fixed to $\{1, 2\}$ as these are the values that represent a nurse who is working. Constraint (3) simply constrains the number of work days to be at least t_1 and at most t_2 every week. The parameter variables t_1 and t_2 have for domain $\mathrm{dom}(t_1) = \mathrm{dom}(t_2) = [0, 7]$. One, and only one, constraint among (2) or (3) must be activated. We therefore constrain the activation variables of both constraints with a `Xor`, just like the line 15 of Listing 1. The benchmark **Sequence** is composed of 368 instances generated with distinct constraints, parameters, and examples. These instances satisfy the Sequence constraint and the parameters lie in the intervals $l, u \in [1, 6]$ and $k \in [2, 7]$.

The second benchmark, denoted **Complex**, inherits all the characteristics of the **Sequence** benchmark, including the constraint to learn, to which additional known constraints are added on the decision variables. These constraints have for goal to encode a more realistic situation where constraints that we want to learn are mixed with constraints that are known. For each column $[X_{(1,d)}, \ldots, X_{(\eta,d)}]$ of the matrix that represents the schedule for the day d, we have the constraint $\textsc{Among}(b, 3, [X_{(1,d)}, \ldots, X_{(\eta,d)}], V)$ where $b = 1$ if d is a Monday, Tuesday, Wednesday, or Thursday and $b = 2$ otherwise. This constraint and its parameters are known and added to the Meta-CSP with an activation variable set to $\top$. This constraint does not need to be learned. For instances with 3 or more nurses, we also have another known constraint $X_{(\eta,j)} = 0 \vee X_{(\eta-2,j)} = 0 \quad \forall j \in \{1, \ldots, d\}$ in order to prevent nurse η from working at the same time as $\eta - 2$. When applicable, this constraint is also included in the Meta-CSP as a known constraint. The **Complex** benchmark has 247 instances that satisfy the Sequence constraint with the parameters lying in the intervals $l, u \in [1, 6]$ and $k \in [2, 7]$.

In the third benchmark denoted **Vacation**, the Meta-CSP is identical to the one of **Complex**. However, the examples E that are provided to the solver are particular: nurses can be non-working for 7 consecutive days. This represents a situation where the staff goes on leave during the vacation period. These leaves violate the Sequence constraint and force the solver to activate the Among constraint and learn its parameters t_1 and t_2. The examples were created such that nurse η never takes a vacation but other nurses do. For a problem spanning w weeks, nurses globally take no more than w weeks of vacation. We generated 272 instances for this benchmark such that the instances satisfy the Among constraint. The parameters lie in the intervals $t_1 \in [2, 3]$ and $t_2 \in [3, 7]$.

The last benchmark **Overtime** uses the same Meta-CSP as **Complex** and **Vacation**, but the examples E provided to learn the CSP differ from **Vacation** on one point: rather than leaving for vacations for 7 consecutive days, the nurses in the **Overtime** benchmark work on a stretch of 7 consecutive days. This represents a situation when the hospital is understaffed and nurses need to work overtime. This benchmark has 304 instances such that the instances satisfy the Among constraint and the parameters lie in the intervals $t_1 \in [2, 7]$ and $t_2 \in [4, 7]$ with the restriction $t_1 \leqslant t_2$.

For all benchmarks, the solver aims to learn exactly one constraint among (2) and (3). The selection depends on the known constraints added to the Meta-CSPs and the examples.

5.3 Experimental Setup

For each instance, the CSP we want to learn was written in the MiniZinc language [12] and used to randomly generate up to a thousand solutions. The Meta-CSP model was written in our augmented-MiniZinc language in order to learn which constraint, between the Sequence and the Among constraints, is activated and what are the parameters that were used to generate the examples.

CabscSolver supports two model counters. We first used the solver with the model counter GANAK [17]. By setting the parameter δ to 0.05, we state that the value returned by the model counter is guaranteed to be exact with a probability of at least 0.95. Tighter guarantees can be used, but the time taken to count the number of solutions of the models increases accordingly. Using this model counter and this configuration, we nevertheless assume the given number of solutions to be exact. GANAK was used with a maximum cache size of 2000 Mb. We ran all benchmarks on the solver using this model counter.

As a second series of tests, we used a mix of ApproxMC4 [9, 18] and GANAK. Some CSP models are faster to evaluate with ApproxMC4, so we tried to make CABSC faster using both model counters. Since ApproxMC4 is not an exact model counter, we did not want to run both model counters at the same time and simply use the result returned by the fastest of the two. When using both model counters, GANAK and ApproxMC4 are simultaneously launched. If GANAK finishes first, ApproxMC4 is terminated. If ApproxMC4 finishes first, GANAK is terminated only if the returned result is conclusive. Indeed, ApproxMC4 returns a solution count that is guaranteed to be within an interval with a parametrized confidence. A solution returned by this model counter could be largely underestimated, which could lead to the wrong CSP model being learned. If F is the exact number of solutions of a CSP, the number of solutions returned by ApproxMC4 lies in $[\frac{F}{(1+\epsilon)}, F \cdot (1+\epsilon)]$ with probability $1-\delta$. When ApproxMC4 returns a number of solutions that is $(1+\epsilon)$ times greater than the number of solutions accepted by the incumbent CSP, the computation of GANAK is halted, and the node is closed, i.e. no children of this node will be explored in the search tree. Otherwise, we draw no conclusion and let GANAK terminate its computation. ApproxMC4 is rather used as a means to close nodes faster than substituting GANAK.

The same way we assumed that GANAK would return exact values, we assume that ApproxMC4 does not give a solution count that is lower than the minimum value of the interval. We used $\delta = 0.10$ and $\epsilon = 0.5$ which means that the count calculated is guaranteed to be in the range $[\frac{F}{1.5}, 1.5F]$ with a probability of at least 0.90. A lower probability is accepted from ApproxMC4 than GANAK since the main focus of using ApproxMC4 is to count CSP models faster than GANAK.

We ran the experiments on a computer with the following configuration: CentOS 7.6.1810, 32 GB ram, Processor Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz, 32 Cores. We simultaneously launch 7 instances of the solver.

From each instance, random subsets of 1, 2, 3, 5, 10, 25, and 100 examples were used. Each time, the top 3 solutions are returned by the solver, and we verify that one of these solutions is the one used to generate the examples. For the **Sequence** and **Complex** benchmarks, the expected constraint to be learned is the SEQUENCE constraint with parameters l, u, and k. For the **Vacation** and **Overtime** benchmarks, the examples violate the SEQUENCE constraint, and the AMONG constraint is expected to be learned with parameters t_1 and t_2.

6 Results and Discussion

Figure 3 presents the results obtained when running CabscSolver using only GANAK for the four benchmarks presented at Section 5.2. On the y-axis is the number of examples that are given to the solver. On the x-axis is the proportion of instances for which the solution is the best one returned by the solver, the second best, the third best, or whether the CSP that was used to produce the examples does not appear at all in the top-3 learned models. We recall that the solver returns the CSP that minimizes the number of solutions.

6.1 Accuracy

CABSC performs generally well as seen in Figure 3. Each benchmark presents a distinct behavior regarding the quality of the results. The first observable behavior is that CABSC succeeds in learning the CSP that was used to generate the data as seen with the benchmark

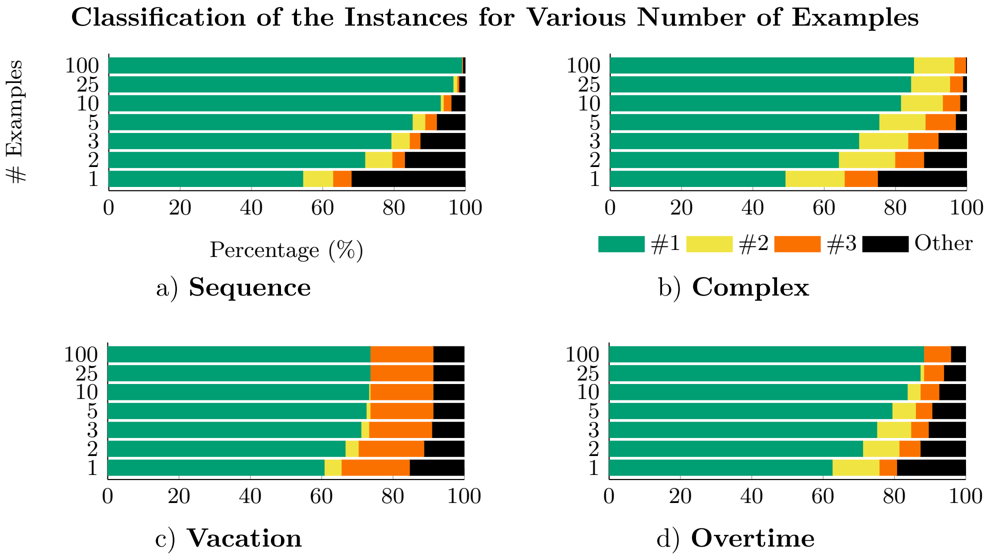


Figure 3 Classification of the instances in percentage for each number of examples. CabscSolver uses GANAK as the only model counter.

Sequence. In this simpler case, the solver has to count the solutions of a conjunction of Sequence constraints, i.e. the constraints to learn. With few examples, our approach stays coherent with the results of Picard-Cantin et al. [13] where they reach above 70% accuracy with a single example of solution, and around 85% accuracy with 5 examples. Extending the number of examples drastically reduces the margin of incorrectly learned instances while the number of examples needed is still relatively low. With only 25 examples, 96.47% of the instances resulted in a correctly learned Sequence constraint at the first try. A few instances could not be resolved even with 100 examples. The unsolved instances occur when the solver finds a more restrictive constraint than the one that was used to generate the examples. This can happen if all the examples given are not enough to filter out parameters that would make the constraints more restrictive. This is why we see that with more examples given, fewer instances remain unsolved. The same phenomenon happens with the **Complex** benchmark where we see an efficient progression as the number of given examples increases.

Finding a more restrictive constraint is not the only way to get an incorrect model. As Figure 3 c) shows, the results for the **Vacation** benchmark converge toward a point where increasing the number of examples does not affect the results while still having a non-negligible proportion (8.82%) of unsolved instances. This is caused by multiple CSPs that are tied. A tie occurs when two distinct CSPs have the same number of solutions. In an instance from **Vacation**, the constraint we want to learn restricts 2 nurses to work a minimum of 3 days and a maximum of 4 days from Monday to Sunday. Since at least one nurse is required to work each day and that a nurse can work a maximum of 4 days within the week, the only way to satisfy the requirements is by having a first nurse working 4 days and the second nurse working 3 or 4 days. It is impossible for one of the nurses to work fewer than 3 days without violating the constraints. The problem comes when setting the value for the minimum number of days a nurse can work during the week. Consider a second selection of parameters where a minimum of 2 working days is required instead of 3. The same solutions are available since this change in parameters does not add solutions. The same goes with a minimum of 1 or 0 working day. This situation leads to four distinct CSPs with the same solution space. Since the objective is to find the CSP accepting the fewest solutions, these four CSPs are equivalent and the solver returns them in an arbitrary order. Most of the unsolved instances in the benchmark **Vacation** have the correct CSP in fourth position, which would have been first if the branching heuristics broke ties differently. We did not observe in our benchmarks situations where the solution spaces differ which let us believe that these models are equivalent. If we pretend for a moment that the **Vacation** benchmark was completed using heuristics that break ties without errors, we obtain the Figure 4.

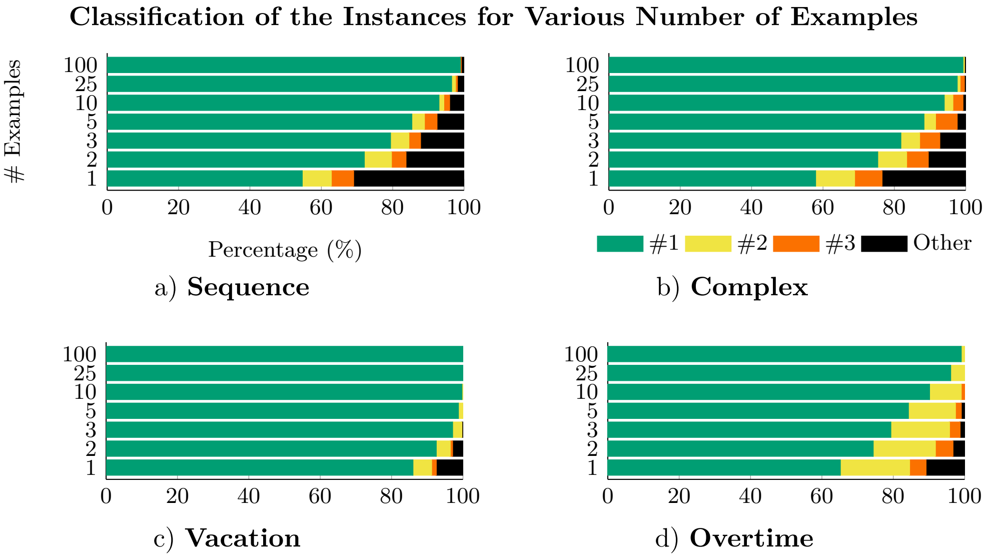


Figure 4 Hypothetical best results for each benchmark.

Figure 4 shows that this hypothetical heuristic allows solving perfectly the **Vacation** benchmark using as few as 10 examples. Improvements are also present with the other benchmarks. This confirms that finding equivalent CSPs is the main reason why the solver does not succeed to correctly learn some CSPs.

The unsolved instances from the **Complex** benchmark are mainly caused by constraints found more restrictive than the correct one while the unsolved instances from the **Vacation** benchmark are mostly caused by equivalent CSPs. The unsolved instances of the **Overtime** benchmark are caused by a mix of these two reasons.

The final results show that our model can accurately learn the constraints even when the schedules contain vacations, overtime, or constraints that interfere with the constraints one wants to learn. Few examples are needed to obtain good results. These figures demonstrate that CABSC can learn constraints with the right parameters in diverse situations.

6.2 Execution Time

6.2.1 Using GANAK alone

For the **Complex** benchmark, model counting represents on average 93.6% of the time spent in the solver. Solution counting is a #P-difficult problem with few effective algorithms. Even with state-of-the-art tools, computing a lower bound on the number of solutions can take several minutes. The bound that took the longest time to compute by GANAK took 648 seconds. Figure 5 represents the time taken to solve all instances, i.e. the $(368 + 247 + 272 + 304) \times 7$ instances that come from the four benchmarks that were solved with 1, 2, 3, 5, 10, 25, and 100 examples using only GANAK as a model counter. Most of the instances are solved within a minute, but the solving time quickly and abruptly rises. This time limitation comes from a few main elements.

First, the size of the Meta-CSP greatly impacts the time needed for CABSC to find a solution. This size is measured in the number of parameter variables and activation variables since their number affects the depth of the search tree, thus the number of nodes explored in the branch and bound. For our instances, a few hundreds nodes could be observed on average resulting in around 30 to 60 unique calls to a model counter.

Second, the examples also impact the total runtime in two ways. With a higher number of examples, the solver is able to filter out more values from the domain of the parameter variables which directly decreases the number of potential calls to a model counter. Using a single example, the instances in the **Complex** benchmark takes on average 385.3 seconds to solve. With a hundred examples, the average time drops to 306.9 seconds, an improvement of 20.35%. The second way the solving time is impacted by the examples is with their length, i.e. the number of decision variables. The more decision variables, the more Boolean variables in the SAT model to count. For this reason, we were not able to learn the constraints of schedules with a horizon of 56 days or more.

Lastly, all bounds do not take the same computation time. Indeed, we obtain SAT instances with various numbers of Boolean variables and clauses. The internal structure of these SAT instances can also vary. The bound that is the slowest to calculate uses a SAT instance with 672 Boolean variables and 1172 clauses and takes 648 seconds to count. The Boolean model with the greatest number of variables has 804 variables and 2052 clauses and is counted in 0.11 seconds. This demonstrates that the counting time does not only depend on the number of decision variables, but also the structure of the problem.

6.2.2 Using both GANAK and ApproxMC4

One method used to improve the time needed to solve a Meta-CSP is by combining a probabilistic exact model counter with an approximate model counter. This allows some CSP models to have their solutions counted quicker. The way ApproxMC4 was added to

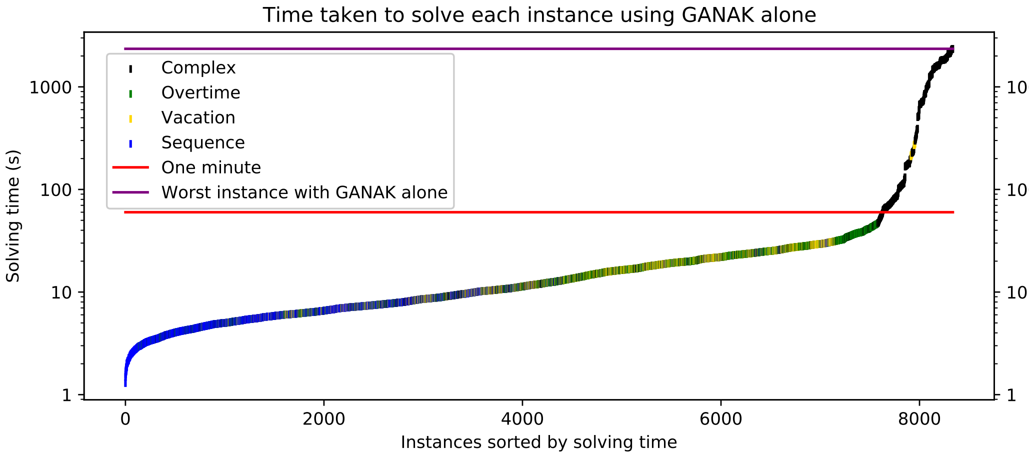


Figure 5 Measures of time for all instances using GANAK alone.

CabscSolver was to use it to prune CSP models from the search tree when the number of solutions was reasonably far from the number of solutions of the best CSP model found so far, as explained in Section 5.3.

This method is a lot faster than using GANAK as the only model counter as demonstrated by the Figure 6. The worst instance with GANAK alone lasted 2350 seconds while the same instance lasted 1099 seconds using ApproxMC4. The arithmetic average solving time of the **Complex** drops from 333.0 seconds to 158.7 seconds. This represents an improvement of 52.3% in average. The geometric average drops from 54.2 seconds to 41.0 seconds, an improvement of 24.4%.

The results obtained using both GANAK and ApproxMC4 have a lower accuracy by a small margin. While the accuracy of the results for the **Sequence**, **Vacation** and **Overtime** benchmarks remain unchanged, **Complex** suffers slight changes when few examples of solutions are given. Since the results have no significant differences to be seen on a graph, the changes are textually reported. With a single example of solution, the percentage of correctly learned CSP models drops from 48.99% to 48.48%. When using two examples of solutions, the percentage of correctly learned CSP models drops from 63.97% to 63.56% and with three examples, it drops from 69.64% to 68.83%. When using five examples of solutions or more, adding ApproxMC4 do not change the results anymore. All the other accuracy results are exactly the same, whether ApproxMC4 was used or not.

The lack of changes in the accuracy of **Sequence**, **Vacation** and **Overtime** benchmarks is mainly caused by the fact that ApproxMC4 returns approximations that are often too close to take into account. The solver then has to ask GANAK to finish calculating the number of solutions of the CSP model regardless of the time needed by ApproxMC4. For the **Complex** benchmark, many CSP models were approximated by ApproxMC4 a lot faster than GANAK could and with values that allow pruning many nodes. ApproxMC4 sometimes overestimates the count of solutions outside the wanted interval of values. Since we used $\delta = 0.10$ for the model counters, ApproxMC4 therefore has a probability of at most 0.10 to return values outside the wanted interval. This can cause many of the evaluations to accidentally prune correct CSP models, which can cause the Meta-CSP not to be properly solved. On the opposite side, it is possible to see improvements in the CSP learned due

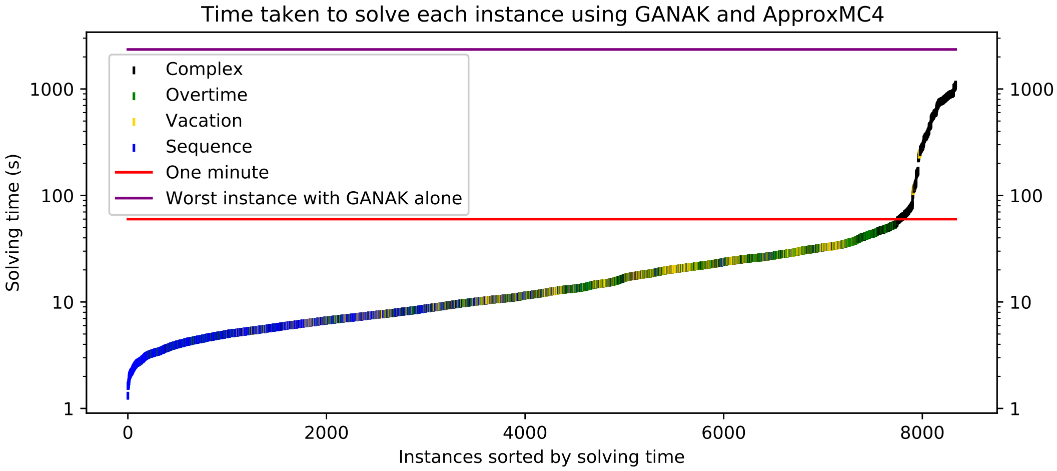


Figure 6 Measures of time for all instances using GANAK with ApproxMC4.

to overestimations that prune CSP models that would be learned if counted exactly. This happened on few instances from the **Complex** benchmark where the correct CSP went from being the third suggestion to the second. Since the correct CSP was not suggested as a first choice, the accuracy of correctly learned CSP models did not improve from these.

6.3 Potential Improvements

There exist several open source model counters that are efficient at counting SAT models, but fewer available programs to count the solutions of a CSP. Translating Sequence constraints into pseudo-Boolean constraints and then to CNF offers no guarantee in the efficiency of the model. Directly counting the solution of a CSP could be faster and would certainly prevent from translating the model.

Parallelization could also speed up the exploration of the search tree. An approach like Embarassingly Parallel Search [15] could be appropriate, but also parallelization within the model counters would be suited as it is offered by ApproxMC3 [9, 18].

7 Conclusion

We introduced CABSC, a technique for Constraint Acquisition Based on Solution Counting. Our approach learns the CSP that accepts all provided examples but that minimizes the size of its solution space. This criterion has proven to return good solutions. The branch and bound uses model counters to compute a bound on the number of solutions for a given CSP. Experimental results show that CABSC successfully learns models and require few examples for our benchmarks.

References

1 Hajar Ait Addi and Redouane Ezzahir. P_a-QUACQ: Algorithm for constraint acquisition system. In *Smart Data and Computational Intelligence*, pages 249–256, 2019.

2 Hajar Ait Addi, Christian Bessiere, Redouane Ezzahir, and Nadjib Lazaar. Time-bounded query generator for constraint acquisition. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR 2018)*, pages 1–17, 2018.

3 Robin Arcangioli and Nadjib Lazaar. Multiple constraint acquisition. In *Proceedings of the 2015 International Conference on Constraints and Preferences for Configuration and Recommendation and Intelligent Techniques for Web Personalization*, pages 16–20, 2015.

4 Nicolas Beldiceanu and Évelyne Contejean. Introducing global constraints in chip. *Mathematical and Computer Modelling*, 20(12):97–123, 1994.

5 Nicolas Beldiceanu and Helmut Simonis. A constraint seeker: Finding and ranking global constraints from examples. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP 2011)*, pages 12–26, 2011.

6 Nicolas Beldiceanu and Helmut Simonis. A model seeker: Extracting global constraint models from positive examples. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP 2012)*, pages 141–157, 2012.

7 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. Partial queries for constraint acquisition. Technical Report abs/2003.06649, CoRR, 2020. `arXiv:2003.06649`.

8 Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Constraint acquisition via partial queries. In Francesca Rossi, editor, *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI-13)*, pages 475–481, 2013.

9 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI-16)*, pages 3569–3576, 2016.

10 Abderrazak Daoudi, Younes Mechqrane, Christian Bessiere, Nadjib Lazaar, and El-Houssine Bouyakhf. Constraint acquisition with recommendation queries. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI-16)*, pages 720–726, 2016.

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

12 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, 2007.

13 Émilie Picard-Cantin, Mathieu Bouchard, Claude-Guy Quimper, and Jason Sweeney. Learning parameters for the sequence constraint from solutions. In *Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming (CP 2016)*, pages 405–420, 2016.

14 Émilie Picard-Cantin, Mathieu Bouchard, Claude-Guy Quimper, and Jason Sweeney. Learning the parameters of global constraints using branch-and-bound. In *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming (CP 2017)*, pages 512–528, 2017.

15 Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP 2013)*, pages 596–610, 2013.

16 Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

17 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. Ganak: A scalable probabilistic exact model counter. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI-19)*, pages 1169–1176, 2019.

18 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 33rd AAAI Conference on Artificial Intelligence (AAAI-19)*, pages 1592–1599, 2019.

19 Dimosthenis C. Tsouros, Kostas Stergiou, and Christian Bessiere. Structure-driven multiple constraint acquisition. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP 2019)*, pages 709–725, 2019.

20 Dimosthenis C. Tsouros, Kostas Stergiou, and Panagiotis G. Sarigiannidis. Efficient methods for constraint acquisition. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP 2018)*, pages 373–388, 2018.

Computing Relaxations for the Three-Dimensional Stable Matching Problem with Cyclic Preferences

Ágnes Cseh
Institute of Economics, Centre for Economic and Regional Studies, Budapest, Hungary

Guillaume Escamocher
Insight Centre for Data Analytics, School of Computer Science and Information Technology, University College Cork, Ireland

Luis Quesada
Insight Centre for Data Analytics, School of Computer Science and Information Technology, University College Cork, Ireland

Abstract

Constraint programming has proven to be a successful framework for determining whether a given instance of the three-dimensional stable matching problem with cyclic preferences (3DSM-CYC) admits a solution. If such an instance is satisfiable, constraint models can even compute its optimal solution for several different objective functions. On the other hand, the only existing output for unsatisfiable 3DSM-CYC instances is a simple declaration of impossibility.

In this paper, we explore four ways to adapt constraint models designed for 3DSM-CYC to the maximum relaxation version of the problem, that is, the computation of the smallest part of an instance whose modification leads to satisfiability. We also extend our models to support the presence of costs on elements in the instance, and to return the relaxation with lowest total cost for each of the four types of relaxation. Empirical results reveal that our relaxation models are efficient, as in most cases, they show little overhead compared to the satisfaction version.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Design and analysis of algorithms

Keywords and phrases Three-dimensional stable matching with cyclic preferences, 3DSM-CYC, Constraint Programming, relaxation, almost stable matching

Digital Object Identifier 10.4230/LIPIcs.CP.2022.16

Supplementary Material *Software (Source Code)*: `https://doi.org/10.5281/zenodo.6798122`

Funding This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant numbers 12/RC/2289-P2 and 16/SP/3804, which are co-funded under the European Regional Development Fund. Cseh was supported by OTKA grant K128611 and the János Bolyai Research Fellowship.

Acknowledgements COST Action CA16228 European Network for Game Theory.

1 Introduction

Defined on three instead of two agent sets, the 3-dimensional stable matching problem [34] is a natural generalisation of the well-known stable marriage problem [24]. Its most studied variant is the *3-dimensional stable matching problem with cyclic preferences* (3DSM-CYC) [42], in which agents from the first set only have preferences over agents from the second set, agents from the second set only have preferences over agents from the third set, and finally, agents from the third set only have preferences over agents from the first set.

A matching is a set of triples such that each triple contains one agent from each agent set and each agent appears in at most one triple. A *weakly stable matching* does not admit a blocking triple such that *all three* agents would improve, while according to *strong stability*, a triple already blocks if *at least one* of its agents improves, and the others in the triple remain equally satisfied.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 16; pp. 16:1–16:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Constraint programming approaches allow one to identify instances that do *not* admit a weakly or strongly stable matching – these will be in the focus of our investigation. For such an instance, how to construct a matching that is blocked by only a few triples? Alternatively, which matching minimises the number of justifiably disappointed agents who appear in a blocking triple? A somewhat more sophisticated approach is to assume that a central authority is able to compensate blocking triples or even single agents appearing in blocking triples. If such a compensation has been allocated, then the triple or agent withdraws their claim to form a more advantageous coalition. How to find a matching with the lowest compensation needed to eliminate all blocking triples?

In order to facilitate a general framework, we associate a cost with each agent. The goal is then to minimise the total cost of triples or agents who block the matching, or have to be compensated in order to withdraw from blocking.

1.1 Literature review

We first restrict our attention to related work in the 2-dimensional and non-bipartite stable matching settings. We mention two already established relaxations of stability and also elaborate on problem variants with costs. Then we turn to the 3-dimensional setting, review related work on 3DSM-CYC, and finally discuss constraint models.

1.1.1 Relaxing stability

Various stable matching problems need not admit a stable solution. The relaxation of stability by definition necessarily involves the occurrence of blocking pairs. In the literature, two main relaxations have been defined.

The number of blocking pairs is a characteristic property of every matching. A natural goal is to find a matching with the lowest number of blocking pairs; such a matching is called *almost stable*. This approach has a broad literature: almost stable matchings have been investigated in bipartite [33, 29, 6, 27] and non-bipartite stable matching instances [1, 5, 11, 14], but not in the 3-dimensional setting yet.

Agents who appear in blocking pairs in a solution are called blocking agents. Besides minimizing the number of blocking pairs, another intuitive objective is to minimise the number of blocking agents [49]. The complexity of minimizing the number of blocking agents in a non-bipartite stable matching instance is an open problem that was posed in the seminal book of Manlove [38]. Similar, but slightly more complicated instability measures can be found in the paper of Eriksson and Häggström [19].

1.1.2 Costs and preference negotiation in stable matching problems

Arguably the most natural extension of various matching problems is to consider graphs with edge or vertex costs. For bipartite instances with edge costs, finding a minimum-cost stable matching can be done in polynomial time [31, 28, 21, 22]. The same problem for non-bipartite graphs is NP-hard, but 2-approximable under certain monotonicity constraints using LP methods [53, 54].

Vertex costs play a role in stable matching problems if the agents are part of some type of instance manipulation. In their theoretical study, Boehmer et al. [9] allow agents to reshuffle their preference list. College admission is possibly the most widespread application of stability. Surveys report that bribes have been performed in college admission systems in China, Bulgaria, Moldova, and Serbia [30, 37]. However, preference list manipulation, potentially done by assigning money to the affected agents, does not imply an illegal action. The internal assignment process of humanitarian organisations [52, 3, 48] aims at stability in the first place, but it also routinely features salary premium negotiations for staff members sent to a less desirable location.

1.1.3 3DSM-CYC

Several applications areas have been modeled by extended 3DSM-CYC instances. Cui and Jia [16] modeled three-sided networking services, such as frameworks connecting users, data sources, and servers. In their setting, users have identical preferences over data sources, data sources have preferences over servers based on the transferred data, and servers have preferences over users. Building upon this work, Panchal and Sharma [44] provided a distributed algorithm that finds a stable solution. Raveendran et al. [47] tested resource allocation in Network Function Virtualisation. They demonstrated the superior performance of the proposed cyclic stable matching framework in terms of data rates and user satisfaction, compared to a centralised random allocation approach.

A recent real application was described by Bloch et al. [8] who analysed the Paris public housing market. In their work, the first agent set consists of various housing institutions such as the Ministry of Housing, the second agent set is the set of households looking for an apartment, and finally, the third agent set contains the social housing apartments that are to be assigned to these households. Institutions have preferences over household-apartment pairs, and households rank apartments in their order of preference. Cseh and Peters [15] studied a restricted variant where the institutions have preferences directly over the households, no matter which apartment they are matched to.

Maximum relaxations in these applications correspond to the smallest number or cost of users, data sources, servers, households, or housing agencies, who need to be compensated for being part of a blocking triple.

As for the complecity of 3DSM-CYC, Biró and McDermid [7] showed that deciding whether a weakly stable matching exists is NP-complete if preference lists are allowed to be incomplete, and that the same complexity result holds for strong stability even with complete lists. However, the combination of complete lists and weak stability proved to be extremely challenging to solve. After a series of papers [10, 20, 45] proving that small 3DSM-CYC instances always admit a weakly stable matching, Lam and Plaxton [36] recently showed NP-hardness for instances with at least 90 agents per agent set – this is also the size of the smallest known no-instance.

1.1.4 CP models for 3DSM-CYC

Several constraint models have been developed for the bipartite stable matching problem and its many-to-one variant [26, 56, 55, 39, 43, 50]. We build upon the recent work of Cseh et al. [13], who introduced five constraint models for 3DSM-CYC. Besides capturing both weak and strong stability, they translated three fairness notions into 3-dimensional matchings.

1.2 Our contribution

In this paper we study four types of relaxation to 3DSM-CYC, based on two established and two new relaxation principles. For each of these types we propose CP approaches that are built on top of the best two approaches from Cseh et al. [13]. We carry out a comprehensive empirical evaluation on a generated data set that includes both satisfiable and unsatisfiable instances. We analyse the behaviour of our constraint models based on different preference structures, cost functions, and their scalability. The results of the evaluation give insight into the convenience of the introduced types of relaxation, in particular in those cases where the four methods agree on the optimal relaxation.

2 Notation and problem definitions

In Section 2.1 we formally define input and output formats for 3DSM-CYC, using previous notations [13]. The four ways of relaxing stability are then discussed in Section 2.2. Finally, matching costs are introduced in Section 2.3.

2.1 Problem definition

Input and output. Formally, a 3DSM-CYC instance is defined over three disjoint sets of agents of size n, denoted by $A = \{a_1, \ldots, a_n\}$, $B = \{b_1, \ldots, b_n\}$, and $C = \{c_1, \ldots, c_n\}$. A *matching* M corresponds to a disjoint set of triples, where each triple, denoted by (a_i, b_j, c_k), contains exactly one agent from each agent set. Each agent is equipped with her own preferences in the input. The cyclic property of the preferences means the following: each agent in A has a strict and complete preference list over the agents in B, each agent in B has a strict and complete preference list over the agents in C, and finally, each agent in C has a strict and complete preference list over the agents in A. These preferences are captured by the *rank function*, where $\text{rank}_{a_i}(b_j)$ is the position of agent b_j in the preference list of a_i, from 1 if b_j is a_i's most preferred agent to n if b_j is a_i's least preferred agent.

Preferences over triples. The preference relation of an agent on possible triples is derived naturally from the preference list of this agent. Agent a_i is indifferent between triples (a_i, b_j, c_{k_1}) and (a_i, b_j, c_{k_2}), since she only has preferences over the agents in B and the same agent b_j appears in both triples. However, when comparing triples (a_i, b_{j_1}, c_{k_1}) and (a_i, b_{j_2}, c_{k_2}), where $b_{j_1} \neq b_{j_2}$, a_i prefers the first triple if $\text{rank}_{a_i}(b_{j_1}) < \text{rank}_{a_i}(b_{j_2})$, and she prefers the second triple otherwise. The preference relation is defined analogously for agents in B and C as well.

Weak and strong stability. A triple $t = (a_i, b_j, c_k)$ is said to be a *strongly blocking triple* to matching M if each of a_i, b_j, and c_k prefer t to their respective triples in M. Practically, this means that a_i, b_j, and c_k could abandon their triples to form triple t on their own, and each of them would be strictly better off in t than in M. If a matching M does not admit any strongly blocking triple, then M is called a *weakly stable* matching. Similarly, a triple $t = (a_i, b_j, c_k)$ is called a *weakly blocking triple* if at least two agents in the triple prefer t to their triple in M, while the third agent does not prefer her triple in M to t. This means that at least two agents in the triple can improve their situation by switching to t, while the third agent does not mind the change. A matching that does not admit any weakly blocking triple is referred to as *strongly stable*. By definition, strongly stable matchings are also weakly stable, but not the other way round. Observe that it is impossible to construct a triple t that keeps exactly two agents of a triple equally satisfied, while making the third agent happier, since the earlier two agents need to keep their partners to reach this, which then defines the triple as one already in M.

2.2 Relaxing stability

We examine four different ways to relax stability in 3DSM-CYC. Two of them are standard in the stable matching literature and are based on minimising the number of blocking elements, see Section 2.2.1. The other two relaxation notions are introduced in Section 2.2.2, and they build upon elements that are prohibited to be part of a blocking triple. We remark that all four relaxations can be translated to other stable matching problems as well.

2.2.1 Almost stable matchings

Let $\text{sbt}(M)$ denote the set of strongly blocking triples, and $\text{wbt}(M)$ denote the set of weakly blocking triples to a matching M. Since strongly blocking triples are also weakly blocking, $\text{sbt}(M) \subseteq \text{wbt}(M)$.

▶ **Definition 1.** *A* strong triple-almost stable *(TAS) matching is a matching that minimises the function* $|\text{wbt}(M)|$ *over all matchings* M*. Analogously, a* weak TAS *matching is a matching that minimises the function* $|\text{sbt}(M)|$ *over all matchings* M*.*

If the instance admits a strongly stable matching, then it minimises both functions, but otherwise, there is no connection between the sets of weak TAS and strong TAS matchings.

The agents involved in a strongly blocking triple are called strongly blocking agents, and form the set $\text{sba}(M)$. Analogously, agents involved in any weakly blocking triple are called weakly blocking agents, and form the set $\text{wba}(M)$. Notice that $\text{sba}(M) \subseteq \text{wba}(M)$. A natural objective is to find a matching that minimises the functions $\text{sba}(M)$ or $\text{wba}(M)$.

▶ **Definition 2.** *A matching that minimises* $\text{sba}(M)$ *is called* weak agent-almost stable (AAS)*, while a matching that minimises* $\text{wba}(M)$ *is called* strong AAS*.*

Notice that weak AAS and strong AAS matchings are not identical to weak TAS and strong TAS matchings. As an example, consider two matchings M_1 and M_2 such that $\text{wbt}(M_1) = \{(a_1, b_1, c_1), (a_1, b_1, c_2), (a_1, b_1, c_3)\}$ and $\text{wbt}(M_2) = \{(a_1, b_1, c_1), (a_2, b_2, c_2)\}$. We have $|\text{wbt}(M_1)| = 3$ and $|\text{wbt}(M_2)| = 2$, so M_2 is a better strong TAS candidate than M_1. However $|\text{wba}(M_1)| = |\{a_1, b_1, c_1, c_2, c_3\}| = 5$ and $|\text{wba}(M_2)| = |\{a_1, a_2, b_1, b_2, c_1, c_2\}| = 6$, so M_1 is a better strong AAS candidate than M_2.

2.2.2 Accommodating elements

Instead of minimising the number of blocking elements, we can eliminate them altogether by setting some agents to be *accommodating*. Accommodating agents never report that they are part of a blocking triple, which eliminates all blocking triples containing at least one of those agents. In a realistic scenario, accommodating agents are allocated compensation for their poor match.

▶ **Definition 3.** *A* weak minimally-accommodating stable (MAS) *matching is a matching that minimises the number of accommodating agents needed to eliminate all of its strongly blocking triples. Analogously, a* strong MAS *matching is a matching that minimises the number of accommodating agents needed to eliminate all of its weakly blocking triples.*

Notice that MAS matchings are distinct from AAS matchings. As an example, consider the matchings M_2 from before, where $\text{wbt}(M_2) = \{(a_1, b_1, c_1), (a_2, b_2, c_2)\}$, and the matching M_3 such that $\text{wbt}(M_3) = \{(a_1, b_1, c_1), (a_1, b_2, c_2), (a_1, b_3, c_3)\}$. We have $|\text{wba}(M_2)| = 6$ and $|\text{wba}(M_3)| = 7$, so M_2 is a better strong AAS candidate than M_3. However, we need both an

Table 1 Different ways of interpreting relaxation.

	single agent	more than one agent
minimise the number of blocking elements	agent-almost stable (AAS)	triple-almost stable (TAS)
minimise the number of accommodating elements	minimally-accommodating stable (MAS)	minimally-pair-accommodating stable (MPAS)

agent from $\{a_1, b_1, c_1\}$ and an agent from $\{a_2, b_2, c_2\}$ to be accommodating to eliminate the blocking triples in $\text{wbt}(M_2)$, while setting a single agent, a_1, to be accommodating eliminates all blocking triples in $\text{wbt}(M_3)$. Therefore M_3 is a better strong MAS candidate than M_2.

We can extend the definition of accommodating to groups of agents. Agents x and y from different agent sets form an *accommodating pair* if they are prevented from appearing *together* in a blocking triple. In 3DSM-CYC, exactly one of the two agents has preferences over the other agent, without loss of generality let us assume that it is x. Setting x and y to be an accommodating pair expresses that x receives compensation for not being matched to y specifically. However, x can appear in a blocking triple with another agent from the set of y, and y also can block with any other agent than x. This compensation is thus less powerful than the previous one.

▶ **Definition 4.** *A* weak minimally-pair-accommodating stable (MPAS) *matching is a matching that minimises the number of accommodating pairs needed to eliminate all of its strongly blocking triples. Analogously, a* strong MPAS *matching is a matching that minimises the number of accommodating pairs needed to eliminate all of its weakly blocking triples.*

The sets of MPAS and MAS matchings are incomparable. As an example, consider the matching M_3 from before, where $\text{wbt}(M_3) = \{(a_1, b_1, c_1), (a_1, b_2, c_2), (a_1, b_3, c_3)\}$, and the matching M_4 such that $\text{wbt}(M_4) = (a_1, b_2, c_3), (a_1, b_2, c_2), (a_2, b_3, c_1)$. Only the agent a_1 needs to be accommodating to eliminate all blocking triples in $\text{wbt}(M_3)$, but no single agent appears in all blocking triples of $\text{wbt}(M_4)$, so M_3 is a better strong MAS candidate than M_4. On the other hand, we can eliminate all blocking triples in $\text{wbt}(M_4)$ by setting only two pairs to be accommodating, while we need three to do the same for $\text{wbt}(M_3)$. Therefore M_4 is a better strong MPAS candidate than M_3.

Further extending MPAS to groups of three agents would mean minimising the number of accommodating triples, which is equivalent to TAS.

Table 1 summarises the four different notions of relaxation that we have explored. We remark that while AAS and TAS require that the relaxation set covers every blocking element, for MAS and MPAS, the relaxation set must hit every blocking element.

2.3 Matching costs

When computing a minimal set of elements for relaxation, not all agents might be given equal importance. The central authority might allocate a higher compensation to prioritised blocking pairs or to popular agents. For a given relaxation version, the cost of a matching is the sum of the costs of the elements in the minimal set of this particular relaxation. For a given matching M and arbitrary costs on agents and triples, we thus have for strong stability:

$$\text{Cost}_{\text{AAS}}(M) = \sum_{a \in \text{wba}(M)} \text{Cost}(a)$$

$$\text{Cost}_{\text{TAS}}(M) = \sum_{t \in \text{wbt}(M)} \text{Cost}(t)$$

The definitions for weak stability can be obtained by replacing wbt by sbt and wba by sba. For $\mathrm{Cost}_{\mathrm{MAS}}$ and $\mathrm{Cost}_{\mathrm{MPAS}}$, we need a further definition.

▶ **Definition 5.** *For a matching M, set S of agents is* agent-convenient *if setting all agents in S to accommodating implies that M is stable. Analogously, set S of pairs of agents is* pair-convenient *for M if setting all pairs in S to accommodating implies the stability of M.*

This definition is the same for both types of stability. We can now write the remaining matching cost definitions for arbitrary agent and pair costs as follows.

$$\mathrm{Cost}_{\mathrm{MAS}}(M) = \min_{S \text{ is agent-convenient for } M} \sum_{a \in S} \mathrm{Cost}(a)$$

$$\mathrm{Cost}_{\mathrm{MPAS}}(M) = \min_{S \text{ is pair-convenient for } M} \sum_{p \in S} \mathrm{Cost}(p)$$

Notice that in all four types of relaxation, not specifying element costs is equivalent to having them all set to 1. We will therefore refer to a relaxation as an *arbitrary-cost relaxation* when elements have an explicit cost, and as a *unit-cost relaxation* when they do not.

3 Methodology

In this section, we explain how we modified the two best performing models for 3DSM-CYC, called DIV-ranks and HS [13], to enable them to deal with soft constraints.

3.1 Soft DIV-ranks model

The DIV-ranks model for 3DSM-CYC with only hard constraints consists of $3n$ variables $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_n\}$, and $Z = \{z_1, \ldots, z_n\}$, where the domain of each variable v is set as $D(v) = \{1, \ldots, n\}$. Assigning $x_i = j$ (respectively $y_i = j$, or $z_i = j$) corresponds to matching a_i (respectively b_i, or c_i) to her j^{th} preferred agent. The constraints used to find a stable matching M, if any exists, are defined in [13] in the following manner.

- (matching) For all $1 \leq i, j, k \leq n$, the constraint $x_i = \mathrm{rank}_{a_i}(b_j) \wedge y_j = \mathrm{rank}_{b_j}(c_k) \Rightarrow z_k = \mathrm{rank}_{c_k}(a_i)$ is added. This is to ensure that each solution corresponds to a feasible, if not stable, matching. Since domain values correspond to positions in preference lists and not to agents, it is possible for two variables from the same agent set to be assigned the same value. This is why all-different constraints are not used for this model.
- (stability) Under *weak* stability, for all $1 \leq i, j, k \leq n$, the constraint $x_i \leq \mathrm{rank}_{a_i}(b_j) \vee y_j \leq \mathrm{rank}_{b_j}(c_k) \vee z_k \leq \mathrm{rank}_{c_k}(a_i)$ is added. This is to ensure that the triple (a_i, b_j, c_k) is not strongly blocking. When solving the problem under *strong* stability, the inequalities are strict but the following part is added to each disjunction: $\vee(x_i = \mathrm{rank}_{a_i}(b_j) \wedge y_j = \mathrm{rank}_{b_j}(c_k) \wedge z_k = \mathrm{rank}_{c_k}(a_i))$.
- (redundancy) For all $1 \leq i, j, k \leq n$, the constraint $y_j = \mathrm{rank}_{b_j}(c_k) \wedge z_k = \mathrm{rank}_{c_k}(a_i) \Rightarrow x_k = \mathrm{rank}_{a_i}(b_j)$ is added.
- (redundancy) For all $1 \leq i, j, k \leq n$, the constraint $z_k = \mathrm{rank}_{c_k}(a_i) \wedge x_i = \mathrm{rank}_{a_i}(b_j) \Rightarrow y_j = \mathrm{rank}_{b_j}(c_k)$ is added.

For the relaxation version of 3DSM-CYC, we add to the DIV-ranks model an integer variable c_{rel} corresponding to the cost of the relaxation, as well as additional Boolean variables whose exact number depends on the type of relaxation.

- AAS and MAS: a Boolean variable $relA_i$ for each of the n agents a_i in A, a Boolean variable $relB_j$ for each of the n agents b_j in B, and a Boolean variable $relC_k$ for each of the n agents c_k in C, which amounts to $3n$ additional variables.

- TAS: a Boolean variable $rel_{i,j,k}$ for each of the n^3 potential blocking triples (a_i, b_j, c_k).
- MPAS: a Boolean variable $relAB_{i,j}$ for each of the n^2 agent pairs a_i, b_j from $A \times B$, a Boolean variable $relBC_{j,k}$ for each of the n^2 agent pairs b_j, c_k from $B \times C$, and a Boolean variable $relCA_{k,i}$ for each of the n^2 agent pairs c_k, a_i from $C \times A$, which amounts to $3n^2$ additional variables.

For all four types, a variable set to 1 means that its corresponding element is part of the correction set. Determining from the composition of the correction set whether a given triple is allowed to be blocking is expressed in the model by extending the disjunction of the stability constraint corresponding to this triple. The part added depends on the type of the relaxation but not on the kind of stability, so for a given type of relaxation the same part will be added to both weak and strong stability constraints.

- For AAS, we add $\vee(relA_i \wedge relB_j \wedge relC_k)$ to the constraint that checks whether the triple (a_i, b_j, c_k) is blocking. If all three agents are in the correction set, then the constraint is satisfied, and whether this triple is blocking has no effect on the stability of the instance.
- For TAS, we add $\vee rel_{i,j,k}$ to the stability constraint. This immediately satisfies the constraint when the triple is in the correction set.
- For MAS, we add $\vee(relA_i \vee relB_j \vee relC_k)$. Because of the distinction between blocking and accommodating agents, for MAS we only need one agent to be in the correction set for the triple to be disregarded, while for AAS we needed all three agents.
- For MPAS, we add $\vee(relAB_{i,j} \vee relBC_{j,k} \vee relCA_{k,i})$. The constraint is satisfied when any two agents in the triple are present as an accommodating pair in the correction set.

Because relaxation has been added to the stability constraints in a disjunctive way, a trivial solution for the instance can be obtained by assigning 1 to all Boolean variables. Therefore we add a final constraint for the objective function which sums the costs of the elements in the correction set. Minimising this value results in a correction set of minimum cardinality (for unit-cost relaxation), or in a solution of mininum cost (for arbitrary-cost relaxation). Both cases represent a maximum relaxation for the instance. For the unit-cost relaxation, all cost factors in the objective function are replaced by 1.

- For AAS and MAS:
  $c_{rel} = \sum_{i=1}^{n}(relA_i \times \mathrm{Cost}(a_i)) + \sum_{j=1}^{n}(relB_j \times \mathrm{Cost}(b_j)) + \sum_{k=1}^{n}(relC_k \times \mathrm{Cost}(c_k))$.
- For TAS: $c_{rel} = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n}(rel_{i,j,k} \times (\mathrm{Cost}(a_i, b_j, c_k)))$.
- For MPAS:
  $c_{rel} = \sum_{i=1}^{n} \sum_{j=1}^{n}(relAB_{i,j} \times (\mathrm{Cost}(a_i, b_j))) + \sum_{j=1}^{n} \sum_{k=1}^{n}(relBC_{j,k} \times (\mathrm{Cost}(b_j, c_k)))$
  $+ \sum_{k=1}^{n} \sum_{i=1}^{n}(relCA_{k,i} \times (\mathrm{Cost}(c_k, a_i)))$.

3.2 Soft HS model

We extend the HS model from Cseh et al. [13] by relaxing the constraints that enforce the stability of the matching. Following Cseh et al. [13], in the soft HS model, we assume that T is the set of all possible triples $\{(a_1, b_1, c_1), (a_1, b_1, c_2), \ldots, (a_n, b_n, c_n)\}$, where without loss of generality, the triples in T are ordered, that is, $t_i \in T$ refers to the i^{th} triple of T. We also borrow their definition of non-blocking triples, that is, given a triple $t \in T$, we denote by $BT(t)$ all the triples in T that prevent t from becoming a blocking triple given the preferences. The variables and constraints of the model are as follows:

- Let M be a set variable whose upper bound is T.
- Let S be a set variable whose upper bound is as follows.
  - For AAS/MAS: $A \cup B \cup C$
  - For TAS: T
  - For MPAS: $A \times B \cup B \times C \cup C \times A$
- Let c be an integer variable corresponding to the cost of the relaxation.
- (matching) Ensure that each agent from each set is matched by having:
  - $\forall a \in A : \sum_{t_i \in T : a \in t_i} (t_i \in M) = 1;$
  - $\forall b \in B : \sum_{t_i \in T : b \in t_i} (t_i \in M) = 1;$
  - $\forall c \in C : \sum_{t_i \in T : c \in t_i} (t_i \in M) = 1.$
- (stability) In the original version, each stable matching is a hitting set of the non-blocking triples (i.e., $\forall t_j \in T : M \cap \{i : t_i \in BT(t_j)\} \neq \emptyset$). We relax this definition as follows.
  - For AAS: $\forall t_j \in T : \exists \langle a, b, c\rangle \in BT(t_j) : \langle a, b, c\rangle \in M \vee \{a, b, c\} \subseteq S$
  - For TAS: $\forall t_j \in T : \exists t_i \in BT(t_j) : t_i \in M \vee t_i \in S$
  - For MAS: $\forall t_j \in T : \exists \langle a, b, c\rangle \in BT(t_j) : \langle a, b, c\rangle \in M \vee \{a, b, c\} \cap S \neq \emptyset$
  - For MPAS: $\forall t_j \in T : \exists \langle a, b, c\rangle \in BT(t_j) : \langle a, b, c\rangle \in M \vee \{\langle a, b\rangle, \langle b, c\rangle, \langle c, a\rangle\} \cap S \neq \emptyset$
- (cost of relaxation) The cost variable is constrained as follows:
  - For AAS/MAS: $c = \sum_{x \in S} Cost(x)$
  - For TAS: $c = \sum_{\langle a,b,c\rangle \in S} Cost(a, b, c)$
  - For MPAS: $c = \sum_{\langle x,y\rangle \in S} Cost(x, y)$

The type of stability is addressed in the computation of the BT sets – the model as such is not concerned with this aspect. In HS, matching M is constrained to be a set of triples representing M as defined in Section 2.1, so the cost of the relaxation follows the definitions in Section 2.3. In the actual implementation, M is represented in terms of an array of n^3 Boolean variables, where each variable refers to the inclusion/exclusion of the corresponding triple in the mapping. Similarly, S is also represented as an array of Boolean variables. The size of this array is either $3n$, $3n^2$ or n^3, depending on the type of relaxation.

4 Experimental results

All experiments were performed on machines with Intel(R) Xeon(R) CPU with 2.40GHz running on Ubuntu 18.04. Tests for the DIV-ranks model were processed by MiniZinc 2.5.5 [41] before being given to the two constraint solvers Chuffed 0.10.4. [12], which is based on lazy-clause generation, and Gecode 6.3.0 [25]. The HS model on the other hand has been directly encoded using Gecode 6.2.0.

4.1 Dataset

4.1.1 Preference lists

The instances used in our experiments belong to three different classes: Random, ML1swap, and ML2swaps. In the latter two, the preferences are based on master lists. Master list instances are instances where the preference lists of all agents in the same agent set are identical. Master lists provide a natural way to represent the fact that in practice agent preferences are often not independent. Examples of their real-life applications occur in resident matching programs [4], dormitory room assignments [46], cooperative download applications such as BitTorrent [2], and 3-sided networking services [16].

The precise method to create an instance from each class is as follows:

- **Random**: generated randomly from uniform distribution.
- **ML1swap**: all agents in the same agent set follow the same randomly chosen master list. Then in each preference list, the positions of two randomly chosen agents are swapped.

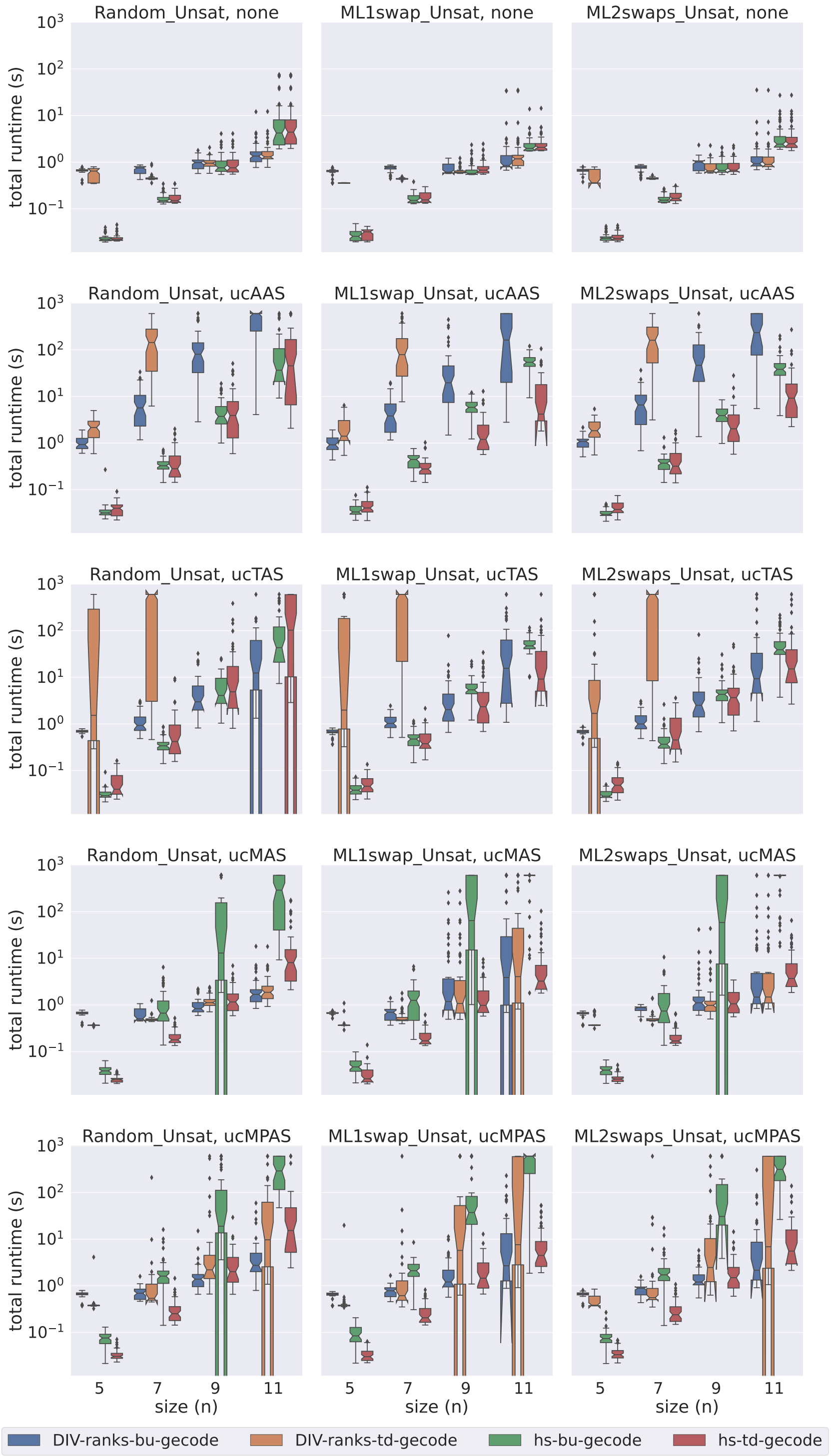


Figure 1 A comparison of total time spent by all Gecode models on the unsatisfiable unit-cost instances.

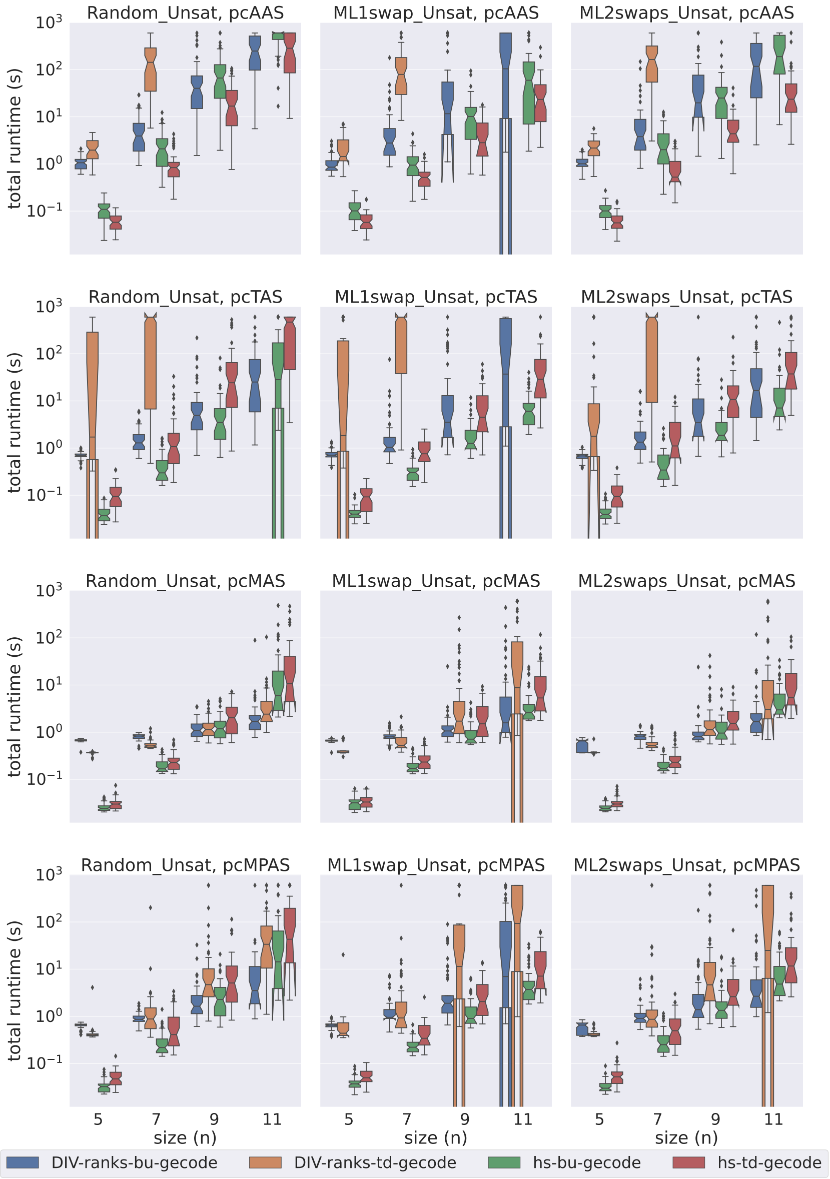


Figure 2 A comparison of total time spent by all Gecode models on the unsatisfiable popularity-cost instances.

- **ML2swaps**: each agent set has a randomly chosen master list that all agents in the set follow. First, two agents are randomly chosen from each agent's preference list, and their positions are swapped. Then, two more agents from each list are randomly chosen such that the new agents were not involved in the first swap, and their positions are swapped.

For each instance class and each odd size $n \in \{5, 7, \ldots, 19\}$, we generated instances with n agents in each agent set, solved the instances under strong stability, and kept the first 50 that were satisfiable and the first 50 that were unsatisfiable. This gave us a total of 300 instances for each size, 150 with a strongly stable matching and 150 without. We had to restrict ourselves to strong stability for unsatisfiability, because the smallest known instance without a weakly stable matching is of size 90 [36], so it would not have been feasible to obtain a representative sample of reasonably-sized unsatisfiable instances for weak stability.

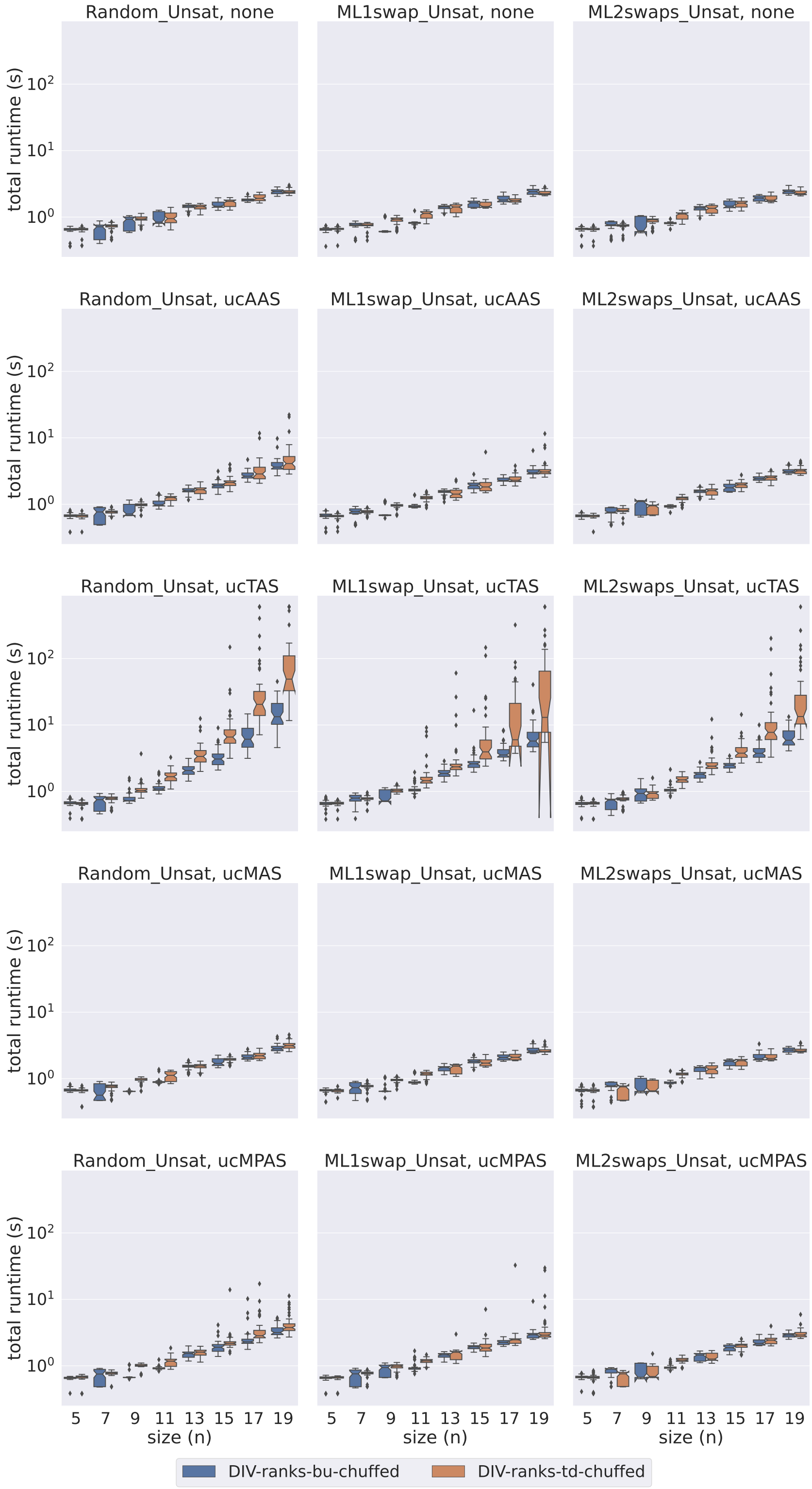


Figure 3 A comparison of total time spent by all Chuffed models on the unsatisfiable unit-cost instances.

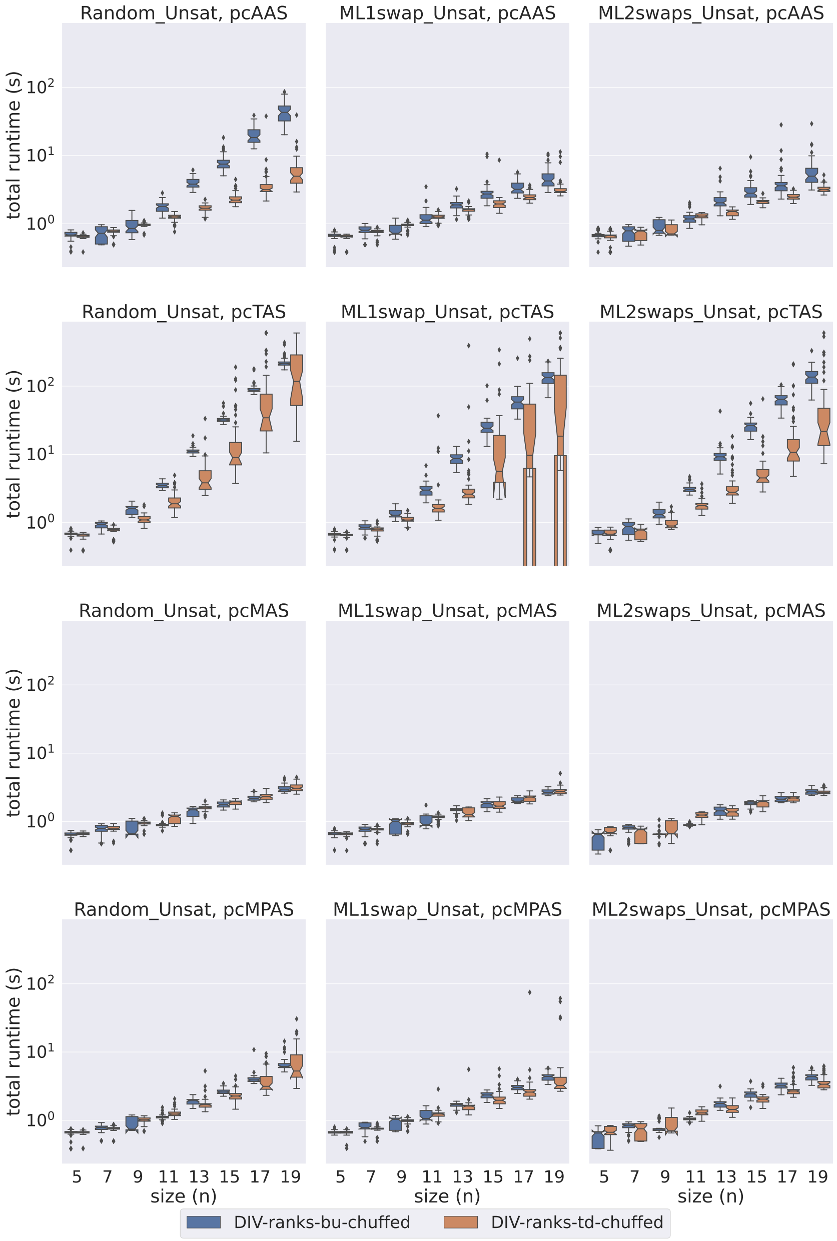


Figure 4 A comparison of total time spent by all Chuffed models on the unsatisfiable popularity-cost instances.

The three types of instances that we studied have been previously used to test the DIV-ranks and HS models, along with a fourth class named ML_oneset [13]. Since ML_oneset instances always admit a strongly stable matching [13], we did not include this additional instance class in our experiments.

4.1.2 Cost formulas

For each configuration of the model, solver, and relaxation type, each instance was set up with two definitions of costs on its elements. The first one is a unit-cost relaxation, corresponding to a cost of 1 for every agent, pair, and triple in the instance. For the second one, that we call *popularity-cost relaxation*, the cost of an agent is a measure of how well she is ranked in other agents' preference lists. Formally the cost of an agent $b \in B$ is defined as:

$$\mathrm{Cost}(b) = \sum_{i=1}^{n} n - \mathrm{rank}_{a_i}(b).$$

The costs of agents from A and C are defined analogously. The intent is to penalise putting popular agents in the correction set, by giving a higher cost to better ranked agents. The cost of a pair (respectively triple) of agents is the sum of the individual costs of the two (respectively three) agents composing it.

4.2 Scalability

In this section we evaluate the performance of DIV-ranks and HS by considering how well they scale with respect to the number of agents in the set. We have decided to classify the experiments into eight groups depending on: (a) the satisfiability of the instance, (b) the solver used and (c) whether the soft constraints have unit cost or not.

The focus of this paper is on dealing with unsatisfiable instances. However, since in practice we cannot always know in advance whether an instance admits a solution, we found it important to check that the satisfiable cases are solved efficiently too. As the instances are satisfiable, the cost of the optimal relaxation is 0 for each one of them, regardless of the type of relaxation. While we could not include the results because of lack of space, all approaches deal with satisfiable instances without major issue.

In Figures 1, 2 3, and 4 we present the results for the unsatisfiable instances. The approaches evaluated are classified in terms of: (a) the model used (DIV-ranks vs HS), (b) the solver used (Gecode vs Chuffed) and (c) the search strategy used (Bottom Up (bu) vs Top Down (td)). *Bottom Up* consists of branching on the cost variable first by selecting the smallest value in the domain first. Effectively this means that we follow a succession of unsatisfiable checks and end with a satisfiable check, which is bound to lead to an optimal solution since we have already proved that there is no solution with a smaller cost. With the *Top Down* strategy we do the opposite: we find a solution and keep on restricting the next one to be better until that is no longer possible. Effectively this means that we follow a succession of satisfiable checks and end with an unsatisfiable check. The unsatisfiable check ensures that the last satisfiable check corresponds to an optimal solution [18].

Our first observation is that the Chuffed approaches clearly outperform the Gecode approaches. As demonstrated by Figures 3 and 4, all Chuffed approaches solve the vast majority of instances of size 15 in less than 10 seconds, while the Gecode approaches struggle with instances of size 11 in quite a few cases. The Chuffed approaches also result in much fewer failures – in some cases the gap is of more than two orders of magnitude.

We consider 9 relaxation types. The first one (none) corresponds to the case where all soft constraints are considered hard. This category was included to gauge the amount of overhead added by modeling each type of relaxation. The other 8 categories correspond to the unit-cost and popularity-cost versions of the four relaxation options introduced in Section 2.2.

In general we observe that our approaches deal much better with MAS and MPAS than with TAS and AAS. In instances where all relaxation types lead to the same optimal relaxation, we can save a considerable amount of time by computing one of our two relaxation types. When it comes to the type of cost, this does not seem to deteriorate much the performance of the Chuffed approaches. In the Gecode approaches we actually observe an improvement in performance when we consider our popularity-cost instances in most of the cases. The situation might be different for instances with completely arbitrary costs.

The Bottom Up vs Top Down comparison is another point where we observe differences between the Chuffed and the Gecode approaches. In the Chuffed approaches, even though in most of the cases we did not observe major differences, in some cases the Top Down exploration led us to visibly better results. The situation in Gecode is quite the opposite. The very same model (DIV-ranks) presented very different behaviours depending on whether Top Down or Bottom Up was used. The Bottom Up tests were completed for all the (small) sizes. However, we had to discard some of the Top Down tests since it was already known that they were going to time out. It is important to remark, though, that the Bottom Up strategy did not always lead to improvements. The improvements were mostly observed when dealing with AAS/TAS instances. Similarly, we observed differences in the performance of HS with respect to the Bottom Up vs Top Down comparison. The Bottom Up strategy led us to better results when dealing with the popularity-cost instances in most of the cases.

We test the scalability of the different relaxation versions on a few large instances in Appendix A.

5 Conclusion and future work

We extended 3DSM-CYC constraint models to four relaxation versions of the problem, two based on already established two-dimensional relaxation notions, and two that we introduced. For each of these four relaxations, we tested our models on instances of various sizes and types, for two different cost functions, and using both a bottom-up and a top-down approach. Our results show that our models are able to efficiently compute a maximum relaxation for unsatisfiable 3DSM-CYC instances.

While our relaxation models performed well for the two cost functions that we studied, it would be interesting to know in what ways their behavior would be affected when given different formulas for the costs of the elements in the instance. For example, one could set the cost of a triple as the difference between the highest and lowest costs of its agents, mirroring the definition of sex-equal [32, 55, 40, 51] optimisation for satisfiable instances. It would be also interesting to find out how the presence of mandatory agents/pairs/triples affect the performance since these constraints are highly motivated in the literature [17, 23, 35].

Another possible avenue of research would be to explore the relations between minimum correction sets of different relaxation types. If for a particular class of instances the maximum relaxations are identical for different types, then one could use our findings that the two new relaxation versions lead to better performance, and search for minimally-accommodating stable matchings instead of almost stable matchings to get the same result faster.

References

1 David J. Abraham, Péter Biró, and David F. Manlove. “Almost stable” matchings in the roommates problem. In Thomas Erlebach and Giuseppe Persiano, editors, *Proceedings of WAOA '05: the 3rd Workshop on Approximation and Online Algorithms*, volume 3879 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2006.

2 David J Abraham, Ariel Levavi, David F Manlove, and Gregg O'Malley. The stable roommates problem with globally-ranked pairs. *Internet Mathematics*, 5:493–515, 2008.

3 Péter Biró. Applications of matching models under preferences. *Trends in Computational Social Choice*, page 345, 2017.

4 Péter Biró, Robert W Irving, and Ildikó Schlotter. Stable matching with couples: An empirical study. *Journal of Experimental Algorithmics (JEA)*, 16:Article no. 1.2, 2011.

5 Péter Biró, David F. Manlove, and Eric J. McDermid. “Almost stable” matchings in the roommates problem with bounded preference lists. *Theoretical Computer Science*, 432:10–20, 2012.

6 Péter Biró, David F. Manlove, and Shubham Mittal. Size versus stability in the marriage problem. *Theoretical Computer Science*, 411:1828–1841, 2010.

7 Péter Biró and Eric McDermid. Three-sided stable matchings with cyclic preferences. *Algorithmica*, 58(1):5–18, 2010. doi:10.1007/s00453-009-9315-2.

8 Francis Bloch, David Cantala, and Damián Gibaja. Matching through institutions. *Games Econ. Behav.*, 121:204–231, 2020. doi:10.1016/j.geb.2020.01.010.

9 Niclas Boehmer, Robert Bredereck, Klaus Heeger, and Rolf Niedermeier. Bribery and control in stable marriage. *Journal of Artificial Intelligence Research*, 71:993–1048, 2021.

10 Endre Boros, Vladimir Gurvich, Steven Jaslar, and Daniel Krasner. Stable matchings in three-sided systems with cyclic preferences. *Discrete Mathematics*, 289(1-3):1–10, 2004. doi:10.1016/j.disc.2004.08.012.

11 Jiehua Chen, Danny Hermelin, Manuel Sorge, and Harel Yedidsion. How hard is it to satisfy (almost) all roommates? In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

12 Geoffrey Chu. *Improving combinatorial optimization*. PhD thesis, University of Melbourne, Australia, 2011. URL: https://hdl.handle.net/11343/36679.

13 Ágnes Cseh, Guillaume Escamocher, Begüm Genç, and Luis Quesada. A collection of constraint programming models for the three-dimensional stable matching problem with cyclic preferences. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021), Montpellier, France, 25-29 October 2021*, pages 1–19. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021.

14 Ágnes Cseh, Robert W. Irving, and David F. Manlove. The stable roommates problem with short lists. *Theory of Computing Systems*, 63(1):128–149, 2019.

15 Ágnes Cseh and Jannik Peters. Three-dimensional popular matching with cyclic preferences. *CoRR*, abs/2105.09115, 2021. arXiv:2105.09115.

16 Lin Cui and Weijia Jia. Cyclic stable matching for three-sided networking services. *Comput. Networks*, 57(1):351–363, 2013. doi:10.1016/j.comnet.2012.09.021.

17 Vânia M.F. Dias, Guilherme D. Da Fonseca, Celina M.H. De Figueiredo, and Jayme L. Szwarcfiter. The stable marriage problem with restricted pairs. *Theoretical Computer Science*, 306:391–405, 2003.

18 Ulrich Dorndorf, Erwin Pesch, and Toàn Phan-Huy. Solving the open shop scheduling problem. *Journal of Scheduling*, 4(3):157–174, 2001.

19 Kimmo Eriksson and Olle Häggström. Instability of matchings in decentralized markets with various preference structures. *International Journal of Game Theory*, 36(3-4):409–420, 2008.

20 Kimmo Eriksson, Jonas Sjöstrand, and Pontus Strimling. Three-dimensional stable matching with cyclic preferences. *Mathematical Social Sciences*, 52(1):77–87, 2006. doi:10.1016/j.mathsocsci.2006.03.005.

21 Tomás Feder. A new fixed point approach for stable networks and stable marriages. *Journal of Computer and System Sciences*, 45(2):233–284, 1992. doi:10.1016/0022-0000(92)90048-N.

22 Tomás Feder. Network flow and 2-satisfiability. *Algorithmica*, 11(3):291–319, 1994. doi:10.1007/BF01240738.

23 Tamás Fleiner, Robert W. Irving, and David F. Manlove. Efficient algorithms for generalised stable marriage and roommates problems. *Theoretical Computer Science*, 381:162–176, 2007.

24 David Gale and Lloyd S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 120(5):386–391, 1962. doi:10.4169/amer.math.monthly.120.05.386.

25 Gecode Team. Gecode: Generic constraint development environment, 2019. Available from http://www.gecode.org.

26 Ian P. Gent, Robert W. Irving, David F. Manlove, Patrick Prosser, and Barbara M. Smith. A constraint programming approach to the stable marriage problem. In *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239, pages 225–239. Springer, 2001. doi:10.1007/3-540-45578-7_16.

27 Sushmita Gupta, Pallavi Jain, Sanjukta Roy, Saket Saurabh, and Meirav Zehavi. On the (Parameterized) Complexity of Almost Stable Marriage. In *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2020)*, volume 182 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

28 Dan Gusfield and Robert W. Irving. *The Stable marriage problem - structure and algorithms*. MIT Press, 1989.

29 Koki Hamada, Kazuo Iwama, and Shuichi Miyazaki. An improved approximation lower bound for finding almost stable maximum matchings. *Information Processing Letters*, 109:1036–1040, 2009.

30 Stephen P. Heyneman, Kathryn H. Anderson, and Nazym Nuraliyeva. The cost of corruption in higher education. *Comparative Education Review*, 52(1):1–25, 2008.

31 Robert W. Irving, Paul Leather, and Dan Gusfield. An efficient algorithm for the "optimal" stable marriage. *Journal of the ACM*, 34(3):532–543, 1987. doi:10.1145/28869.28871.

32 Akiko Kato. Complexity of the sex-equal stable marriage problem. *Japan Journal of Industrial and Applied Mathematics*, 10:1–19, 1993.

33 Samir Khuller, Stephen G. Mitchell, and Vijay V. Vazirani. On-line algorithms for weighted bipartite matching and stable marriages. *Theoretical Computer Science*, 127:255–267, 1994.

34 Donald E. Knuth. *Mariages Stables*. Les Presses de L'Université de Montréal, 1976. English translation in *Stable Marriage and its Relation to Other Combinatorial Problems*, volume 10 of CRM Proceedings and Lecture Notes, American Mathematical Society, 1997.

35 Augustine Kwanashie. *Efficient algorithms for optimal matching problems under preferences*. PhD thesis, University of Glasgow, 2015.

36 Chi-Kit Lam and C. Gregory Plaxton. On the existence of three-dimensional stable matchings with cyclic preferences. *Theory of Computing Systems*, pages 1–17, 2021.

37 Qijun Liu and Yaping Peng. Corruption in college admissions examinations in china. *International Journal of Educational Development*, 41:104–111, 2015.

38 David F. Manlove. *Algorithmics of Matching Under Preferences*, volume 2. WorldScientific, 2013. doi:10.1142/8591.

39 David F. Manlove, Gregg O'Malley, Patrick Prosser, and Chris Unsworth. A constraint programming approach to the hospitals / residents problem. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 4th International Conference, CPAIOR 2007, Brussels, Belgium, May 23-26, 2007, Proceedings*, volume 4510, pages 155–170. Springer, 2007. doi:10.1007/978-3-540-72397-4_12.

40 Eric McDermid and Robert W. Irving. Sex-equal stable matchings: Complexity and exact algorithms. *Algorithmica*, 68(3):545–570, 2014.

41 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

42 Cheng Ng and Daniel S. Hirschberg. Three-dimensional stable matching problems. *SIAM Journal on Discrete Mathematics*, 4(2):245–252, 1991. doi:10.1137/0404023.

43 Gregg O'Malley. *Algorithmic aspects of stable matching problems.* PhD thesis, University of Glasgow, UK, 2007. URL: http://theses.gla.ac.uk/64/.

44 Nikita Panchal and Seema Sharma. An efficient algorithm for three dimensional cyclic stable matching. *International Journal of Engineering Research and Technology*, 3(4), 2014.

45 Kanstantsin Pashkovich and Laurent Poirrier. Three-dimensional stable matching with cyclic preferences. *Optimization Letters*, 14(8):2615–2623, 2020. doi:10.1007/s11590-020-01557-4.

46 Nitsan Perach, Julia Polak, and Uriel G. Rothblum. A stable matching model with an entrance criterion applied to the assignment of students to dormitories at the Technion. *International Journal of Game Theory*, 36(3-4):519–535, 2008. doi:10.1007/s00182-007-0083-4.

47 Neetu Raveendran, Yiyong Zha, Yunfei Zhang, Xin Liu, and Zhu Han. Virtual core network resource allocation in 5g systems using three-sided matching. In *2019 IEEE International Conference on Communications, ICC 2019, Shanghai, China, May 20-24, 2019*, pages 1–6. IEEE, 2019. doi:10.1109/ICC.2019.8762095.

48 Tina Rezvanian. *Integrating Data-Driven Forecasting and Large-Scale Optimization to Improve Humanitarian Response Planning and Preparedness.* PhD thesis, Northeastern University, 2019.

49 Alvin E. Roth and Xiaolin Xing. Turnaround time and bottlenecks in market clearing: Decentralized matching in the market for clinical psychologists. *Journal of Political Economy*, 105(2):284–329, 1997.

50 Mohamed Siala and Barry O'Sullivan. Revisiting two-sided stability constraints. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 342–357. Springer, 2016.

51 Mohamed Siala and Barry O'Sullivan. Rotation-based formulation for stable matching. In *International Conference on Principles and Practice of Constraint Programming*, pages 262–277. Springer, 2017.

52 Mallory Soldner. *Optimization and measurement in humanitarian operations: Addressing practical needs.* PhD thesis, Georgia Institute of Technology, 2014.

53 Chung-Piaw Teo and Jay Sethuraman. LP based approach to optimal stable matchings. In Michael E. Saks, editor, *Proceedings of SODA '97: the 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 710–719. ACM-SIAM, 1997.

54 Chung-Piaw Teo and Jay Sethuraman. The geometry of fractional stable matchings and its applications. *Mathematics of Operations Research*, 23:874–891, 1998.

55 Chris Unsworth and Patrick Prosser. An n-ary constraint for the stable marriage problem. In *Proceedings of the 5th Workshop on Modelling and Solving Problems with Constraints, held at IJCAI '05: the 19th International Joint Conference on Artificial Intelligence*, pages 32–38, 2005.

56 Chris Unsworth and Patrick Prosser. A specialised binary constraint for the stable marriage problem. In *Abstraction, Reformulation and Approximation, 6th International Symposium, SARA 2005, Airth Castle, Scotland, UK, July 26-29, 2005, Proceedings*, volume 3607, pages 218–233. Springer, 2005. doi:10.1007/11527862_16.

A Scalability on larger instances

To see how well our relaxation models scale on larger instances, we chose the best performing approach for each configuration and ran it on unsatisfiable instances with more than 20 agents in each agent set. These instances, 20 in total, were the ones that were determined

Table 2 Largest solved instance sizes and smallest unsolved instance sizes for each relaxation version when run with a timeout of one hour, using the DIV-ranks model and the Chuffed solver. *: There were two ML1swap instances of size 29 in the dataset. A popularity-cost TAS matching was found before timeout for one but not the other.

Relaxation	Random		ML1swap		ML2swaps	
	largest solved	smallest unsolved	largest solved	smallest unsolved	largest solved	smallest unsolved
none	35	-	110	-	90	-
unit-cost AAS	35	-	90	110	70	90
popularity-cost AAS	32	35	29	90	70	90
unit-cost TAS	23	29	29	90	35	45
popularity-cost TAS	23	29	29*	29*	29	35
unit-cost MAS	35	-	90	110	70	90
popularity-cost MAS	35	-	90	110	70	90
unit-cost MPAS	35	-	90	110	70	90
popularity-cost MPAS	32	35	90	110	70	90

unsatisfiable for strong stability in the experiments by Cseh et al. [13]. We chose the DIV-ranks model with the Chuffed solver, using the Bottom Up strategy for unit-cost relaxation and Top Down for popularity-cost, because this showed the best performance in our other tests. The results, displayed in Table 2, confirm that it is more efficient to compute MAS relaxations, although AAS and MPAS also scale well for some combinations of instance class and cost function.

DUELMIPs: Optimizing SDN Functionality and Security

Timothy Curry ✉
University of Connecticut, Storrs, CT, USA

Gabriel De Pace ✉
University of Rhode Island, Kingston, RI, USA

Benjamin Fuller ✉
University of Connecticut, Storrs, CT, USA

Laurent Michel ✉
University of Connecticut, Storrs, CT, USA

Yan (Lindsay) Sun ✉
University of Rhode Island, Kingston, RI, USA

Abstract

Software defined networks (SDNs) define a programmable network fabric that can be reconfigured to respect global networks properties. Securing against adversaries who try to exploit the network is an objective that conflicts with providing functionality. This paper proposes a two-stage mixed-integer programming framework. The first stage automates routing decisions for the flows to be carried by the network while maximizing readability and ease of use for network engineers. The second stage is meant to quickly respond to security breaches to automatically decide on network counter-measures to block the detected adversary. Both stages are computationally challenging and the security stage leverages large neighborhood search to quickly deliver effective response strategies. The approach is evaluated on synthetic networks of various sizes and shown to be effective for both its functional and security objectives.

2012 ACM Subject Classification Theory of computation → Network optimization; Networks → Network security; Security and privacy → Trust frameworks

Keywords and phrases Network security, mixed integer programming, large neighborhood search

Digital Object Identifier 10.4230/LIPIcs.CP.2022.17

Funding This work is supported by ONR Grant N00014-19-1-2327. B. F. is supported by NSF Awards #1849904, 2141033. L.M. and T. C. are partially supported by the Synchrony Chair in Cybersecurity. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. government.

1 Introduction

Software-Defined Networks (SDN) create a programmable network framework, potentially allowing organizations to simultaneously account for functionality, usability, and trust. Optimization techniques have been used to improve an SDN configuration *along a single dimension*: *Functionality* [2,17,24,36], *Usability* [21,28,32] or *Trust* [6,35]. Little work [10,19] integrates these goals because of the antagonistic objectives of the three dimensions. For instance, a fully permissive functionality policy based on shortest path optimizes routing but reduces trust as any adversary has unrestricted access through the network. For those readers unfamiliar with network attacks, Example 1 presents a toy network and attack.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 17; pp. 17:1–17:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This work produces optimization models that simultaneously consider routing, usability, and trust of the proposed SDN configuration. We consider two use cases:

Planning. When a new application is being stood up and network engineers want a configuration. To explain configurations, it is crucial to use models with clear objectives that prove solution quality. One is comfortable with a moderate computation time.

Rapid Response. Adapting when a security event occurs. Attackers rely on the sluggishness of defense responses to pivot within an organization [12]. In this setting, one needs to prepare a response in minutes. Automated responses are used to allow a security analyst to craft a permanent response.

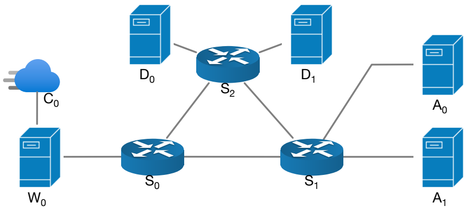
This work introduces DuelMIPs a *reactive* two-stage online mathematical programming model in which a mixed-integer programming (MIP) method incorporates conflicting requirements of routing, usability, and trust. The goal is, given a set of required flows and disallowed flows, to find a configuration that either routes or blocks flows while minimizing (1) the complexity of the resulting configuration, (2) the residual trust reduction if some device in the network is possibly compromised. The **Routing Model** is computationally heavy and runs first to produce a functional and usable configuration. The routing model, which one executes whenever functional requirements have substantively changed, produces routing decisions for all SDN devices in the network. It synthesizes small configurations that are easy to understand and minimizes conflicts and defects that hurt interpretability [8, 34]. The production of an optimized configuration is followed by a deployment of the decisions onto the SDN devices.

Network appliances detect security events [20, 25]. The **Trust Model** is run in response to such a detection. This new information affects trust levels of the targeted devices and related devices, allowing the Trust model to decide on actions to mitigate the detected threat. Depending on which device was targeted, the network flows, and the loss of trust, the second stage decides on traffic restrictions to minimize the global trust reduction. It proposes blocking rules meant to hamper lateral movements within the network.

This paper offers the following contributions:

- It adapts multi-commodity flows, common-place in supply chains, to the specifics of routing in data networks. It handles wildcards in routing rules and the production of SDN programs (rule sets) at SDN devices.
- It articulates a trust model that is suitable for rapid response to security events. This model is a physics-based approach using an elastic string networks analogy to capture inter-node trust relations and mitigate the impact on trust with counter-measures.

▶ **Example 1.** Consider the network topology and routing policy presented in Figure 1. This is a simplified example from our evaluation topology described in Section 5. The three parts of Figure 1 are the inputs for the DuelMIPs's routing model. Figure 1a shows the physical connection between the SDN routers and end hosts. The goal of the **Routing Model** is to install rulesets on the routers such that the flows in the required (Figure 1b) and restricted tables (Figure 1c) will be serviced and blocked, respectively. In both cases, flows are described by a triple of source (W_0), destination (A_0), and application-level protocol (`SQL`). There are multiple paths that could be chosen for each flow (the set of possible paths is defined by the topology) and *many* different valid rulesets to satisfy each pathing choice. Specifically, SDN routers route a packet according to the most specific (and first) rule that applies to that packet, describing which physical *switch port* the packet should be sent on (or not forwarded). This is known as the longest prefix match rule. We slightly abuse notation in Table 1 by listing the device connected through the *switch port*.



(a) Network topology.

Required
$W_0 \xleftrightarrow{\texttt{HTTP}} A_0$
$W_0 \xleftrightarrow{\texttt{HTTP}} A_1$
$A_0 \xleftrightarrow{\texttt{SQL}} D_0$
$A_1 \xleftrightarrow{\texttt{SQL}} D_1$

(b) Required flows.

Restricted
$W_0 \xleftrightarrow{\texttt{HTTP}} D_0$
$W_0 \xleftrightarrow{\texttt{HTTP}} D_1$
$W_0 \xleftrightarrow{\texttt{SQL}} D_0$
$W_0 \xleftrightarrow{\texttt{SQL}} D_1$

(c) Restricted flows.

Figure 1 Network topology, routing policy, and inputs for DuelMIPs's routing model.

Table 1 Rulesets for SDN routers that satisfy the requirements presented in Figure 1. These rules are part of the output of DuelMIPs in this example. Note the red, bold rule at the top of S_2's ruleset is part of the trust model output, while all other rules come from the routing model.

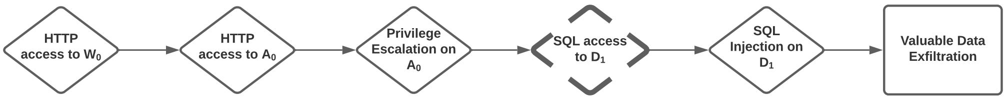
S_0 Ruleset		S_1 Ruleset		S_2 Ruleset	
Match	Action	Match	Action	Match	Action
$(W_0, D_0, *)$	DROP	$(*, W_0, *)$	SEND(S_0)	$(\mathbf{D_1}, \mathbf{A_0}, \texttt{SQL})$	**DROP**
$(W_0, D_1, *)$	DROP	$(*, A_0, *)$	SEND(A_0)	$(*, A_0, *)$	SEND(S_1)
$(*, W_0, *)$	SEND(W_0)	$(*, A_1, *)$	SEND(A_1)	$(*, A_1, *)$	SEND(S_1)
$(*, A_0, *)$	SEND(S_1)	$(*, D_0, *)$	SEND(S_2)	$(*, D_0, *)$	SEND(D_0)
$(*, A_1, *)$	SEND(S_1)	$(*, D_1, *)$	SEND(S_2)	$(*, D_1, *)$	SEND(D_1)

The routing model in DuelMIPs can generate rules with match fields containing up to two wildcards. (As rules are described by a triple, three wildcards would mean that the same rule is applied universally which is not appropriate for a router.) Furthermore, the model prefers to use rules with more wildcards (which match any value), as it allows for more compact and readable rulesets. This is in line with ruleset configurations designed by human network engineers [28]. Producing rules with wildcards is important as network engineers will only deploy rulesets that they can understand and modify. An example configuration that could be produced by DuelMIPs is in Table 1.

One downside of using wildcard-based rules is that they allow *incidental* flows to be serviced within the network. Consider the solution to the given input represented by Table 1 (minus the bolded, red rule in S_2's table). This is the output of DuelMIPs's routing model. The use of wildcards makes the ruleset readable. However, it routes flows that are not included in the network policy. A packet of type $A_0 \xrightarrow{\texttt{SQL}} D_1$ will be routed $A_0 \longrightarrow S_1 \longrightarrow S_2 \longrightarrow D_1$ because of S_1's fifth rule and S_2's fifth rule. An analogous route exists for the return traffic $D_1 \xrightarrow{\texttt{SQL}} A_0$. Wildcard usage is cruical for ruleset compactness, utility, and readability, but it can permit incidental internal flows that enable network attacks.

The **Trust Model** reduces the trust loss due to incidental flows in cases when an end host is marked as suspicious. Suppose a network appliance alerts DuelMIPs that W_0 has received suspicious traffic from an external client, C_0. Modern network attacks often require multiple steps to reach their goal [22], called exploit chains. The suspicious traffic at W_0 is due to an attacker who is attempting to leverage the exploit chain (see Figure 2). The trust model is run with W_0 as the focus of distrust. DuelMIPs observes the incidental SQL flow between D_1 and A_0. Since A_0 is directly connected to W_0, there is a *transitive* trust loss induced on D_1. DuelMIPs reduces this loss by blocking incidental flows with firewalls (if the improvement surpasses the complexity cost of adding the firewalls). In this case, a drop rule is added to S_2's routing table, shown in red. This rule does not block any required flows.

Figure 2 Exploit chain using in a pivot attack on the example network. Here the only link that DuelMIPs could affect via a routing configuration is shown with a dashed outline.



Organization. The rest of this work is organized as follows: Section 2 provides an overview of the model goals, Section 3 presents the models, Section 4 the assessment metrics, Section 5 our experimental setup, Section 6 the results, and Section 7 concludes.

2 DuelMIPs objectives

This section outlines the routing (Sec 2.1) and trust (Sec 2.2) objectives. Section 3 describes how they give rise to mathematical programming models. In the following, a network is formed from set of *devices* $D = H \cup R$ that are either hosts H or SDN routers R. Assume that $|D| = n$. Devices are connected by network arcs (wires) from $A \subseteq D \times D$. Giving rise to a graph $\mathcal{G}(D, A)$. A data flow $f = \langle s, d, t\rangle$ indicates the requirement to transport data packets for an application-level protocol t from a source device s (typically a host) to a destination device d (again, typically a host). A protocol t is from $\mathcal{P} = \{\texttt{SSL}, \texttt{HTTP}, \texttt{LDAP}, \texttt{MYSQL}, \texttt{SMTP}, \cdots\}$ with $|\mathcal{P}| = p$. Let $\mathcal{F} = D^2 \times \mathcal{P}$ be the universe of all possible flows and $|\mathcal{F}| = n^2 \cdot p$.

2.1 Routing Objective

The network configuration problem is given a set of required $F^+ \subseteq \mathcal{F}$ and disallowed $F^- \subseteq \mathcal{F}$ flows. The objective is to find a set of rules, for each router, which routes all flows in F^+ and ensures that all flows in F^- are blocked. We consider routing rules of the type $\langle \texttt{s}, \texttt{d}, \texttt{t}\rangle \Rightarrow \texttt{action}$ in which $s, d \in D \cup \{*\}$, $t \in \mathcal{P} \cup \{*\}$ and the wildcard `*` means that any value in the corresponding set is permissible. Headers of the packet being routed [26] are against s, d, and t to select the applicable rule from the routing table. The `action` could be dropping the packet, or forwarding to a specific neighboring router g, i.e., $\texttt{action} \in \{\texttt{DROP}, \texttt{SEND}(g)\}$. For example, $\langle *, \texttt{192.168.1.10}, \texttt{HTTP}\rangle \Rightarrow \texttt{DROP}$ states that any inbound `HTTP` traffic meant for `192.168.1.10` should be dropped. Observe how a route for a flow f is implemented by routing rules in network devices along the path. For instance, in Figure 1, the flow $W_0 \overset{\texttt{HTTP}}{\longleftrightarrow} A_1$ uses the fourth wildcard routing rule in S_0 and the third in S_1 to reach its destination A_1.

When configuring a network one must determine 1) the paths for all flows in F^+ and 2) the table rules to be installed in routers. Specifically, the routing objective is the following:

1. Ensure that all flows in F^+ have a single path through the network (known as single path routing used in intranet routing [23] and Internet routing [27]),
2. Ensure that all flows in F^- are blocked,
3. Ensure for each r on the path of $f \in F^+ \cup F^-$ there is *at most one wildcard rule* that matches f. We do not require that $f \in F^-$ are blocked at the first possible device so this constraint is applied to both F^+ and F^-,
4. Minimize the total length of routes for required flows, and
5. Minimize the complexity of SDN rules that are deployed on each device.

Property 3 improves usability of the resulting SDN rules. See Section 4 which builds on [5, 33]. This objective ensures that routes are set for all required flows that avoid routing loops and minimize latency (shortest paths). This should be achieved with a small and simple rule set that is *usable* by network engineers, i.e., the rules must be natural and easily explainable. While wildcards reduce the complexity of a rule set, they *also* authorize *incidental flows* that were not explicitly required nor explicitly forbidden, this set is denoted by I. It is this set of *incidental* flows that is the focus of the trust model. We use F_r to denote the set of all routable flows, mathematically $F_r = F^+ \cup I$. We use $F_r(a, b)$ to restrict to all flows between hosts a and b with analogous notation for F^+, F^- and I.

2.2 Trust Objective

The trust model minimizes the impact of a *detected* security event on the *overall trust* in the network. It is used in the setting when an attacker is trying to pivot to collect important *privileges* [22], usually resulting in compromising, i.e., gaining unfettered access to some critical component such as a customer database. For instance, one could compromise the Gmail password of an HR representative and try this password in the HR system, learn biographic information about a system administrator and use it to carry out a spear phishing campaign. In many modern exploit chains, the attacker relies on *similarity* between systems they are compromising to carry out such a pivot and gain new privileges.

This section describes basic trust modeling concepts [1, 14, 18, 37]. A trust model is defined over an undirected graph $\mathcal{G}'(H, F_r)$ as above H represents network end devices or hosts, while F_r is the set of flows that can be routed in the network. Note this graph $\mathcal{G}'$ is different than $\mathcal{G}(D, A)$ considered in the routing model. We consider two time values `Init` and `Final`, however, all notation and modeling can naturally be extended to a sequence of events over time.

Trust. Let the trust of a host $h \in H$, be $\texttt{Trust}(h) \in \mathbb{R}^+$. Let $\texttt{Trust}^{\texttt{Init}}(h)$ and $\texttt{Trust}^{\texttt{Final}}(h)$ refer to the initial and final trust values for a host reflecting its hardening [29].

Similarities. *Incidental* flows increase connectivity between hosts and offer additional pathways to reach other hosts. This paper codifies similarity as the fraction of *incidental flows* over *total flows* between two hosts. That is, given a universe of possible flows $F_r(a, b)$ between two distinct hosts a and b, a set of required flows $F^+(a, b)$ between them and $I(a, b)$ as the set of incidental flows between a and b, one can define

$$s_{a,b} \overset{def}{=} |F^+(a, b) \cup I(a, b)|/|F_r(a, b)| \in [0, 1] \tag{1}$$

as the similarity between hosts a and b [16, 40]. It is affected by paths in $\mathcal{G}'$ connecting a to b and it affects how the trust of a device changes as a result of security events at nearby connected devices.

Event. Let $e(h_i)$ denote a security event affecting host $h_i \in H$. The event $e(h_i)$ induces a *direct trust reduction* defined as $\texttt{TrustRed}(h_i) = \texttt{Trust}^{\texttt{Init}}(h_i) - \texttt{Trust}^{\texttt{Final}}(h_i) \geq 0$.

Event Set. An event set $\mathcal{E}$ conveys the simultaneous occurrence of multiple events.

Trust models propagate trust reductions that one experiences as a result of an event set $\mathcal{E}$. A trust reduction at host h_i *affects* all hosts indirectly connected to h_i. This propagation of *indirect trust reductions* is captured with an elastic string model which will be described in Section 3.2. The objective is to choose traffic blocking measures that minimize the total trust reduction.

3 Optimization Models

The optimization framework consists of two models, a **Routing Model** and a **Trust Model**. The Routing model produces network configurations that meet functional requirements and are usable by network engineers (we describe metrics in Section 4). The Trust model minimizes trust reduction of a presented network compromise.

The coupling between the two models is achieved through *similarity variables* defined in Section 2.2. These variables are derived from the routing model solution and can be altered due to trust model decisions. An increased similarity between two nodes can either increase or reduce trust levels. Simpler, shorter and more usable configurations often leverage routing wildcards that adversely impact node similarities and prompt the creation of more complex defensive measures. This tension between desirable and secure configurations is the heart of the problem hardness.

3.1 Routing Model

The routing model decides on the content of all routing tables to carry required flow, block restricted flows while minimizing routing cost and maximizing the usability of the rule sets to be installed on SDN devices. The input is the graph $\mathcal{G}(D, A)$, required flows F^+, and disallowed flows F^-. The mathematical formulation is a multi-commodity network flow. Required flows originate from a super-source node, flow into their actual source node (the host from which the flow originates) and must reach a super-sink via the actual destination node. An unusual structure in the model results from wildcards, a feature not normally seen in multi-commodity flows. There are, for each network device, Boolean variables to convey forwarding or dropping actions for each possible packet header that condition the routing rule (the $r^s_{i,j}$ and w^s_i variables below).

Parameters

F^+, F^- – set of required (resp. restricted) flows that must be routed (resp. blocked).
$F = F^+ \cup F^-$ – set of all flows in network routing policy.
S – set of possible SDN routing table headers in network.
S_f – set of all SDN routing table headers that flow f matches.
$S^*_f \subset S_f$ – set of SDN routing table headers with wildcards that flow f matches.
R, H, $D = H \cup R$ – set of routers, hosts and all devices in the network.
$A \subseteq D \times D$ – set of arcs (links) in the network.
σ,τ – abstract flow network super source and super sink.
$\delta^-(i)$ and $\delta^+(i)$ – set of predecessors, resp. successors of device $i \in D$.
$\mathsf{src}(f)$ and $\mathsf{dst}(f)$ – the source and destination of flow f
$\mathsf{cost}(s)$ – the cost (in terms of complexity) of using header s for a routing rule.
$\mathcal{M}$ – big-M constant.

Decision Variables

$c^f_{i,j} \in \{0, 1\}$ – Link (i, j) *carries* flow f.
$p^f_{i,j} \in \{0, 1\}$ – Link (i, j) is *permitted* (via chosen SDN rules) to carry flow f.
$a^f_i \in \{0, 1\}$ – Flow f *reaches* device $i \in D$.
$r^s_{i,j} \in \{0, 1\}$ – Device i *forwards* flows matching header $s \in S$ to device j.
$w^s_i \in \{0, 1\}$ – Device i *firewalls* flows matching header $s \in S$.

Constraints

The first constraints model network flows (injection 2 and conservation 3,4).

$$\forall f \in F \;:\; c^f_{\sigma,f.src} = 1 \tag{2}$$

$$\forall j \in R \;:\; \sum_{i \in \delta^-(j)} c^f_{i,j} = \sum_{k \in \delta^+(j)} c^f_{j,k} + \sum_{s \in S_f} w^s_j \tag{3}$$

$$\forall j \in H \;:\; \sum_{i \in \delta^-(j)} c^f_{i,j} = \sum_{k \in \delta^+(j)} c^f_{j,k} \tag{4}$$

$$\forall f \in F \;:\; a^f_\sigma = 1 \tag{5}$$

$$\forall f \in F \;:\; p^f_{\sigma,\mathsf{src}(f)} = 1 \tag{6}$$

$$\forall f \in F \;, \forall j \in \delta^+(\sigma) \text{ where } i \neq \mathsf{src}(f) \;:\, p^f_{\sigma,i} = 0$$

Equations 5 - 10 ensure that flows are well formed. Namely, they properly spawn from the source, arrive at the sink, are carried when required (F^+) and blocked when forbidden (F^-).

$$\forall f \in F \;:\; p^f_{\mathsf{dst}(f),\tau} = 1 \tag{7}$$

$$\forall f \in F \;, \forall i \in \delta^-(\tau) \text{ where } i \neq \mathsf{dst}(f) \;:\, p^f_{i,\tau} = 0$$

$$\forall f \in F \;,\; \forall i \in \delta^+(\mathsf{src}(f)) \;:\, p^f_{\mathsf{src}(f),i} = 1 \tag{8}$$

$$\forall f \in F^+ \;:\, c^f_{\mathsf{dst}(f),\tau} = 1 \tag{9}$$

$$\forall f \in F^- \;:\, c^f_{\mathsf{dst}(f),\tau} = 0 \tag{10}$$

The remaining constraints focus on the rule implementations on network devices.

$$\forall (i,j) \in A \;,\; \forall f \in F \;:\; p^f_{i,j} = \bigvee_{s \in S_f} r^s_{i,j} \tag{11}$$

$$\forall (i,j) \in A \;,\; \forall f \in F \;:\; c^f_{i,j} \leq p^f_{i,j} \tag{12}$$

$$\forall j \in D \;,\; \forall f \in F \;:\; a^f_j = \bigvee_{i \in \delta^-(j)} (a^f_i \wedge (p^f_{i,j})) \tag{13}$$

$$\forall (i,j) \in A \;,\; \forall f \in F \;:\; \sum_{s \in S_f} r^s_{i,j} \geq c^f_{i,j} \tag{14}$$

$$\forall j \in R \;,\; \forall f \in F \;:\; \sum_{k \in \delta^+(j)} \sum_{s \in S^*_f} r^s_{j,k} + \sum_{s \in S^*_f} w^s_j \leq 1 + \mathcal{M}(1 - a^f_j) \tag{15}$$

Equation 11 states that a flow is permitted to travel over a link if there is a rule installed on a device that allows it to do so. Equation 12 ensures that a required flow will only travel over a link if that link permits it. Equation 13 enforces that a flow is able to reach a device j only if the flow can reach a direct predecessor and the predecessor has a rule installed that permits the flow to travel over the direct link to j.

Equation 14 requires that a proper forwarding rule is installed if a link is set to carry a flow. Equation 15 states that if a flow reaches a router, then there is at most one wildcard rule covering that flow on the router. This constraint is needed to ensure clear switch behavior (see Section 4). Equation 15 is vacuously true for a flow that does not reach the router.

Routing Objective

The objective function for the routing model is

$$\min \quad \alpha_0 \cdot \sum_{f \in F} \sum_{(i,j) \in A} c^f_{i,j} + \alpha_1 \cdot \sum_{s \in S} \left(\sum_{(i,j) \in A} r^s_{i,j} \cdot \mathsf{cost}(s) + \sum_{i \in R} w^s_i \cdot \mathsf{cost}(s) \right)$$

It minimizes the routing costs (number of links utilized) and the complexity of all rule sets.

3.1.1 Incidental Flow Extraction

Finally, it is desirable to compute the set of incidental flows I introduced by using rules with wildcards. We identify such flows using any complete graph traversal algorithm over the topology and rulesets to identify flows that were not in F^+ nor F^- but are routable.

3.2 Trust Model

The trust model receives information regarding the trustworthiness of network hosts (H), as well as a list of all deliverable flows, i.e., $F_r = F^+ \cup I$. The purpose of the model is to reason about the connections induced by incidental flows (I) that are potentially damaging and should be blocked via additional rules.

This model is inspired by physics. Each host $h \in H$ is modeled as a metal ball resting at a specific height in one-dimensional space. The host's initial trust M_h is its *mass* while the height of a ball represents $\texttt{TrustRed}(h)$. The model considers a similarity graph $\mathcal{G}'(H, C)$ in which the edge set $C \subseteq F_r$ may have a connection for each flow in F_r. The edge set C is defined as

$$C = \{(h_1, h_2) | (h_1, h_2) \in F_r \wedge s_{h_1,h_2} > 0\} \tag{16}$$

Namely, an edge exists if there is a flow between the two hosts and the two hosts have a non-zero similarity as defined in Equation 1. Note that similarity is over $\mathcal{G}'(H, C)$ while routing considered $\mathcal{G}(D, A)$. In the physics analogy, each edge in C is an *elastic string whose tautness captures the similarity*. Indeed, highly similar nodes that are connected are more likely to see their trust move in unison if one is affected by a security event whereas highly dissimilar nodes would not affect each other if one was compromised. Tautness is modeled as the maximal length that the elastic string can have. At the onset, there are no trust reductions anywhere, i.e., all balls lie on a flat surface at height 0.

Without loss of generality, assume a single security event $e(h)$. The event $e(h)$ causes $\texttt{TrustRed}(h) > 0$ by picking up host h to a prescribed height equal to the trust reduction directly incurred. That is, *height* $= \texttt{TrustRed}(h)$. Raising host h to some height impacts connected neighbors. Those hosts are lifted (a trust reduction) because of their respective similarities to h. The behaviors of the elastic strings impact the magnitude of the collateral lifts. In the physics analogy, connected devices are lifted as a function of their mass and connections to neighbors. Massive hosts (highly trusted) are not easily lifted leading to a dissipation of the upward force through the stretching of the elastic links as well as tensile forces. There are multiple ways to model similarity:

- alter the initial length of strings,
- characterize the elasticity constant behind each string, or
- bound the maximal extension length of a string.

which have subtle computational implications. Each approach leads to quantitatively different yet qualitatively similar results. We chose to encode similarity as the maximum extension length of a string. This proved most suitable for linear optimization.

Theoretical trust models [9] exist to address how trust can propagate [31]. Here, we focus on how *distrust* propagates, i.e., direct and indirect trust loss. Distrust propagation is different and is usually application dependent. Guha et al. [15] review the process for developing a trust model for an application. The elastic string model, which is novel, elegantly captures both direct and indirect (i.e., transitive) trust loss and is intuitive. It reflects the nature of modern attackers for two reasons: (1) it chains together exploits across the network and build up capabilities, (2) it capture pivoting abilities through similarities between nodes.

Parameters

H – set of nodes in the trust network. Same as set of hosts in physical network.
M_h – mass of node $h \in H$.
C – similarity connections. Each connection $(h_1, h_2) \in C$ connects h_1 to h_2. See Eq 16.
$C(h)$ – set of connections touching host $h \in H$.
L_c – maximum length of string for connection $c \in C$.
K_c – elastic constant for connection $c \in C$.
$\mathcal{E} \subseteq H$ - set of hosts experiencing trust loss due to potential network compromise.
$\mathcal{E}(h) \subset \mathbb{R}^+$ - trust reduction experienced by host $h \in \mathcal{E}$.
F^+ – the set of required flows to be delivered based on the routing policy.
I – the set of incidental flows induced by the network configuration.
$F^+(h_1, h_2) \subseteq F^+$ – the set of flows in F^+ involving h_1 and h_2.
$I(h_1, h_2) \subseteq I$ – the set of flows in I involving h_1 and h_2.

Decision Variables

$tl_h \in \mathbb{R}_{\geq 0}$ – trust loss for host $h \in H$.
$f_{c,h} \in \mathbb{R}$ – elastic force experienced by host $h \in H$ due to connection $c \in C$.
$t_{c,h} \in \mathbb{R}$ – tensile force experienced by host $h \in H$ due to connection $c \in C$.
$b_h \in \mathbb{R}_{\geq 0}$ – supporting force from ground experienced by host $h \in H$.
$l_h \in \mathbb{R}_{\geq 0}$ – lifting force experienced by host $h \in H$.
$s_{h_1,h_2} \in \mathbb{R}_{\geq 0}$ – flow similarity between hosts $h_1, h_2 \in H$.
$w_c \in \mathbb{R}_{\geq 0}$ – effective maximum length of connection $c \in C$.
$z_f \in \{0, 1\}$ – binary variable to install rule to block an incidental flow $f \in I$.

Constraints

The first set of constraints deal with network events which compromise the trust of hosts, causing them to be lifted

$$\forall h \in \mathcal{E} \ : \ tl_h \geq \mathcal{E}(h) \ , \qquad \forall h \in H \setminus \mathcal{E} \ : l_h = 0 \qquad (17, 18)$$

Equation 17 lower bounds the trust loss for the hosts involved in the event to be at least the loss specified in the event. These hosts will then experience a lifting force necessary to stay at height tl_h and be in equilibrium described by Equation 29. Equation 18 states that hosts that are not involved in the event do not experience a lifting force.

$$\forall h \in H \ : b_h \leq M_h \ , \qquad b_h > 0 \iff tl_h = 0 \qquad (19, 20)$$

Equation 19 bounds the supporting force exerted by the ground on a host to be no greater than the weight of the host. Equation 20 dictates that the supporting force on a host is present if and only if the host is resting on the ground (did not incur a trust loss).

$$\forall (h_1, h_2) \in C \ : w_{(h_1,h_2)} = L_{(h_1,h_2)} - s_{h_1,h_2} \tag{21}$$

$$\forall (h_1, h_2) \in C \ : s_{h_1,h_2} = |F^+(h_1, h_2)| \ + \sum_{f \in I(h_1,h_2)} (1 - z_f) \tag{22}$$

Equation 21 gives the effective length of a string to be used in the force calculations. Higher similarity between two hosts entails a shorter maximal distance between them. Equation 22 calculates the flow similarity between two hosts. By design, $s_{h_1,h_2} \leq L_c$ for all $(h_1, h_2) \in C$.

$$\forall c \in C \ : f_{c,c_1} = K_c \cdot (tl_{c_2} - tl_{c_1}) \ , \qquad f_{c,c_2} = K_c \cdot (tl_{c_1} - tl_{c_2}) \qquad (23,\ 24)$$

$$\forall c \in C \ : |tl_{c_1} - tl_{c_2}| \leq w_c \ , \qquad t_{c,c_1} + t_{c,c_2} = 0 \qquad (25,\ 26)$$

Equations 23 and 24 determine the elastic forces a host experiences due to connections in which it is involved. This calculation is based directly on Hooke's Law. Equation 25 ensures that the distance between the hosts does not exceed the maximum length of the connection. Equation 26 states that the tensile forces experienced by the hosts sum to zero, i.e., the forces are equal in magnitude and opposite in direction.

$$\forall c \in C \ : t_{c,c_1} < 0 \iff tl_{c_1} > tl_{c_2} \ , \quad t_{c,c_1} \neq 0 \iff w_c - |tl_{c_1} - tl_{c_2}| = 0 \qquad (27,\ 28)$$

Equation 27 states that a host can only experience negative tension from a connection if it is higher than the other host in the connection. Furthermore, Equation 28 prevents a host from experiencing any tension from a connection if the string is not at its maximum length. The last set of constraints require the solution to be an equilibrium state for the string network.

$$\forall h \in H \ : \left(\sum_{c \in C(h)} f_{c,h} + t_{c,h} \right) + b_h + l_h - M_h = 0 \tag{29}$$

Equation 29 stipulates that the upward and downward forces experienced by each host must balance. This integrates lifting, ground, tensile, gravitational, and connection forces. To summarize, equations 17-29 are based on laws of classical mechanics and the desire to capture indirect impacts of events described at the beginning of this subsection.

Trust Objective

The trust model minimizes a linear combination of the heights of the hosts and the number of needed additional firewall rules. This mitigates trust reduction across network without adding excessive complexity to the current routing configuration. This objective function is expressed below (β_0, β_1, and γ_f are parameters).

$$\min \beta_0 \sum_{h \in H} tl_h + \beta_1 \sum_{f \in I} \gamma_f \cdot z_f \tag{30}$$

Solving the Trust Model

A direct resolution of the trust model cannot complete to optimality in the small amount of time (minutes) one would have to react to a security incident.

With LNS [30] as a meta-strategy, one can achieve high-quality solutions in seconds to minutes at worst. The incomplete search can then proceed as shown in Algorithm 1 with t being a parameter denoting the chunk size for the LNS.

Algorithm 1 Solve_TrustModel(M over $G(V, E), e(h) \in \mathcal{E}, t$).

```
$S^* = \text{solve}_{LNS}\ MIP(M(e(h)) \cup \{z_f = 0 | f \in F_r\})$
while $more\ time$ do
    $Pick\ X$ such that $|X| = t$ and $X$ is a random subset of $I$
    $S = \text{solve}_{LNS}\ MIP(M(e(h)) \cup \{z_f = S^*(z_f) | f \in F_r \setminus X\})$
    if $S < S^* \wedge \text{feasible}(S)$ then
        $S^* = S$
    end if
end while
```

The algorithm starts by fixing *all* z variables to 0, i.e., it reports on the trust reduction one would experience when taking no actions whatsoever. $M(e(h))$ denotes the MIP model for the considered event. Then it proceeds in waves of LNS searches, each focusing on a random subset X (of fixed cardinality t) of edges that coincide with incidental flows. Each LNS searches fixes all other edges ($E \setminus X$) to their values in the incumbent solution ($S^*(z_f)$) and optimizes over the z_f for all $f \in X$. Any feasible improving solution S in adopted as the new incumbent. For our testing, we set t to be 50.

3.3 Integrated Approach

It is natural to consider an integrated approach with a single model that encompasses both routing and security requirements. Such an model can be produced with a few changes. First, the routing model can no longer limit its attention to the flows in $F = F^+ \cup F^-$. Indeed, the trust part of the model must decide whether to block incidental flows whose existence is implied by the value chosen for the routing variables. Thus, one must consider F as the entire universe of flows, namely $F = \mathcal{F}$. An immediate consequence is a significantly larger set of decision variables for routing. Second, incidental flows are readily encoded in the a^f_τ variables. Third, the trust model must restrict its decisions to z_f variables for flows in $\mathcal{F} \setminus (F^+ \cup F^-)$ since required flows cannot be blocked (and restricted flows are already blocked). Finally, the objective function can be a lexicographic ordering of the objectives from routing and trust (i.e., a weighted sum).

We implemented such a model and the resulting approach was not tractable. On the introductory toy example, it produces the same solution as the 2-stage approach. Looking ahead we evaluate our two stage model on pod-4 and pod-6 fat tree topologies found in data center networks (see Section 5). On the smallest pod-4 instances of the empirical evaluation, the integrated model had a duality gap in excess of 5% after 8 hours of computation time. The last solution offered the same routing as the 2-stage model, while the loss of trust was twice as big (the value of the trust objective that is being minimized). Worse, the model size alone is prohibitive, causing the solver pre-processing phase to take more than 10 minutes. The Achilles' heel of the approach is the quadratic size of $\mathcal{F}$. Finally, note that an LNS approach here requires an incumbent solution that the integrated model does not find early in the execution (the first solution on pod-4 is found after one hour of runtime).

Another tempting avenue is column-generation [11]. The on-demand creation of routes with the addition of an indicator variable to choose such a route seems direct. Yet, every route must create several variables for the incidental flows introduced by itself *but also because of the interplay with other selected routes.* A column generation must therefore produce a bundle of several columns (for the route and the incidental flows it creates by itself) as well as examine all the existing routes to add another set of columns for the incidental flows that arise from mixing routes. While this is doable and possibly competitive, we believe the complexity of the implementation would exceed the two-stage approach.

4 Assessment

Optimization Assessment. To assess our optimization model we report on the following standard metrics: 1) solve time, 2) optimality gap, 3) model size.

Functional Assessment Our motivating application for DuelMIPs is in data center networks where links are uniform and high bandwidth [3]. The following quality metrics are used:

Normalized Path Length. For each f_{h_1,h_2} the path length divided by the shortest network path between h_1 and h_2.

Normalized Number of Rules. The total number of rules written to SDN devices throughout the network is normalized by the number of required rules.[1] The desire for compact rulesets is driven by two factors: 1) SDN devices are limited in their TCAM memory and 2) human interpretability.

SDN Rules. The total number of rules assigned to the SDN devices.

Wildcard Rules. The number of rules that are not fully specified. Wildcard rules are more complex than fully specified rules and thus should be given additional weight when judging the complexity of a configuration.

Usability Assessment. The literature on usability of network rules [4, 5, 33, 38, 39] considers the following factors to be important: structural complexity of the ruleset, size of the ruleset, misconfigurations, conflicts and the presence of comments to aid in the intention of the rules.

All rules have the form of $(\texttt{src}, \texttt{dest}, \texttt{protocol}) \rightarrow \textit{action}$ in which *action* can be `drop` or `send(port)`. Therefore, the structural complexity of each rule largely depends on the number of wildcards. Recall that wildcards can reduce the number of rules needed to specify a routing policy, yet multiple occurrences on the same device can induce conflicts that weaken clarity. The number of rules and wildcard rules is reported in Functional Assessment.

We focus on the number of *conflicts* in the ruleset, using the classification due to Al-Shaer and Hamed [5]. Before describing conflict types recall that SDN devices match the most specific rule, if two rules are the same specificity, the first listed rule is applied. Conflicts indicate the complexity for human understanding.

Shadowing. A generic rule R_B appears before a specific rule R_A, hiding R_A.

Generalization. A generic rule R_A appears after a specific rule R_B.

Correlation. Two rules match some packets in common.

Redundancy. Rule R_A generalizes R_B and both feature the same action.

Irrelevance. Rule R_A matches no packets or all packets are handled by earlier rules.

Trust Assessment. To assess the Trust Model, we consider two versions. The first we call Initial, denoted as $\aleph$, where no extraneous flows are blocked. This corresponds to a model in which all z_f are fixed to 0. The second version is the actual solution, denoted as F. The event node in each scenario is lifted to height 8 (see Section 5). We split our evaluation into two components: 1) directly examining the solution quality and 2) evaluating the solution with respect to stopping potential attacks. Before these metrics, we introduce required notation.

Notation. Denote heights buckets of $[0, 2], (2, 4], (4, 6], (6, 8]$, as vLow, Low, Med, High respectively. Consider the multigraph $G_{\mathsf{m}}(H, E)$, where edges in E represent a permitted flow between two end hosts in H. Define $G_{\mathsf{m},F}$ as G_{m} when $I = \emptyset$.

[1] Let $\texttt{RequiredRules}(f) = 1$ if $f \in F^-$ and $\texttt{RequiredRules}(f) = \texttt{ShortestPath}(f) - 1$ otherwise. Then we normalize the number of rules by $\sum_{f \in F} \texttt{RequiredRules}(f)$.

Table 2 Network Roles. For intrapod flows, each server in a pod communicates with a chosen *manager* server within the pod via `HTTP`. Communicate pattern is created randomly with the required type and number of flows.

Pod #	Role	Each server communicates with
1	Web server	2-3 application servers via HTTP **NO** HTTP to database or authentication
2	Application Server	1-2 content servers in each pod via RPC protocol 1-2 databases via SQL protocol 1-2 authentication servers via LDAP
3, 4	Content Server	Application
5	Database Server	Application
6	Authentication Server	Application

Node Heights Δ. The difference between configurations $\aleph$ and F of $|\mathsf{vLow}|, |\mathsf{Low}|, |\mathsf{Med}|, |\mathsf{High}|$.

Potential Attacks – Attack Paths and Min Cut. Let $G_{\mathsf{m},T}$ be the subgraph of G derived from DUELMIPS$'s$ solution and let n denote the number of edges in I removed. Let $G_{\mathsf{m},R}$ be a subgraph of G where n random edges in I are removed. We consider two metrics:

1. How many attack paths are removed by DUELMIPS's defensive actions. Consider all paths up to some maximum length in G_m which i) contain at least one edge from I and ii) that connect a web server (an attack source a) to a database or authentication server (attack target t), denoted as $P_{a,t}(G_\mathsf{m})$. The number of removed attack paths is measured by $\mathsf{Gsize}(G_{\mathsf{m},T}) := P_{a,t}(G_{\mathsf{m},T})/P_{a,t}(G_\mathsf{m})$ (defined analogously for $G_{\mathsf{m},R}$).
2. The variety of attacks available to the attacker. Measured by the relative min-cut

$$\mathsf{resMin}(G_{\mathsf{m},T}, a, t) := \frac{\mathsf{minCut}(G_{\mathsf{m},T}, a, t) - \mathsf{minCut}(G_{\mathsf{m},F}, a, t)}{\mathsf{minCut}(G_\mathsf{m}, a, t) - \mathsf{minCut}(G_{\mathsf{m},F}, a, t)}$$

with $\mathsf{resMin}(G_{\mathsf{m},R,a,t})$ defined analogously. Define $\mathsf{resMin}(G_{\mathsf{m},T})$ as the expected value of $\mathsf{resMin}(G_{\mathsf{m},T}, a, t)$ over all webservers a and database or authentication servers t with $\mathsf{resMin}(G_{\mathsf{m},R})$ defined analogously.

5 Experimental Setup

We created synthetic instances that model traffic that would be observed in a moderately sized data center. The instances utilize the Fat-Tree [3] topology commonly seen in data centers due to its large aggregate bandwidth, high fault tolerance, and robust scalability. We use six pods, allowing for 54 endpoint devices to be present with 46 SDN devices. We consider these endpoint devices to be servers in our experiments.

Network Topology, F^+, and F^-. We assign roles to the servers in each of the six pods. The required and restricted communication patterns are based around these roles, this is found in Table 2. All evaluation considers three variants: no wildcards, at most one wildcard per rule, and at most two wildcards. No incidental flows are possible when no wildcards are used, i.e. $I = \emptyset$. The parameters in the objective are set as $\alpha_0 = 10$ and $\alpha_1 = 1$.

The routing model evaluation is based on 15 instances. Callbacks were used to terminate an optimization if the optimality gap was within 5% and the runtime had exceeded 3 hours.

Trust Inputs. Each network device was assigned a mass based on role. Maximum mass for web, application, content, database, and authentication devices is $1, 2, 3, 3, 4$ respectively. Mass is drawn uniformly from the interval $[\frac{m}{2}, ..., m]$. All event nodes are lifted to a height

Table 3 Optimization and functional assessments for routing model solutions when varying the number of wildcards in the rule header. The 15 instances were each evaluated with all three settings.

	0 wildcards			$\leq$ 1 wildcards			$\leq$ 2 wildcards		
Optimization	μ	min	max	μ	min	max	μ	min	max
Solve Time (s)	51	48	54	8.1K	910	12.6K	11.3K	10.8K	13.2K
Optimality Gap (%)	0	0	0	0	0	0	2	2	3
Millions of Variables	6.08	6.08	6.09	7.65	7.65	7.66	7.71	7.70	7.71
Millions of Constraints	1.75	1.73	1.76	2.33	2.30	2.35	2.90	2.87	2.93
Functional	μ	min	max	μ	min	max	μ	min	max
Norm Path Length	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.3
Norm # of Rules	1.0	1.0	1.0	.47	.46	.49	.51	.41	.60
# of Wildcard Rules	0	0	0	606	590	627	577	460	800
Total Rules	1340	1310	1360	627	612	647	677	546	803

of 8 (maximum height of 10). A height of 10 would reflect a node certainly compromised and which should be islanded outright. In all instances, the elasticity constants for strings were set to 1 to simplify the analysis of the outputs. The maximum length for any connection was set to 8 to allow nodes to rest on the ground if appropriate defensive actions are taken.

Trust Objective Parameters. In the trust model objective, we set $\beta_0 = 10$, the penalty associated to the trust loss (height) of a node, $\beta_1 = 1$ for simplicity, and each firewall rule added to an SDN device induced a penalty cost (γ_f) based on the protocol being blocked. Specifically, for `HTTP` and `RPC` firewalls, the penalty was set to 1. For `SQL` traffic, the penalty was 2. Finally, for `LDAP`, the penalty was 3. Adding firewalls that filter packets using critical protocols would be more likely to impact overall function.

Trust Optimization. The generated network configurations were given to the trust model for analysis. Only the solutions from the one wildcard and two wildcard methodologies were considered. That is, $I = \emptyset$ if no wildcards are used. The LNS strategy presented in Algorithm 1 was used with a time limit of five minutes.

6 Results

All evaluations were conducted on a Linux machine equipped with an Intel Xeon CPU E5-2620 2.00 GHz and 64GB of RAM. The MIP solver used was Gurobi Optimizer 9.1.

Routing. Here we ask if DuelMIPs's routing stage: 1) produces solutions are output quickly, 2) generates good configurations, and 3) produces rulesets that are lighter or less complex than fully-specific rulesets. The routing model assessment is in Table 3.

Increasing wildcard usage significantly affects the model size and solving time. Yet, even with up to 2 wildcards, this remains acceptable for use in network planning.

With no wildcards, all rules used the shortest path, and all restricted flows were blocked at the first available SDN device. When wildcards are allowed, the number of rules decreases. Interestingly, adopting 2 wildcards per rule does not deliver any further rule set compression In some instances, we even observed an increase in the total number of rules when compared to 1 wildcard. The ruleset size increases were due to Equation 15 (fully specified rules had to be added alongside some 2 wildcard rules to prevent its violation), but were outweighed by the improvements to the usability of the rulesets. Indeed, recall that ruleset size is not the sole indicator of usability. A larger ruleset comprised of simple-to-understand two wildcard rules could be considered more usable than a smaller ruleset that contains mainly one wildcard rules.

A Python implementation of the state machine from [5] is used to detect conflicts. DuelMIPs includes the constraint that each required flow matches at most one wildcard rule. Two wildcard rules are correlated due to Equation 15. Thus, DuelMIPs can output non-wildcard rules first, followed by wildcard rules without any risk of correlations. DuelMIPs may exhibit correlation and generalization, it should never exhibit Irrelevance, Shadowing, or Redundancy. This was confirmed in all experiments. The 1 wildcard instances displayed an average of 21 generalizations and 10 correlations and 2 wildcard instances displayed an average of 23 generalizations and 26 correlations. This slight increase is due to more general rules that require (potentially multiple) specific rules to handle exceptions.

Trust. The Trust model takes a routing solution, a set of incidental flows, and an event node referred to as a potentially compromised host. It must produce recommendations for reducing collateral damage. The Trust Model runs in at most five minutes which is appropriate. In many instances, the LNS uses several hundred iterations to get those results. Table 4 shows results, averaging over the 54 potential security events, for each of the 15 routing instances that use up to 1 or 2 wildcards, respectively.

The optimizer drastically affects the height of nodes. For the routing solutions using at most 1 wildcard, we see most hosts shift from High to Medium or Low to vLow. The result is more dramatic with 2 wildcards with hosts shifting from High to vLow. This is achieved by blocking only 1% of incidental flows in the 1 wildcard case, and, on average, 3% of the incidental flows in the 2 wildcard case. No required flows are blocked.

Potential Attacks The trust model recommendations reduce an adversary's ability to pivot in the network. In Table 5, we consider the trust model's output when a web server is the event node. In both wildcard settings, the incidental flows that DuelMIPs decides to firewall meaningfully reduce an attacker's potential attack paths (see Section 4 for the metric definition). Here the maximum path length is five nodes, which accounts for up to 5 pivots (most attacks use only a handful [7]) The two wildcard setting allows an attacker more freedom to pivot in the network. It is important that the trust model's recommendations improve the security metrics more. This is confirmed. In either setting, DuelMIPs's choices are superior to removing random edges. This indicates that the trust model appropriately considers the network structure when choosing which flows to block.

7 Conclusion

This work presents DuelMIPs, a two stage optimization approach. The first model, which focuses on routing, is an IP that creates SDN rules for a moderate size data center network in at most a few hours. The Routing Model outputs are functional and usable. By using wildcards intelligently, the routing model is able to compress rule sets at the cost of allowing some extraneous flows to be routed through the network.

The second model is a MIP that focuses on Trust. It is run in response to an identified compromise and considers residual trust loss due to the extraneous flows created by the Routing Model. It quickly creates recommendations that prevent an attacker from spreading to other similar nodes. To retain tractability it utilizes LNS.

This work focused on the complex task of balancing SDN rulesets for functionality, usability, and trust. The Routing and Trust models are coupled through the set of extraneous flows. As mentioned in Section 3.2, other notions of similarity can be used to model other

Table 4 The average number of hosts within each trust reduction classification for each 1 and 2 wildcard routing configuration given to the trust model. Note that each cell is averaged over all 54 possible singleton trust reduction events. For each level in $\mathsf{Level}_{\triangle} = \mathsf{Level}_{F} - \mathsf{Level}_{\aleph}$.

	1 wildcard				2 wildcard			
Instance	$\mathsf{High}_{\triangle}$	$\mathsf{Med}_{\triangle}$	$\mathsf{Low}_{\triangle}$	$\mathsf{vLow}_{\triangle}$	$\mathsf{High}_{\triangle}$	$\mathsf{Med}_{\triangle}$	$\mathsf{Low}_{\triangle}$	$\mathsf{vLow}_{\triangle}$
1	-8	6	-1	3	-14	-21	-2	37
2	-8	7	-5	5	-23	-17	9	30
3	-8	7	-3	4	-44	5	9	30
4	-8	7	-2	3	-20	-19	0	38
5	-8	7	-2	3	-27	-1	-3	31
6	-7	6	-1	1	-19	-17	0	36
7	-8	7	-5	5	-27	-13	3	36
8	-9	9	-15	15	-13	-20	-3	35
9	-7	7	-3	3	-36	21	9	6
10	-7	7	-7	7	-14	-23	-1	38
11	-8	8	-6	6	-11	-25	3	33
12	-8	9	-3	2	-23	-10	-3	30
13	-8	6	-1	3	-28	-12	5	35
14	-8	7	-1	2	-20	-24	10	35
15	-8	7	-4	5	-28	-3	-3	34

Table 5 Improvement of subgraph size and residual min-cut for DUELMIPS and random recommendations. Attack source is web server and target is database or authentication server.

	≤ 1 wildcards				≤ 2 wildcards			
Metric	μ	σ^2	min	max	μ	σ^2	min	max
$\mathsf{Gsize}(G_{\mathsf{m},T})$	0.95	< 0.01	0.84	1.0	0.79	0.01	0.56	1.0
$\mathsf{Gsize}(G_{\mathsf{m},R})$	0.97	< 0.01	0.94	0.99	0.97	0.01	0.88	1.0
$\mathsf{resMin}(G_{\mathsf{m},T})$	0.95	< 0.01	0.86	1.0	0.31	0.02	0.15	0.69
$\mathsf{resMin}(G_{\mathsf{m},R})$	0.99	< 0.01	0.93	1.0	0.95	< 0.01	0.91	0.99

network security best practices. An important piece of future work is understanding the impact of multiple event nodes on the Trust Model solve time. We see two potential applications: 1) supporting a response to an attacker that has compromised multiple nodes and 2) allowing the Trust Model to be useful in the Planning scenario as well. One could try to preemptively deploy firewall rules that are likely to prevent collateral damage over a variety of scenarios [13].

References

1 Alfarez Abdul-Rahman and Stephen Hailes. A distributed trust model. In *Proceedings of the 1997 workshop on new security paradigms*, pages 48–60, 1998.

2 Sugam Agarwal, Murali Kodialam, and TV Lakshman. Traffic engineering in software defined networks. In *2013 Proceedings IEEE INFOCOM*, pages 2211–2219. IEEE, 2013.

3 Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM. doi:10.1145/1402958.1402967.

4 Saeed Al-Haj and Ehab Al-Shaer. Measuring firewall security. In *2011 4th Symposium on Configuration Analytics and Automation (SAFECONFIG)*, pages 1–4. IEEE, 2011.

5 Ehab S Al-Shaer and Hazem H Hamed. Firewall policy advisor for anomaly discovery and rule editing. In *International Symposium on Integrated Network Management*, pages 17–30. Springer, 2003.

6 Rashid Amin, Nadir Shah, Babar Shah, and Omar Alfandi. Auto-configuration of acl policy in case of topology change in hybrid sdn. *IEEE Access*, 4:9437–9450, 2016.

7 Giovanni Apruzzese, Fabio Pierazzi, Michele Colajanni, and Mirco Marchetti. Detection and Threat Prioritization of Pivoting Attacks in Large Networks. *IEEE Transactions on Emerging Topics in Computing*, 8(2):404–415, April 2020. doi:10.1109/TETC.2017.2764885.

8 Randall J Boyle and Raymond R Panko. *Corporate computer security.* Pearson, 2015.

9 Jin-Hee Cho, Kevin Chan, and Sibel Adali. A survey on trust modeling. *ACM Computing Surveys (CSUR)*, 48(2):1–40, 2015.

10 Timothy Curry, Devon Callahan, Benjamin Fuller, and Laurent Michel. DOCSDN: Dynamic and optimal configuration of software-defined networks. In *Australasian Conference on Information Security and Privacy*, pages 456–474. Springer, 2019.

11 George B. Dantzig and Philip Wolfe. Decomposition principle for linear programs. *Oper. Res.*, 8(1):101–111, February 1960. doi:10.1287/opre.8.1.101.

12 Rup Kumar Deka, Kausthav Pratim Kalita, Dhruba K Bhattacharya, and Jugal K Kalita. Network defense: Approaches, methods and techniques. *Journal of Network and Computer Applications*, 57:71–84, 2015.

13 Ron S Dembo. Scenario optimization. *Annals of Operations Research*, 30(1):63–80, 1991.

14 Diego Gambetta et al. Can we trust trust. *Trust: Making and breaking cooperative relations*, 13:213–237, 2000.

15 Ramanthan Guha, Ravi Kumar, Prabhakar Raghavan, and Andrew Tomkins. Propagation of trust and distrust. In *Proceedings of the 13th international conference on World Wide Web*, pages 403–412, 2004.

16 Guibing Guo, Jie Zhang, and Neil Yorke-Smith. Leveraging multiviews of trust and similarity to enhance clustering-based recommender systems. *Knowledge-Based Systems*, 74:14–27, 2015.

17 Jun He and Wei Song. Achieving near-optimal traffic engineering in hybrid software defined networks. In *2015 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2015.

18 Lance J Hoffman, Kim Lawson-Jenkins, and Jeremy Blum. Trust beyond security: an expanded trust model. *Communications of the ACM*, 49(7):94–101, 2006.

19 MHR H.R. Khouzani, Zhengliang Liu, and Pasquale Malacaria. Scalable min-max multi-objective cyber-security optimisation over probabilistic attack graphs. *European Journal of Operational Research*, 278(3):894–903, 2019. doi:10.1016/j.ejor.2019.04.035.

20 Ansam Khraisat, Iqbal Gondal, Peter Vamplew, and Joarder Kamruzzaman. Survey of intrusion detection systems: techniques, datasets and challenges. *Cybersecurity*, 2(1):1–22, 2019.

21 Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, 2013.

22 Man Yue Mo. One day short of a full chain: Real world exploit chains explained, March 2021. URL: https://github.blog/2021-03-24-real-world-exploit-chains-explained/.

23 John Moy. Rfc2328: Ospf version 2, 1998.

24 Xuan-Nam Nguyen, Damien Saucez, Chadi Barakat, and Thierry Turletti. Officer: A general optimization framework for openflow rule allocation and endpoint policy enforcement. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 478–486. IEEE, 2015.

25 Ahmed Patel, Qais Qassim, and Christopher Wills. A survey of intrusion detection and prevention systems. *Information Management & Computer Security*, 2010.

26 Jon Postel. Internet protocol—darpa internet program protocol specification, rfc 791, 1981.

27 Yakov Rekhter, Tony Li, Susan Hares, et al. A border gateway protocol 4 (bgp-4), 1994.

28 Myriana Rifai, Nicolas Huin, Christelle Caillouet, Frédéric Giroire, D Lopez-Pacheco, Joanna Moulierac, and Guillaume Urvoy-Keller. Too many sdn rules? compress them with minnie. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2015.

29 Karen Scarfone, Wayne Jansen, Miles Tracy, et al. Guide to general server security. *NIST Special Publication*, 800(123), 2008.

30 P. Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In *Proceedings of Fourth International Conference on the Principles and Practice of Constraint Programming (CP'98)*, pages 417–431. Springer Verlag, October 1998.

31 Yan Lindsay Sun, Wei Yu, Zhu Han, and KJ Ray Liu. Information theoretic framework of trust modeling and evaluation for ad hoc networks. *IEEE Journal on Selected Areas in Communications*, 24(2):305–317, 2006.

32 Stefano Vissicchio, Laurent Vanbever, Luca Cittadini, Geoffrey G Xie, and Olivier Bonaventure. Safe update of hybrid sdn networks. *IEEE/ACM Transactions on Networking*, 25(3):1649–1662, 2017.

33 Artem Voronkov, Leonardo A Martucci, and Stefan Lindskog. Measuring the usability of firewall rule sets. *IEEE Access*, 8:27106–27121, 2020.

34 John Wack, Ken Cutler, and Jamie Pole. Guidelines on firewalls and firewall policy. Technical report, BOOZ-ALLEN AND HAMILTON INC MCLEAN VA, 2002.

35 Lei Wang, Qing Li, Yong Jiang, and Jianping Wu. Towards mitigating link flooding attack via incremental sdn deployment. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 397–402. IEEE, 2016.

36 Wen Wang, Wenbo He, and Jinshu Su. Enhancing the effectiveness of traffic engineering in hybrid sdn. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2017.

37 Yonghong Wang and Munindar P Singh. Formal trust model for multiagent systems. In *IJCAI*, volume 7, pages 1551–1556, 2007.

38 Tina Wong. On the usability of firewall configuration. In *Symposium on usable privacy and security*, 2008.

39 Avishai Wool. A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67, 2004.

40 Cai-Nicolas Ziegler and Georg Lausen. Analyzing correlation between trust and user similarity in online communities. In *International Conference on Trust Management*, pages 251–265. Springer, 2004.

A Framework for Generating Informative Benchmark Instances

Nguyen Dang
School of Computer Science, University of St Andrews, UK

Özgür Akgün
School of Computer Science, University of St Andrews, UK

Joan Espasa
School of Computer Science, University of St Andrews, UK

Ian Miguel
School of Computer Science, University of St Andrews, UK

Peter Nightingale
Department of Computer Science, University of York, UK

Abstract

Benchmarking is an important tool for assessing the relative performance of alternative solving approaches. However, the utility of benchmarking is limited by the quantity and quality of the available problem instances. Modern constraint programming languages typically allow the specification of a class-level model that is parameterised over instance data. This separation presents an opportunity for automated approaches to generate instance data that define instances that are *graded* (solvable at a certain difficulty level for a solver) or can *discriminate* between two solving approaches. In this paper, we introduce a framework that combines these two properties to generate a large number of benchmark instances, purposely generated for effective and informative benchmarking. We use five problems that were used in the MiniZinc competition to demonstrate the usage of our framework. In addition to producing a ranking among solvers, our framework gives a broader understanding of the behaviour of each solver for the whole instance space; for example by finding subsets of instances where the solver performance significantly varies from its average performance.

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

Keywords and phrases Instance generation, Benchmarking, Constraint Programming

Digital Object Identifier 10.4230/LIPIcs.CP.2022.18

Supplementary Material *Software (Source Code)*: `https://github.com/stacs-cp/AutoIG`

Funding This work uses the Cirrus UK National Tier-2 HPC Service at EPCC (`http://www.cirrus.ac.uk`) funded by the University of Edinburgh and EPSRC (EP/P020267/1).
Nguyen Dang: is a Leverhulme Early Career Fellow.
Ian Miguel: supported by EPSRC EP/V027182/1.

1 Introduction

A practitioner faced with solving a new problem has a difficult choice among many solving algorithms, whose performance on the new problem is unknown and is likely to be variable. One approach is to draw instances from the problem to *benchmark* the various solvers under consideration, i.e. an empirical study of relative performance. This approach is favoured for computationally challenging tasks since the performance behaviour of a non-trivial algorithm is difficult to predict and is unlikely to be susceptible to a purely theoretical analysis [10]. As Beiranvand et al. [11] argue, care must be taken to select an instance set with a variety of difficulty for benchmarking in order to obtain the best insight into solver performance.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 18; pp. 18:1–18:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

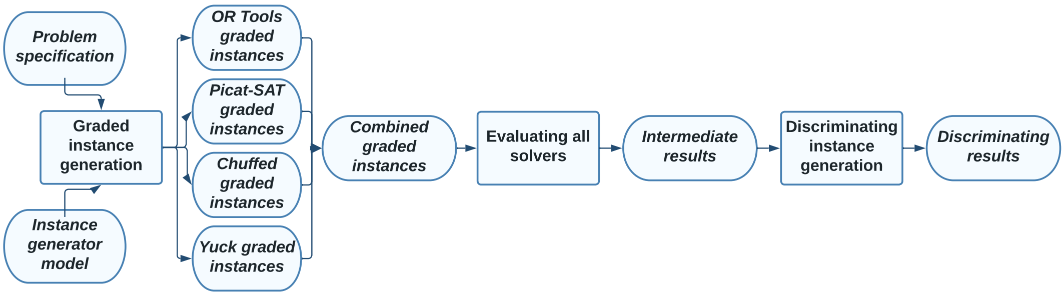


Figure 1 Flowchart of the whole AutoIG application process.

Constraint programming (CP) approaches particularly benefit from empirical analysis, since modern tool chains like MiniZinc [34] and savilerow [35] support targeting multiple solvers from a solver-independent constraint model. These may be entirely different paradigms, such as SAT [13], SMT [9] or indeed CP, and so can vary in performance significantly.

The need for empirical benchmarking is further supported by competitions run by several research communities, like the MiniZinc challenge [44] in the CP community, the SAT competition [18] and the AI planning competition [47]. Solver developers enter a competition by providing a default configuration of their solver. Each solver supports a common interface for specifying their input and output. The competition is then run on a set of problem instances and the solvers are ranked with respect to their comparative performance.

In the main solver competition for CP, the MiniZinc challenge, each solver is given two inputs: a solver-independent problem-level model and instance data written in a separate data file. Then MiniZinc is used to instantiate and translate the solver-independent model into input suitable for each solver. The main result of the challenge is a ranking of solvers. More detailed results pertaining to the ranking of solvers per problem class are also published.

The selection of problem instances to be used in a competition is extremely important to avoid conclusions that are unintentionally biased towards the chosen instances. Competitions somewhat mitigate this problem by inviting solver authors to submit benchmark instances. This is a promising sociotechnical attempt at alleviating the problem of bias, but it is laborious and does not provide a comprehensive solution.

Benchmarking is not only useful for finding an overall ranking among options, but also for finding subsets of instances where the performance of a solver is significantly different from the performance of the same solver overall. For example, solver A might perform better for most instances of a problem class in comparison with B, yet perform very poorly for a particular subset of the instances. Information like this can be extremely valuable to solver developers. A traditional competition that works by running all solvers on a fixed set of instances can occasionally detect such cases even though it does not actively look for them.

For an informative benchmark we need a sufficient *quantity* of high *quality* instances and the ability to dynamically explore subsets of the instance space to detect performance discrimination. In this work we present AutoIG, a constraint-based instance generation framework, that supports automatically generating *graded* instances (i.e., solvable at a certain difficulty level for a given solver), and finding discriminating instances (i.e. easy for one solver and difficult for another solver). In combination, these two methods can be used to generate a large number of high-quality instances. Furthermore, they can be used to find interesting subsets of the instance space as opposed to leaving their discovery to chance.

Figure 1 gives a flowchart for an end-to-end application of AutoIG, whose instance generation process is explained in Section 3. Without loss of generality, the flowchart lists the four solvers used for the evaluation of AutoIG in this paper. Section 4 explains the choice of these solvers and the five problem classes we use. Both stages of AutoIG can be applied to other solvers and solver configurations. The AutoIG process has two main inputs: a problem specification (in the form of a MiniZinc model in this paper) and a problem specific instance generator. The instance generator is parameterised to allow AutoIG to generate a variety of instances. There are two main places where we can extract results from AutoIG, evaluating all solvers on the combined set of graded instances (marked intermediate results in the flowchart, see Section 6) and evaluating the results of discriminating instances (marked discriminating results in the flowchart, see Section 7). AutoIG source code and all data and models used in this paper are available at `https://github.com/stacs-cp/AutoIG`.

The main contributions of this paper include:

1. A novel constraint-based framework for generating informative benchmark instances which combines two approaches (*graded* and *discriminating* instance generation) that were previously used in isolation [4, 3].
2. Support for MiniZinc and hand-written instance generators. The new system accepts a user-defined generator as a constraint model, thus allowing problem-specific knowledge to be injected into the instance generation process.
3. Support for the evaluation of local search solvers in addition to systematic solvers. The instance evaluation also considers both solution quality and running time.
4. An extensive evaluation on five problems from the MiniZinc challenge, showing that we can gain new interesting insights that complement the competition's results.

2 Related Work

A series of papers uses evolutionary algorithms and applies *instance space analysis* methods to problems in machine learning (classification [32], regression [33], clustering [17]) and in combinatorial optimisation (personnel scheduling [24], bin packing [27], course timetabling [16]). They use evolutionary algorithms to generate problem instances [43, 42], whereas we take a constraint-based approach. Part of their work is analysing existing instances in benchmark suites and visualising the hardness distribution of instances for particular problems; our framework can be fruitfully combined with their detailed analysis and visualisation methods.

Instance generators have been applied to hard problems in Operations Research as well. For example, NSPLib [48] provides an instance generator and large sets of nurse rostering instances. Their instance generator characterizes an instance through various complexity indicators, including problem sizes, preference distribution measures, coverage distribution measures, and time related constraints. They implement a dedicated procedure for generating instances with properties corresponding to the values of specific indicators as parameters. For the knapsack problem, [37] uses instance generators to identify the regions of the instance space that contain difficult instances. For the traveling thieves problem, [14] uses instance generators that discriminate between more than two options simultaneously.

In communities such as SAT, there have been various works [41, 21] that try to address the generation of instances with desired properties. The SAT competition [19] organisers partly crowdsource the creation of the evaluation set. They require participants to send 20 new instances each, guaranteeing that the competition is run on instances mostly unseen to the solver developers prior to the competition. In addition, a set of previously used instances is manually and carefully selected, using various criteria such as hardness and variety.

The problem of generating a good set of benchmark instances is also studied in the AI planning community [45]. SMAC [23], a tool for optimizing algorithm parameters, is paired with hand-coded programs to generate many sets of instances that smoothly scale in difficulty.

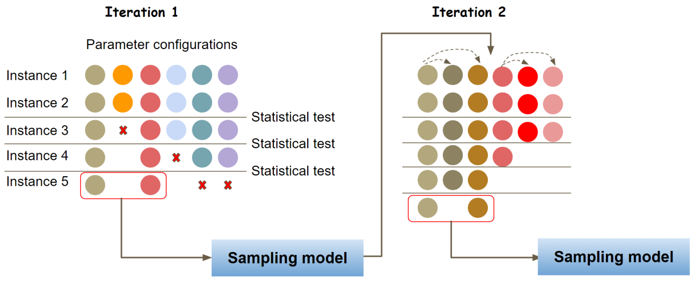


Figure 2 An illustration of IRACE's tuning process.

Afterwards, a subset of the generated sets is selected, according to various criteria such as difficulty and fairness. This results in a set of instances that better reflect the differences between planners when compared to the instances used in the competition.

A related field of study is algorithm configuration/selection, including portfolio-based approaches (SATZilla [49, 50], CPHydra [36], sunny-CP [8, 28]). For these purposes it is important to have a sufficient number of instances with a variety of difficulties that can discriminate between the options [39].

3 Constraint-based Automated Instance Generation

Following the approaches in [4] and [3], our instance generation system AUTOIG makes use of the ESSENCE constraint modelling pipeline [1] and the automated algorithm configurator IRACE [29]. The system receives as input a problem description model, a parameterised instance generator written as a constraint model (referred to as the *generator model*), the solver(s) for which we want to generate graded or discriminating instances, and the types of instances we are interested in (SAT or UNSAT or both). The role of the ESSENCE pipeline is to express the generator model and to create *candidate instances* by solving instances of the generator model (referred to as *generator instances*), while the role of IRACE is to search in the parameter space of the generator model, or in other words, to sample in the generator instance space, to find configurations that can give us candidate instances with the desired properties. In this section, we first describe the search procedure of IRACE (Section 3.1). We then explain how IRACE and constraint modelling are combined in the instance generation process of AUTOIG (Section 3.2). Finally, we discuss in detail how each candidate instance is evaluated during AUTOIG search using gradedness or discriminating criteria (Section 3.3).

3.1 irace's Tuning Process

IRACE [29] is a general-purpose automated algorithm configuration tool for finding the best configurations of a parameterised algorithm. One of its key ideas is *racing* [30]: using statistical tests to eliminate poor configurations early, avoiding wasting computational budget on less promising areas of the configuration space. IRACE leverages this idea with an iterated procedure where each iteration is a *race* among several configurations. Figure 2 illustrates IRACE's tuning process. At the first iteration, a number of random configurations are generated, and a race started by evaluating all configurations on a subset of a given instance set, on a number of random seeds if the algorithm studied is stochastic, or a combination of both. A statistical test is applied to identify and eliminate the worst configurations. Evaluation proceeds with the remaining configurations and a statistical test is conducted

Listing 1 A fragment of an example for RACP problem

```
1  % --- Fragment of MiniZinc model (succ: the immediate successors of tasks) --
2  array [int(1..n_tasks)] of set of int(1..n_tasks): succ;
3  % --- Fragment of generator model, in Essence ---
4  given n_tasks_t : int(1..60)        given s_density : int(1..5)
5  find succ: matrix indexed by [int(1..n_tasks_t)] of set of int(2..n_tasks_t)
6  such that sum([ |succ[t]| | t : int(1..n_tasks_t) ])/n_tasks_t = s_density
7  % --- Fragment of an example generator instance, in Essence ---
8  letting n_tasks_t = 6       letting s_density = 2
9  % --- Fragment of an example candidate instance, in MiniZinc ---
10 succ = [{2, 4, 5, 6}, {3, 4, 5}, {4, 5, 6}, {6}, {6}, {}];
```

Algorithm 1 An evaluation of a generator configuration.

```
Input: generator model $M$, generator instance $G$, solution history $H_G$
Output: penalty $p$
$r \leftarrow solve(M, G, H_G)$        ▷ solve the generator instance $G$ using the ESSENCE pipeline
if $r$ is either UNSAT or timeout on SAVILEROW then
    return $+\infty$        ▷ return a very large penalty, IRACE will discard $G$ immediately
if $r$ is timeout on MINION then
    return 1
$I \leftarrow$ the instance generated by $r$
Add $I$ into $H_G$
$p \leftarrow$ Evaluate $I$ using either GRADED or DISCRIMINATING procedure
return $p$
```

again. This is repeated until only a few good configurations remain or when the budget for the current race has been used. The race is then finished and the surviving configurations are used to update a sampling model. In the next iteration, new configurations are generated based on the updated sampling model and a new race is started. Tuning terminates when a given number of evaluations is exhausted, and the best configuration(s) recorded are returned.

3.2 AutoIG's Instance Generation Process

We give an example of the instance generation process in Listing 1, based on RACP (see Section 4 for details). Fragments of a problem description model, a generator model, a generator instance, and a candidate instance are shown. In this example, a parameter (`succ`) of the problem description model (line 2) is written as a decision variable in the generator model (line 5). The creation of `succ` is controlled by tunable integer parameters of the generator model: `n_tasks_t` (equivalent to `n_tasks` in the original problem description); and `s_density`. Given an instance of the generator model sampled by IRACE (line 8), a candidate instance (line 10) can be created by solving the generator instance.

AUTOIG utilises IRACE for searching in the configuration space of the generator model. The instance generation process starts with IRACE creating a number of random *generator configurations* (a configuration is an instance of the generator model, or in short, a *generator instance*). Each configuration is then evaluated using the procedure described in Algorithm 1 and a penalty is given back to IRACE for the statistical test. The tuning of IRACE then proceeds as normal, interleaving using constraint solving to generate new instances and to evaluate them, and using feedback from the evaluation process to eliminate non-promising configurations and to update the sampling model.

Algorithm 2 An evaluation of an instance using gradedness criteria.

```
1: Input: problem specification $P$, instance $I$, solver $S$, minimum solving time $t_{min}$,
   maximum solving time $t_{max}$, instance types $T$ (that we are interested in)
2: Output: penalty $p$
3: procedure GRADED($P, I, S, t_{min}, t_{max}, T$)
4:     $r \leftarrow solve(P, I, S, t_{max})$        ▷ solve $I$ using $S$ with time limit $t_{max}$, save results to $r$
5:     if $solving\_time(r) < t_{min}$ or $r$ is timeout then
6:         return 0                                  ▷ $I$ is either too easy or too difficult for $S$
7:     if $instance\_type(r) \notin T$ then
8:         return 0                                  ▷ $I$ is not the instance type we are interested in
9:     return -1
```

During each configuration evaluation, the generator instance G is first solved via the ESSENCE pipeline (line 3 of Algorithm 1), whose solving procedure includes two translation steps by the automated constraint modelling tool CONJURE [6, 5] and by SAVILEROW followed by a call to the constraint solver MINION [20]. If G is unsatisfiable or if it is too large to go through the pipeline, a very large penalty is returned so that IRACE will remove the configuration from the current race immediately (line 5). If G is not solved by MINION within the current evaluation, a penalty of 1 is returned. Otherwise, the new candidate instance I is added to the solution history of G to ensure that in the subsequent evaluations of this configuration, the same instance will not be generated again. Solution history is implemented via adding a negative constraint table into the MINION input of G, and this table is constantly updated every time G is evaluated during the tuning. Finally, the candidate instance I is evaluated using one of the two instance evaluation procedures described in Algorithm 2 (for graded instance generation) or Algorithm 3 (for discriminating instance generation), and the corresponding penalty is returned to IRACE. Note that the default setting of IRACE uses the Friedman test, a rank-based statistical test. This is also the setting used by AUTOIG, i.e., the magnitude of difference in the penalty values between evaluations is not taken into account, only the rankings between them matter.

3.3 Evaluating Graded and Discriminating Instances

AUTOIG's instance generation process depends heavily on an effective way of evaluating the quality of candidate instances. In this section, we describe the algorithms used for evaluating whether each candidate instance is graded or for measuring their discriminating power. The algorithms given in this section are invoked in line 10 of Algorithm 1.

To evaluate whether a candidate instance is graded, we employ Algorithm 2. This algorithm has 6 inputs: a problem specification P of the problem under study, an instance I and a solver S to be evaluated, the range of solving times (t_{min} and t_{max}) for the instance to be considered graded for S (to avoid instances that are too easy or too hard to solve), and the type of instances (T) that we are interested in (either satisfiable, unsatisfiable, or both). The instance is first solved by S (line 4) (See Algorithm 2). Results of the solving (r) include the status of the solving process (timeout/UNSAT/SAT), and the returned solution I (if status is SAT). In our experiments S is called via the MINIZINC toolchain. For complete solvers, we use the amount of time to solve the instance to completion (i.e., with a claim of optimality for optimisation problems, or with a feasible solution returned for decision problems or a claim of unsatisfiablity). For local search solvers such as Yuck, since a proof of optimality cannot be achieved for optimisation problems, we use an external complete

Algorithm 3 An evaluation of an instance using discriminating criteria.

```
 1: Input: problem specification $P$, instance $I$, favoured solver $S_F$, base solver $S_B$, minimum
    solving time $t_{min}$ (for $B$ only), maximum solving time $t_{max}$, instance types $T$
 2: Output: penalty $p$
 3: procedure DISCRIMINATING($P, I, S_F, S_B, t_{min}, t_{max}, T$)
 4:     $r_F \leftarrow solve(P, I, S_F, t_{max})$                ▷ solve $I$ using $S_F$ with time limit $t_{max}$
 5:     $r_B \leftarrow solve(P, I, S_B, t_{max})$                ▷ solve $I$ using $S_B$ with time limit $t_{max}$
 6:     if $r_F$ is timeout or $instance\_type(r_F) \notin T$ or $solving\_time(r_B) < t_{min}$ then
 7:         return 0  ▷ $I$ is either too difficult for $S_F$, or not the right instance type, or too
    easy for $S_B$
 8:     $score_F, score_B \leftarrow$ MINIZINC_SCORE($S_F, S_B, P, I$)
 9:     if $score_F = 0$ and $score_B = 0$ then
10:         return 0
11:     return $-score_F / score_B$          ▷ When $score_B = 0$, returns large negative number.
```

solver (called the "oracle") to solve the instance to optimality (with a much longer time limit than t_{max}), and use that to measure the time until S first finds the optimal solution. If the instance turns out to be too easy for S or if the solving process times out (line 5) or the instance type is not interesting to the users (line 7), a penalty of 0 is given back to IRACE. Otherwise, the instance is considered graded and a negative penalty of -1 is returned.

Algorithm 3 is used for evaluating the discriminating power of an instance between two solvers. Each evaluation requires two input solvers: a *favoured solver* S_F and a *base solver* S_B. We want to find instances that are easy to solve by S_F, while being difficult for S_B. The idea is to measure the performance of both solvers on the same instance, and search for instances that maximise the difference in performance. To avoid cases where the performance difference may be due to time measurement sensitivity, we impose a minimum solving time t_{min} on the base solver S_B, i.e., the discriminating instances must be non-trivial to solve by S_B. Similar to the gradedness evaluation, AUTOIG also allows focusing on a particular instance type during the generation process.

The evaluation of the discriminating property starts by applying S_F and S_B on the given instance (lines 4 and 5, Algorithm 3). If the instance does not satisfy our acceptance conditions (incorrect type, too easy for the base solver S_B or unsolvable by the favoured solver S_F (line 6)) a penalty of 0 is returned. Otherwise, we calculate the discriminating power of the instance and use it as feedback to IRACE. The discriminating power is calculated as the ratio between the performance of the favoured solver and the base solver, and the aim of the tuning process is to maximise this ratio. To take into account both solving time and solution quality when evaluating the performance of a solver, we use the *complete scoring* approach of the MINIZINC competitions. After calculating the MINIZINC scores of both solvers (line 8), the discriminating score is calculated as the MINIZINC score of S_F divided by the MINIZINC score of S_B and the negation of that ratio is returned to IRACE (line 11). Note that when both MINIZINC scores are equal to 0, the discriminating score is set to 0 (line 10).

The MINIZINC (complete) score for calculating the relative performance of two solvers on an instance can be found on the competition website (`https://www.minizinc.org/challenge2021/rules2021.html#assessment`). For completeness, in the rest of this section we will describe this score calculation in detail.

Given a solver S, a problem model P and an instance I, the following information is collected for the calculation: TIME(S, P, I) – the solving time of S on I; SOLVED(S, P, I) – whether a correct solution or a correct unsatisfiability result for I is returned by S; QUAL-

Algorithm 4 Check whether one solver performs better than another in terms of solution quality.

```
 1: Input: solver $A$, solver $B$, problem model $P$, instance $I$
 2: procedure $\textsc{IsBetter}(A, B, P, I)$
 3:     if $P$ is a decision problem then
 4:         return $\textsc{solved}(A, P, I)$ and not $\textsc{solved}(B, P, I)$
 5:     else
 6:         return $(\textsc{solved}(A, P, I)$ and not $\textsc{solved}(B, P, I))$ or
 7:         $(\textsc{optimal}(A, P, I)$ and not $\textsc{optimal}(B, P, I))$ or
 8:         $(\textsc{quality}(A, P, I)$ is better than $\textsc{quality}(B, P, I))$
```

Algorithm 5 MiniZinc score calculation between two solvers.

```
 1: Input: solver $A$, solver $B$, problem model $P$, instance $I$
 2: procedure $\textsc{MiniZinc\_Score}(A, B, P, I)$
 3:     if $\textsc{IsBetter}(A, B, P, I)$ then
 4:         $score_A \leftarrow 1$, $score_B \leftarrow 0$
 5:     else if $\textsc{IsBetter}(B, A, P, I)$ then
 6:         $score_A \leftarrow 0$, $score_B \leftarrow 1$
 7:     else if $\textsc{solved}(A, B, P, I)$ then
 8:         $score_A \leftarrow \textsc{time}(B, P, I)/(\textsc{time}(A, P, I)+\textsc{time}(B, P, I))$
 9:         $score_B \leftarrow 1 - score_A$
10:     else
11:         $score_A \leftarrow score_B \leftarrow 0$
12:     return $score_A$ and $score_B$
```

$\textsc{ity}(S, P, I)$ – the best objective value obtained by S; and $\textsc{optimal}(S, P, I)$ – whether a claim of optimality is returned by S. Based on those information, the function $\textsc{IsBetter}(A, B, P, I)$ (Algorithm 4) determines whether solver A is clearly better than solver B in terms of solution quality, for decision problems (line 4) and for optimisation problems (lines 6-8).

Finally, the MiniZinc complete score when comparing two solvers on an instance I is calculated in Algorithm 5. The calculation starts with checking whether one of the two solvers is better than the other in term of solution quality (lines 3-6). If that is not the case, there are two possibilities. First, I is solved by both solvers, and for optimisation problems, the same solution quality is achieved by both. In that case the normalised solving times are used as the scores. Second, both solvers fail to solve I, and in that case a score of 0 is returned for both. Note that this is slightly different from the scoring used in the MiniZinc competitions, where the scores of 1 and 0 are given to A and B, respectively. This is because the final competition ranking is based on the Borda counting system, where the score is calculated for all pairs of solvers, including the same pair in the opposite order.

4 Case Studies

In this section we describe the five problems that are used to evaluate AutoIG, and also the set of four solvers that are used in our experiments. The five problems being used in this study are taken from the latest MiniZinc Challenges. They are chosen with the aim of covering a variety of different problem properties, including the existence of redundant and symmetry breaking constraints, the usage of different global constraints, and a range of problem domains. In this section, we give a brief overview of those problems and how their instance generation problems are modelled.

Multi-Agent Collaborative Construction problem (MACC) [26]. This is a planning problem that involves constructing a building by placing blocks in a 3D map using multiple identical agents. Ramps must be built to access the higher levels of the building. The objective is to minimise the makespan (primary) and the total cost (secondary).

In addition to the basic parameters of a MACC instance indicated in the problem specification (i.e., the number of agents, the time horizon and the map sizes), the instance generation process should include information about the building itself as this is likely to affect instance difficulty. Therefore, two parameters and related constraints are added to the generator model to represent the density of the building on the ground level and its average height.

Carpet Cutting problem (CARPET-CUTTING). The Carpet Cutting Problem [40] is a packing problem in which room and stair carpets composed of rectangular sections must be packed onto a carpet roll of fixed width and whose length must be minimised. The problem is complicated by the ability to rotate the carpets to aid in the packing process.

This problem requires substantial instance data, including the specification of the constituent rectangles of each carpet, their dimensions, and the permitted carpet rotations. There are several implicit constraints on this data that are not captured in the original MiniZinc model and hence these must be injected into the instance generation process through our generator specification. In particular, the rectangles that comprise a carpet must not overlap and must form a contiguous shape, as well as have bounded sizes so as to avoid trivially unsatisfiable instances.

Mario problem (MARIO). The Maximum Profit Subpath Problem is a routing problem that requires us to find a path in a graph where the path endpoints are given. This path is subject to two main constraints, where the sum of weights associated to arcs in the path is restricted (fuel consumption), while the sum of weights associated to nodes in the path has to be maximized (reward).

Regarding the instance generation process, in addition to the basic parameters, the amount of reward per node is represented as a non-negative integer array, while the non-negative cost for each arc is represented as a 2-dimensional matrix. There are a few implicit constraints not represented in the MiniZinc model, where the initial and goal nodes are different and have 0 reward, and the cost matrix is symmetric on the diagonal.

Resource Availability Cost Problem (RACP). The Resource Availability Cost Problem [25] is a scheduling problem with activities that are non-interruptible and have a fixed duration. The problem includes precedence constraints between pairs of activities i, j (that require activity i to be completed before activity j begins), arranged in a directed acyclic graph. There are a set of renewable resources, and each activity (when running) requires a given amount of each resource. All activities must be completed by a given deadline. Each resource has a cost per unit, and the objective is to minimise the peak costs of the resources.

The durations of activities, unit costs of resources, and resource demands of activities are all matrices of integers without complex constraints. However, the precedence graph (represented as a set of successors for each activity) has implicit constraints that are not represented in the MiniZinc model. Firstly, it must be acyclic, and we achieve this by mapping activities to numbered layers and allowing only edges from lower to higher-numbered layers. Secondly, we ensure that each activity has at least one predecessor and at least one successor (except the dummy first and last activities).

Discrete Lot Sizing problem (LOT-SIZING). The Discrete Lot Sizing and Scheduling Problem [22, 46] (CSPLib 58) requires us to find a production schedule for a set of orders, each with a due date within a planning horizon. There are various costs associated with production, such as setup, changeover and stocking costs, the sum of which must be minimised.

This problem requires substantial instance data including the type and due date of each order, and moreover a table of changeover costs between orders. There are a number of implicit constraints on this data, including a dummy order type 0 which incurs 0 cost to change to/from, and the fact that the changeover costs for the remaining order must obey the triangle inequality. Again, these are not captured in the original MiniZinc model and hence must be injected into the instance generation process through our generator specification.

We investigate the performance of four solvers, also taken from the MiniZinc challenges, on the problems described above using our framework. They are chosen such that a variety of solving techniques and different competition rankings are included. The solvers are: OR-Tools [2] (version 9.2) – a systematic solver from Google that combines CP, SAT, and linear programming techniques; Picat-SAT [51] – a SAT compiler for the multi-paradigm programming language Picat which uses KISSAT [12] as the underlying SAT solver; Chuffed [15] (version 0.10.4) – a clause learning CP solver which was not a participant of the challenges but was used in the score calculation process to rank participating solvers; and Yuck [31] (version 20210501) – a constraint-based local search solver.

OR-Tools has consistently won the last several competitions and Picat-SAT has received multiple silver medals. Yuck is the winning solver in the Local Search category of the 2020 and 2021 competitions. However, its ranking was generally low when compared to OR-Tools and Picat-SAT. In particular, based on the competition data, it was completely dominated by OR-Tools on the five problems considered.

5 Experimental Setup

The first set of experiments are on generating graded instances. For each problem, we first generate graded instances for each solver via an AutoIG experiment with a budget of $2,000$ runs. Note that a run is an evaluation of a generator configuration. The gradedness criteria is defined as being solvable by the given solver with the time ranging from 10 seconds (to avoid trivial instances) to 20 minutes (the time limit used by the MiniZinc Challenge). Following the competition approach, MiniZinc translation time is included in the total time measured. Since Yuck is a local search solver, we use OR-Tools (with a budget of 1 hour) for checking whether a solution returned by Yuck is optimal. After all graded instances are collected, we then randomly select 50 graded instances from each experiment to get a combined benchmark instance set for each problem. Finally, we evaluate the performance of all four solvers on the combined instance set.

The second set of experiments are on generating discriminating instances. Since OR-Tools has consistently shown very strong performance on the competition data, the main aim of these experiments is to see whether we can find instances where OR-Tools is performing worse than the other two participating solvers being considered. We do this without loss of generality: our discriminating instance generation procedure can be applied to any pair of solvers. We compare two solvers (Picat-SAT and Yuck) against OR-Tools. For each solver we conduct two separate AutoIG experiments, one where we search for instances that are solved more quickly by OR-Tools and one for the opposite case. The same AutoIG budget and memory limit as in graded experiments are used. To avoid instances where the difference between the performance of two solvers is due to fluctuations in running time measurement, a minimum requirement of 10 seconds is imposed on the solving time of the base solver, i.e. instances that can be trivially solved by the base solver are discarded.

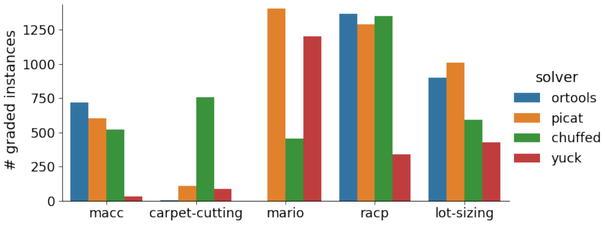


Figure 3 Number of graded instances generated.

All experiments were performed on a computing node of the High Performance Computing cluster [name omitted to preserve anonymity]. Each node is equipped with two 2.1 GHz, 18-core Intel Xeon processors and 256 GB RAM. Each solver except Yuck is given a memory limit of 8GB via the RUNSOLVER tool [38]. For Yuck, the memory limit is controlled directly via the Java Runtime Environment (JRE). For solving the generator models, time limits of 5 and 10 minutes are given to SAVILEROW and MINION, respectively. In this work, we focus on the Free Track of the competitions. Therefore, all solvers are called via the MINIZINC toolchain with a single core and with the free search option being passed to the solver. Although AUTOIG supports focusing on generating either only SAT or only UNSAT instances, in this work we allow both types of instances to be generated.

6 Results on graded instances

First we describe the sets of graded instances produced by AUTOIG for the five problems (Section 6.1) and discuss insights obtained from analysing the results. Then in Section 6.2 we combine the sets of graded instances for each problem, and re-evaluate the four solvers using the combined sets of instances, showing substantially different relative performance in some cases compared with the competition instances.

6.1 Graded instance generation

For each problem, Figure 3 shows the number of graded instances obtained per solver within the given budget. While we can achieve more than a few hundred graded instances in most cases, there are cases where we are only able to generate a small number of instances. For example, with OR-Tools on CARPET-CUTTING and MARIO, we generate only 4 and 1 graded instances, respectively. In addition, the numbers are fairly small for Yuck on MACC and CARPET-CUTTING. There is a large variation in the number of graded instances we are able to generate for different problems and solvers (shown in Figure 3).

The differences in the number of graded instances returned by each experiment suggest that the performance of the solvers varies significantly when solving instances drawn from the same instance space. In order to better understand the performance distribution of each solver we investigate the details of the search space of AUTOIG. More specifically, we check the status of each configuration evaluation run and measure their frequency, as detailed in Figure 4. For OR-Tools on CARPET-CUTTING and MARIO, only a small number of graded instances are found, but this same outcome has entirely different causes. For CARPET-CUTTING, almost half of the runs are with unsolvable generator configurations, and for the rest the candidate instances are mostly trivially proved unsatisfiable by OR-Tools. For MARIO, the majority of the runs produce instances that are trivially satisfiable. Once we understand the underlying reason for the lack of graded instances, we can rectify each

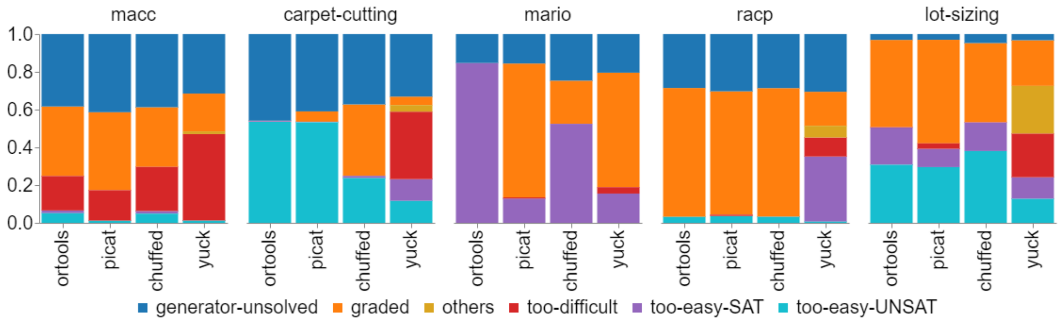


Figure 4 Frequency of all run statuses, including `generator-unsolved` (generator instance is UNSAT or unsolvable); `graded` (a graded instance is obtained); `too-difficult` (the candidate instance is unsolvable by the considered solver within the time limit); `too-easy-SAT` and `too-easy-UNSAT` (the candidate instance is too easy, i.e., solved within less than 10 seconds); and `others` (the considered solver fails due to unexpected errors such as incorrect returned answers).

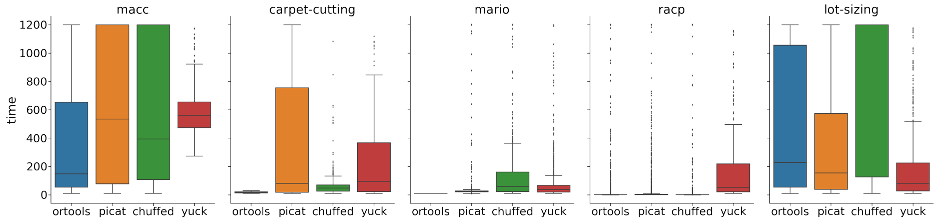


Figure 5 Solving time of graded instances generated for each pair of problems and solvers. Note that the instances presented here are the graded instances found for each solver independently. The performance of these solvers on the combined set of graded instances can be seen in Figure 6.

of these shortcomings: for CARPET-CUTTING, expert knowledge on the problem may be added as constraints to the generator model to avoid trivially unsatisfiable instances, while for MARIO, the current instance space may be too easy for OR-Tools and we may want to increase the upper bounds of some of the generator parameters. On the other hand, the situation is completely different for Yuck: the small number of graded instances obtained for MACC and CARPET-CUTTING is largely due to the fact that the majority of instances generated are too difficult to solve.

In addition to the run statuses, the distribution of solving time of graded instances also gives us interesting insights into the performance of different solvers, as illustrated in Figure 5. Notably, many graded instances for MARIO and RACP are close to the lower bound of graded instances; this is true for all solvers. Nevertheless, AUTOIG is able to find challenging graded instances, which can take several hundred seconds to solve, for all solvers on those two problems (except for OR-Tools on MARIO). For CARPET-CUTTING, OR-Tools and Chuffed can solve most graded instances quickly, while Picat-SAT and Yuck take more time in general. Finally, for MACC and LOT-SIZING, the solving time distributions of all four solvers are more well-spread, indicating a good diversity of difficulties among the generated graded instances.

Note that for the majority of graded instances generated, the MINIZINC flattening times are generally marginal compared to the time taken to solve them. This indicates that the more difficult graded instances are actually challenging for the solvers themselves, and can be useful for solver developers to improve their solver performance.

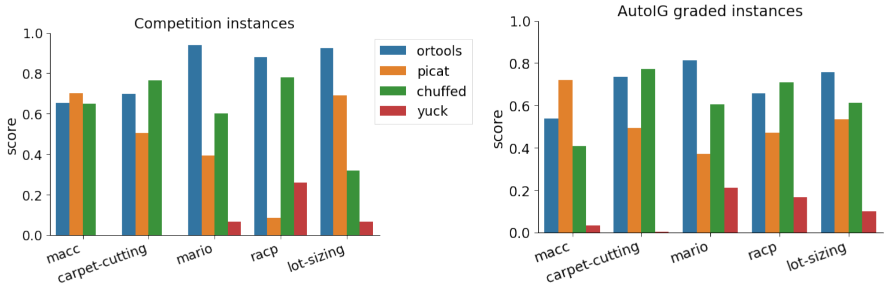


Figure 6 MiniZinc Borda (complete) scores of each solver on the MiniZinc Challenges instance set (left) and on the combined graded instance set generated by AutoIG (right).

6.2 Comparison of Solver Performance on Graded Instances

We combine all graded instances to construct a diverse set of instances for each problem. We then evaluate all four solvers on the combined set and rank them using the Borda (complete) scoring method of the MiniZinc Challenge (`https://www.minizinc.org/challenge2021/rules2021.html`). More specifically, for each problem, 50 graded instances are uniformly sampled from the set of graded instances for each solver. In cases where there are less than 50 graded instances available, we just take them all. For comparison, we also evaluate those solvers on the instances used in the competition. There are 5-10 instances per problem, as some problems are re-used over two different competitions.

Figure 6 shows the scores on the competition instances (left) and on the combined graded instances generated by AutoIG (right). There are similarities between results on the two sets of instances. Performance of OR-Tools and Chuffed remain strong in most cases, followed by Picat-SAT. For macc, carpet-cutting and mario, the overall rankings of the four solvers on both groups are almost the same. However, results on the graded set do show certain changes in relative performance of all solvers. For example, the scores of Yuck on the graded instances are no longer zero for macc and carpet-cutting, and the score for mario increases noticeably. This indicates that Yuck is actually not completely dominated by all other solvers on those three problems as suggested by the competition data. For racp, the ranking has changed significantly: OR-Tools swaps places with Chuffed, and Picat-SAT swaps places with Yuck. For lot-sizing, Picat-SAT is no longer ranked higher than Chuffed.

Thanks to the solution checking process being integrated into each evaluation, we also found a number of cases from the combined graded sets where incorrect answers are returned, which can be of separate interest to the solver developers. There were 41 (out of 183) macc instances and 90 (out of 154) carpet-cutting instances (from the subset of graded instances generated for other solvers) where Yuck reports objective values of infeasible solutions.

Generating a larger number of graded instances for each solver and analysing them using the presented methods gives more information in comparison to a typical competition's result, which would be a ranking of the solvers. In Section 7 we apply the discriminating instance generation feature of AutoIG to gain even more insight into solver performance.

7 Results on Discriminating Instances

Results on MiniZinc competition data indicate that OR-Tools is a very strongly performing solver on the 5 problems considered. It completely dominates Yuck, i.e., Yuck gets zero score on all competition instances when compared directly to OR-Tools. OR-Tools also wins over Picat-SAT on all instances of mario and racp, on 9 out of 10 instances of lot-sizing, and

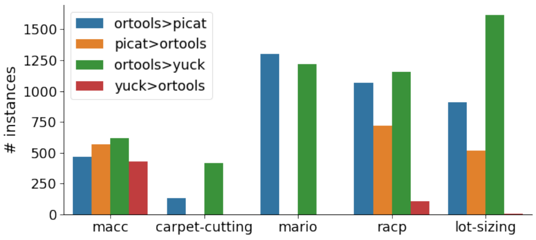


Figure 7 Number of discriminating instances generated per *favoured* and *base* solver pair.

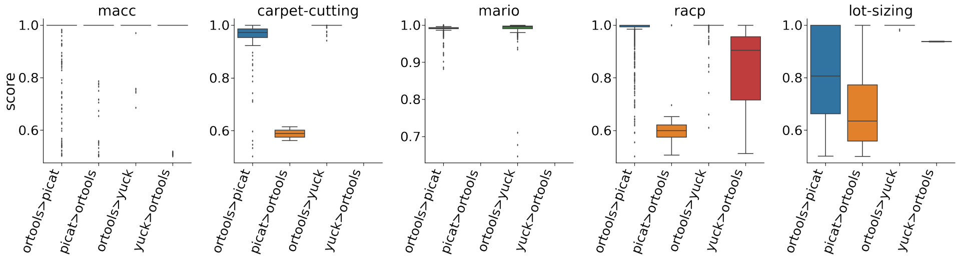


Figure 8 Distribution of scores (of the winning solver) on discriminating instances generated.

on 8 out of 10 instances of CARPET-CUTTING. However, detailed results obtained from the evaluation on graded instances suggest that this may not always be the case. For example, there are 31 instances evaluated on RACP where Picat-SAT performs better than OR-Tools, and 58 MACC instances where Yuck performs better. In this section, we use the discriminating instance generation feature of AUTOIG to get more insights into these cases.

Figure 7 shows the number of discriminating instances generated for the two pairs of solvers. In the experiments on OR-Tools versus Yuck, AUTOIG found 431 MACC instances and 110 RACP instances where Yuck gets a better score than OR-Tools, which indicates that Yuck is not completely dominated by OR-Tools on these two problems. On the other hand, for CARPET-CUTTING and MARIO, results suggest that Yuck may indeed be entirely dominated by OR-Tools, as no instances were found in the experiments that favour Yuck. Furthermore, for LOT-SIZING, only 3 discriminating instances favouring Yuck are found. In the experiment on OR-Tools versus Picat-SAT, OR-Tools shows domination on both CARPET-CUTTING (only 2 instances where Picat-SAT is better than OR-Tools were found) and MARIO (no instances favouring Picat-SAT was found). On the other three problems, there are a good number of discriminating instances in both directions.

The number of discriminating instances tell us if winning instances for a solver can be found, but it does not show the magnitude of the difference in performance. We can get additional insights into comparative performance of the solvers by looking into the detailed scores of the winning solver on discriminating instances for each experiment. As shown in Figure 8, for MACC, the median lines indicate that for all four cases, several discriminating instances found have the highest “discriminating power”, i.e., the winning solver gets the maximum score of 1 (the other solver, in turn, gets zero score). This type of instance is probably the most interesting for understanding the shortcomings of a particular solver. For CARPET-CUTTING, on the only 2 discriminating instances where Picat-SAT has better score than OR-Tools, the score distribution of the corresponding experiment (Picat-SAT>OR-Tools) suggests that OR-Tools performance is not much worse. This suggests that OR-Tools indeed dominates Picat-SAT on this problem. A similar conclusion can be reached for Yuck, i.e., it is clear that OR-Tools is really the dominating solver on CARPET-CUTTING since the magnitude of the performance difference is very small even for the instances where Yuck is faster. Similarly, for MARIO, OR-Tools very clearly dominates in comparison to Picat-SAT and Yuck, as indicated by the discriminating score distributions. This is in line with what was observed in the previous section's results on the same problem.

Interestingly, for RACP, although the number of discriminating instances of Picat-SAT>OR-Tools is larger than of Yuck>OR-Tools as shown in Figure 7, the magnitude of the performance difference of instances found for Yuck is generally much higher. This observation gives a new insight that has not been revealed in all previous experiments on gradedness: even though the performance of Yuck is dominated by other solvers in general (i.e., it is ranked lower) and it has a smaller number of discriminating instances favouring it, the magnitude of the performance difference is very large for these instances. This means there exists a subset of the RACP instances where Yuck's performance is much better than OR-Tools, while this does not seem to be the case for Picat-SAT.

The insights provided by discriminating instances could be useful in constructing a robust portfolio of solvers for a given problem. For example, on RACP, Yuck is the weakest solver by a wide margin on the graded instances (see Figure 6) and second-weakest on competition instances. On the graded instances, Picat-SAT performs considerably better than Yuck. However, the results with discriminating instances show that Yuck would be a good candidate to add to a portfolio (alongside OR-Tools) whereas Picat-SAT may not be.

8 Conclusions and Future Work

Assessing the performance of solving methods via benchmark problems is fundamental to CP research. However, its utility is limited by the availability of problem instances that are of suitable difficulty, and diverse (not inadvertently favouring one solver over another). We have shown that our system AUTOIG can generate large numbers of informative benchmark instances graded for difficulty for a single solver, or that can discriminate between two solvers (favouring one or the other). The only manual part of the AUTOIG process is to capture (in a generator model) any implicit constraints on the instances data.

The essential task of benchmarking is to compare multiple solvers and rank them. As illustrated in our experiments, AUTOIG can be used to generate graded instances for each solver independently, and these can then be combined into one set of instances, providing confidence that the generation process does not favour one solver or class of solvers. Furthermore, we have shown that automatically generated instances can provide more detailed insights than just a ranking. Instances generated by AUTOIG can reveal cases where a solver is weak or even faulty, providing valuable information to solver developers. Finally, discriminating instances can reveal parts of the instance space where a generally weak solver performs well relative to others, and therefore could be useful as part of a portfolio.

There are various directions for future improvement. First, the diversity of instances found during search can be taken into account to increase the quality of the final instance set. This would require a definition of diversity, which could be based on problem-specific instance features or on general constraint programming features such as the FZN2FEAT features [7]. Secondly, similar to the series of work on Instance Space Analysis (e.g. [32, 24, 16]), a detailed visualisation of the instance space based on performance data collected from the tuning and evaluation process of AUTOIG would provide further insights into performance of the solvers under study. Again, instance features would be needed for such analysis.

References

1 ESSENCE modelling pipeline:. `https://constraintmodelling.org/`.

2 Google OR-Tools, 2021. Available from `https://github.com/google/or-tools`.

3 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 *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 41–51. Springer, 2020.

4 Özgür Akgün, Nguyen Dang, Ian Miguel, András Z Salamon, and Christopher Stone. Instance generation via generator instances. In *International Conference on Principles and Practice of Constraint Programming*, pages 3–19. Springer, 2019.

5 Ozgur Akgun, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Breaking conditional symmetry in automated constraint modelling with Conjure. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI)*, pages 3–8, 2014.

6 Ozgur Akgun, Ian Miguel, Christopher Jefferson, Alan M Frisch, and Brahim Hnich. Extensible Automated Constraint Modelling. In Wolfram Burgard and Dan Roth, editors, *AAAI 2011 - Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. AAAI Press, 2011.

7 Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An enhanced features extractor for a portfolio of constraint solvers. In *Proceedings of the 29th annual ACM symposium on applied computing*, pages 1357–1359, 2014.

8 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.

9 Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of model checking*, pages 305–343. Springer, 2018.

10 Thomas Bartz-Beielstein, Carola Doerr, Daan van den Berg, Jakob Bossek, Sowmya Chandrasekaran, Tome Eftimov, Andreas Fischbach, Pascal Kerschke, William La Cava, Manuel Lopez-Ibanez, et al. Benchmarking in optimization: Best practice and open issues. *arXiv preprint arXiv:2007.03488*, 2020.

11 Vahid Beiranvand, Warren Hare, and Yves Lucet. Best practices for comparing optimization algorithms. *Optimization and Engineering*, 18(4):815–848, 2017.

12 Armin Biere, Mathias Fleury, and Maximilian Heisinger. CaDiCaL, Kissat, Paracooba entering the SAT competition 2021. In T Balyo, N Froleyks, M Heule, M Iser, M Järvisalo, and M Suda, editors, *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2021-1, Department of Computer Science, University of Helsinki, Helsinki*, 2021.

13 Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

14 Jakob Bossek and Markus Wagner. Generating instances with performance differences for more than just two algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1423–1432, 2021.

15 Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed, 2018. Available from `https://github.com/chuffed/chuffed/`.

16 Arnaud De Coster, Nysret Musliu, Andrea Schaerf, Johannes Schoisswohl, and Kate Smith-Miles. Algorithm selection and instance space analysis for curriculum-based course timetabling. *Journal of Scheduling*, pages 1–24, 2021.

17 Luiz Henrique dos Santos Fernandes, Ana Carolina Lorena, and Kate Smith-Miles. Towards understanding clustering problems and algorithms: an instance space analysis. *Algorithms*, 14(3):95, 2021.

18 Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artificial Intelligence*, 301:103572, 2021.

19 Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artificial Intelligence*, 301:103572, 2021. doi:10.1016/j.artint.2021.103572.

20 Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *Proceedings ECAI 2006*, pages 98–102, 2006.

21 Jesús Giráldez-Cru and Jordi Levy. A modularity-based random SAT instances generator. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI*, pages 1952–1958. AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/277.

22 Vinasétan Ratheil Houndji, Pierre Schaus, Laurence Wolsey, and Yves Deville. The stockingcost constraint. In *International conference on principles and practice of constraint programming*, pages 382–397. Springer, 2014.

23 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*, volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer, 2011. doi:10.1007/978-3-642-25566-3_40.

24 Lucas Kletzander, Nysret Musliu, and Kate Smith-Miles. Instance space analysis for a personnel scheduling problem. *Annals of Mathematics and Artificial Intelligence*, 89(7):617–637, 2021.

25 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.

26 Edward Lam, Peter J Stuckey, Sven Koenig, and TK Kumar. Exact approaches to the multi-agent collective construction problem. In *International Conference on Principles and Practice of Constraint Programming*, pages 743–758. Springer, 2020.

27 Kelvin Liu, Kate Smith-Miles, and Alysson Costa. Using instance space analysis to study the bin packing problem, 2020.

28 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.

29 Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

30 Oden Maron and Andrew W Moore. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review*, 11(1):193–225, 1997.

31 Michael Marte. Yuck, 2021. Available from https://github.com/informarte/yuck.

32 Mario A Muñoz, Laura Villanova, Davaatseren Baatar, and Kate Smith-Miles. Instance spaces for machine learning classification. *Machine Learning*, 107(1):109–147, 2018.

33 Mario Andrés Muñoz, Tao Yan, Matheus R Leal, Kate Smith-Miles, Ana Carolina Lorena, Gisele L Pappa, and Rômulo Madureira Rodrigues. An instance space analysis of regression problems. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 15(2):1–25, 2021.

34 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.

35 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.

36 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.

37 David Pisinger. Where are the hard knapsack problems? *Computers & Operations Research*, 32(9):2271–2284, 2005.

38 Olivier Roussel. Controlling a solver execution with the runsolver tool. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):139–144, 2011.

39 Marius Schneider and Holger H Hoos. Quantifying homogeneity of instance sets for algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 190–204. Springer, 2012.

40 Andreas Schutt, Peter J Stuckey, and Andrew R Verden. Optimal carpet cutting. In *International Conference on Principles and Practice of Constraint Programming*, pages 69–84. Springer, 2011.

41 Bart Selman, David G. Mitchell, and Hector J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2):17–29, 1996. `doi:10.1016/0004-3702(95)00045-3`.

42 Kate Smith-Miles, Jeffrey Christiansen, and Mario Andrés Muñoz. Revisiting where are the hard knapsack problems? via instance space analysis. *Computers & Operations Research*, 128:105184, 2021.

43 Kate Smith-Miles and Jano van Hemert. Discovering the suitability of optimisation algorithms by learning from evolved instances. *Annals of Mathematics and Artificial Intelligence*, 61(2):87–104, 2011.

44 Peter J Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15(3):307–316, 2010.

45 Alvaro Torralba, Jendrik Seipp, and Silvan Sievers. Automatic instance generation for classical planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 376–384, 2021.

46 Hafiz Ullah and Sultana Parveen. A literature review on inventory lot sizing problems. *Global Journal of Research In Engineering*, 10(5), 2010.

47 Mauro Vallati, Lukás Chrpa, and Thomas Leo McCluskey. What you always wanted to know about the deterministic part of the international planning competition (IPC) 2014 (but were too afraid to ask). *Knowledge Engineering Review*, 33:e3, 2018. `doi:10.1017/S0269888918000012`.

48 Mario Vanhoucke and Broos Maenhout. On the characterization and generation of nurse scheduling problem instances. *European Journal of Operational Research*, 196(2):457–467, 2009.

49 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.

50 Lin Xu, Frank Hutter, Jonathan Shen, Holger H Hoos, and Kevin Leyton-Brown. SATzilla2012: Improved algorithm selection based on cost-sensitive classification models. *Proceedings of SAT Challenge*, 2012, 2012.

51 Neng-Fa Zhou and Håkan Kjellerstrand. Optimizing SAT encodings for arithmetic constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 671–686. Springer, 2017.

Sequence Variables for Routing Problems

Augustin Delecluse
TRAIL, ICTEAM, UCLouvain, Louvain-la-Neuve, Belgium

Pierre Schaus
ICTEAM, UCLouvain, Louvain-la-Neuve, Belgium

Pascal Van Hentenryck
Georgia Institute of Technology, Atlanta, GA, USA

Abstract

Constraint Programming (CP) is one of the most flexible approaches for modeling and solving vehicle routing problems (VRP). This paper proposes the *sequence variable* domain, that is inspired by the insertion graph introduced in [4] and the subset bound domain for set variables. This domain representation, which targets VRP applications, allows for an efficient insertion-based search on a partial tour and the implementation of simple, yet efficient filtering algorithms for constraints that enforce time-windows on the visits and capacities on the vehicles. Experiment results demonstrate the efficiency and flexibility of this CP domain for solving some hard VRP problems, including the Dial-A-Ride, the Patient Transportation, and the asymmetric TSP with time windows.

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

Keywords and phrases Constraint Programming, Dial-A-Ride, Patient Transportation, TSPTW, Vehicle Routing, Sequence Variables, Insertion Variables

Digital Object Identifier 10.4230/LIPIcs.CP.2022.19

Supplementary Material *Software (Source Code)*: `https://github.com/augustindelecluse/minicp-sequences`; archived at `swh:1:dir:144eae8f012ef66bb6f2289b1419f48b6779e294`

Funding *Augustin Delecluse*: This work was supported by Service Public de Wallonie Recherche under grant n°2010235 – ARIAC by DIGITALWALLONIA4.A.

1 Introduction

Vehicle routing problems (VRP) [27] are of great importance for the distribution of goods in the supply chain. In order to cope with increasing urbanization and ecological challenges, it is also expected that flexible transport offers for people, such as on-demand transport, will have to be further developed in the future [12]. This raises new challenges for optimization, in particular the development of generic and reusable tools in many contexts and variants of VRP.

Constraint Programming (CP) is one of the most flexible approaches for modeling vehicle routing problems (VRP). One standard model consists in using the so-called successor model with one variable for each visit that represents the next visit in the tour of a vehicle [3]. Due to its simplicity, this model has two practical limitations involving both the modeling and research components of CP. At the modeling level, it is not straightforward to represent the optional aspect of some visits in the successor model. This requires the introduction of a fake vehicle visiting all excluded visits. At the search level, sub-chains formed by fixed successors do not allow any insertion in the middle of a partial tour during the search. Sub-chains that are formed close to the root during the search are fixed with little information and hardly reconsidered in large search spaces explored with a backtracking search. The goal of the sequence variable is to address those two limitations.

1. It can easily model the exclusion of visits not inserted in a tour similarly to a set variable.
2. Inspired by the idea of the *insertion graph* [4], it allows the insertion of a visit in the middle of the partial tour enabling the implementation of a depth first tree search insertion

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 19; pp. 19:1–19:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

exploration algorithm similar to the ones used in [4, 15] to reinsert optimally a set of relaxed visits in a large neighborhood search (LNS).

We introduce the *Sequence Variable* domain as well as the mechanism to make the domain reversible in a trail-based solver. Two important constraints and their associated filtering algorithms are described: The *TransitionTimes* constraint links the time window constraints of the visits to a distance matrix, removing insertions that would process a request outside of its time window; The *Cumulative* constraint ensures that the load change performed in each visit never exceeds the maximum capacity of a vehicle. We model three constrained VRP problems with the *Sequence Variable* that are illustrative of the functionality offered for both the modeling and the search flexibility: The Dial-A-Ride Problem (DARP) [8, 15], the Patient Transportation Problem (PTP) [5, 19] and the Traveling Salesman Problem with time windows (TSPTW). Experimental results demonstrate the effectiveness of the approach. It obtains better results than baseline models with sequences of conditional task interval variables in CP Optimizer [18] and obtains results competitive with the state-of-the art results published in the literature on the DARP and PTP. For TSPTW we improve the best know solutions for 32 out of the 205 instances tested in the standard benchmark suite [20].

The rest of the paper is organised as follows. Section 2 details related work on VRP and existing Sequence Variables. Section 3 dives into the definition of a Sequence Variable, its interface and implementation. Section 4 shows how VRP constraints are implemented using the variable. Lastly, our models and experimental results for DARP, PTP and TSPTW are presented in Section 5.

2 Related Work

In [26], the authors introduced a Sequence Variable for scheduling and routing problems. This domain representation directly extends the subset bound domain representation for set variables [14] by partitioning the visits into a required, possible and excluded set plus a partial sequence and a set of insertion points. In this work we simplify this idea by getting rid of the required set. As a consequence, a possible visit must be directly scheduled in the partial sequence but cannot be required without being inserted in the sequence. This modification, despite its simplicity, greatly eases the reasoning made by the constraints, their time complexity and the implementation of search heuristics, while losing little to no flexibility in practice. The proposed sequence domain can be seen as the making of the *insertion graph* idea introduced in [4] more generic and encapsulated as the internal implementation of the sequence variable domain.

Although not published, IBM ILOG CP Optimizer [18] also proposes sequence variables to decide the order of visits. The functionalities and constraints of this sequence variable are briefly described [16, 17], no details are given about the exact implementation. According to their documentation [6, 7], they use a Head-Tail structure, maintaining a separate growing head and tail to add new Interval variables to the beginning or end of the sequence, respectively. The head and tail merge to form the final sequence once no Interval can be added to either of them. This implementation appears to be similar to Google OR-Tools [23] and its own Sequence Variables [24]. Although more targeted to scheduling problems, this sequence was used for solving the PTP in [5] and [19], and is suitable for VRP.

3 Sequence Variable

We first introduce useful notations related to sequences and operations on these. We then formally define the domain of a sequence variable before considering the practical algorithmic details for implementing this domain in a constraint programming solver.

3.1 Sequence notations

The notations are largely borrowed from [26] but reintroduced in this paper for reading convenience. The set of locations that can be visited by a vehicle are referred to as *nodes* and the set of all nodes is denoted as $\mathcal{X}$. A sequence over $\mathcal{X}$ is denoted by $\overrightarrow{S}$ and the set of all sequences of $\mathcal{X}$ by $\overrightarrow{\mathcal{P}}(\mathcal{X})$. The sequence $\overrightarrow{S}$ defines an order over the elements of S. The set of elements in the sequence $\overrightarrow{S}$ is denoted S. All the entries of the sequence are different and therefore $|\overrightarrow{S}| = |S|$. The notation $p \overset{\overrightarrow{S}}{\prec} q$ means that the node p precedes node q in $\overrightarrow{S}$ and $p \xrightarrow{\overrightarrow{S}} q$ means that p directly precedes q in $\overrightarrow{S}$. Those notations are simply written $p \prec q$ and $p \rightarrow q$ when clear from the context. If the nodes can be equal, the relation is written $p \preceq q$. A sequence can be grown by using an insertion operation $insert(\overrightarrow{S}, p, q)$ with $p \in S, q \notin S$ that results in inserting q just right after p in the sequence. More exactly assuming $\overrightarrow{S} = \overrightarrow{S}_1 \cdot p \cdot \overrightarrow{S}_2$ the resulting super-sequence is $\overrightarrow{S}' = \overrightarrow{S}_1 \cdot p \cdot q \cdot \overrightarrow{S}_2$. The operation is also noted $\overrightarrow{S} \underset{(p,q)}{\Longrightarrow} \overrightarrow{S}'$. Given I, a set of pairs of type (p, q), each corresponding to a potential insertion in $\overrightarrow{S}$, $\overrightarrow{S} \underset{I}{\Longrightarrow} \overrightarrow{S}'$ means that $\overrightarrow{S}'$ could be produced by applying one insertion from I on $\overrightarrow{S}$: $\exists (p, q) \in I \mid \overrightarrow{S} \underset{(p,q)}{\Longrightarrow} \overrightarrow{S}'$. More generally the *zero or more derivation steps* is defined as $\overrightarrow{S} \overset{*}{\underset{I}{\Longrightarrow}} \overrightarrow{S}' \equiv \overrightarrow{S} = \overrightarrow{S}' \vee \left(\exists (p, q) \in I \mid \overrightarrow{S} \underset{(p,q)}{\Longrightarrow} \overrightarrow{S}'' \wedge \overrightarrow{S}'' \overset{*}{\underset{I \setminus \{(p,q)\}}{\Longrightarrow}} \overrightarrow{S}' \right)$. Note that I may contain tuples that do not correspond to a possible insertion in $\overrightarrow{S}$ but instead to a possible insertion in a super-sequence of $\overrightarrow{S}$. Also note that several sequences of insertions in I may lead to an identical super-sequence.

3.2 The sequence domain

The formal definition of a sequence domain is given next.

▶ **Definition 1.** *The domain of a Sequence Variable Sq is represented as* $\langle \overrightarrow{S}, I, P, E \rangle$*, with* $\overrightarrow{S}$ *a sequence of nodes forming a partial tour, insertion points* $I \subseteq \mathcal{X} \times \mathcal{X}$ *and two subsets of nodes* $P, E \subseteq \mathcal{X}$ *for nodes that can possibly be inserted and nodes that are excluded from the sequence, respectively. The domain of Sq, also noted* $D(Sq)$*, is defined as* $\langle \overrightarrow{S}, I, P, E \rangle \equiv \left\{ \overrightarrow{S}' \in \overrightarrow{\mathcal{P}}(P \cup S) \mid \overrightarrow{S} \overset{*}{\underset{I}{\Longrightarrow}} \overrightarrow{S}' \right\}$ *and capture all the possible valid derivations from the partial tour* $\overrightarrow{S}$ *using insertions of* I*.*

At its initialization the Sequence Variable is composed of a partial tour of two nodes $\alpha \cdot \omega$ for the beginning α and ending ω of the tour and no insertions are allowed after ω to ensure ω remains the last visited node. P is thus equal to $\mathcal{X} \setminus \{\alpha, \omega\}$, $E = \phi$ and the set of insertions is $I = \{(p, q) \in P \times \mathcal{X} \mid p \neq \omega\}$. Imposing a first and last node in the sequence conveniently allows the modeling of problems where the route taken by a vehicle needs to end at its starting point (α lies at the same location as ω) or at another location ($\alpha \neq \omega$) and prevent the API to deal with the special case of empty sequences that require the introduction of a dummy symbol as in [26]. This use of beginning and ending nodes is also used in CP Optimizer and is described as *sinks* in their API [18].

Different forms of consistency could be imagined ensuring for instance that, for all pairs of nodes $(p, q) \in I$, both p and q are reachable from α by using the arcs defined in I and $\overrightarrow{S}$. Checking this consistency would relax the constraint that sequences only visit each node at most once.

In practice we use an even weaker form of consistency on the sequence domain, that is cheap to compute and is captured by the following invariant:

$$S \cup P \cup E = \mathcal{X} \wedge S \cap P = S \cap E = P \cap E = \phi \tag{1}$$

$$\forall (p, q) \in I : p \notin E \wedge q \in P \tag{2}$$

$$\forall q \in P : \exists p \in S \cup P \mid (p, q) \in I \tag{3}$$

(1) Nodes in the partial sequence S, in the possible set P and the set of excluded E form a partition of $\mathcal{X}$; (2) valid insertions are constituted of possible nodes after non excluded nodes (thus not necessarily present in the partial sequence); any excluded node cannot be inserted in $\overrightarrow{S}$ and is not a valid predecessor; any possible element can be inserted after a node (3). This weak consistency can for instance not detect situations where all the edges in I are disconnected from the partial sequence S, forming a disconnected cluster of nodes whose members should be excluded.

3.3 Implementation and data-structures

The implementation of the domain $\langle \overrightarrow{S}, I, P, E \rangle$ should be reversible for trail-based solver such as MiniCP [21] and most of the update and domain iteration operations should be as efficient as possible.

The set partitioning between the sets S, P, E is implemented using a single reversible sparse-sets data-structure as described in [10] ensuring removal and state restoration in constant time.

The insertion points set I is partitioned with one set $I^x = \{p \in (S \cup P) : (p, x) \in I\}$ for each node $x \in \mathcal{X}$ composed of the valid predecessors for node x. Those sets are implemented using reversible sparse-sets ensuring removal and state restoration in constant time. The lower-level consistency invariant expressed in terms of these data structures are given next in equations (4) to (7).

$$S \cup P \cup E = X \wedge S \cap P = S \cap E = P \cap E = \phi \tag{4}$$

$$p \in E \implies I^p = \phi \wedge \forall x : p \notin I^x \tag{5}$$

$$p \in S \implies I^p = \phi \tag{6}$$

$$I^p = \phi \implies p \in S \vee p \in E \tag{7}$$

Through the use of the reversible sparse-set, (1) is directly equivalent to (4). (2) is respected through (5) ($\forall (p, q) \in I : p \notin E$) and through (4), (5), (6) and (7) ($q \in E \cup S \Leftrightarrow q \notin P \implies I^q = \phi \implies \forall (p_1, q_1) \in I : q_1 \neq q$). Finally (3) is retrieved by combining (4) to (7) ($I^x \neq \phi \implies x \notin (S \cup E) \iff x \in P$).

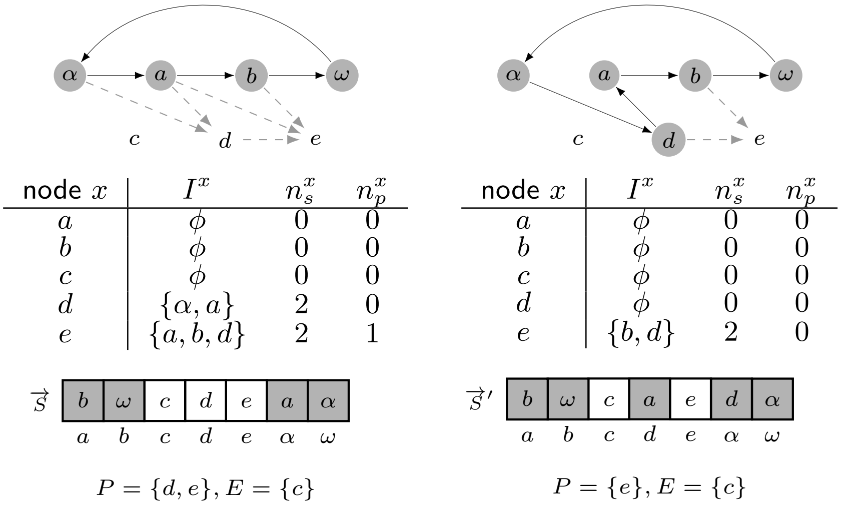
The internal partial sequence $\overrightarrow{S}$ of nodes is implemented using an array of reversible integers, as in [26]. This array stores the current successor of a node, and an element without successor points towards itself.

Additionally, the implementation maintains two reversible integers for every node $x \in \mathcal{X}$: n_s^x for the size of $I^x \cap S$ and n_p^x for the size of $I^x \cap P$. Those values are useful during the search to implement heuristics, for instance when searching the node $i \in P \mid n_s^i \leq n_s^j \; \forall j \in P$ having the least predecessors in the current ordering $\overrightarrow{S}$.

A domain representation example for a Sequence Variable is depicted in Figure 1.

Table 1 highlights the most important operations available on a Sequence Variable and their associated time complexity.

Any global constraint interested to be notified on domain modification of the Sequence Variable can do it using the three following hookup events:



node x	I^x	n_s^x	n_p^x
a	ϕ	0	0
b	ϕ	0	0
c	ϕ	0	0
d	$\{\alpha, a\}$	2	0
e	$\{a, b, d\}$	2	1

$\vec{S}$	a	b	c	d	e	α	ω
	b	ω	c	d	e	a	α

$P = \{d, e\}, E = \{c\}$

node x	I^x	n_s^x	n_p^x
a	ϕ	0	0
b	ϕ	0	0
c	ϕ	0	0
d	ϕ	0	0
e	$\{b, d\}$	2	0

$\vec{S}'$	a	b	c	d	e	α	ω
	b	ω	c	a	e	d	α

$P = \{e\}, E = \{c\}$

Figure 1 Representation of the implementation of a Sequence Variable. The left part shows a sequence ordering as well as possible insertions (dashed lines) for nodes $x \in P$. Below is a table showing the insertions for the nodes, the successor of the sequence (only relevant for nodes $\notin P \cup E$) and the split of nodes between S, P and E. The right part shows a modification $\vec{S} \underset{(\alpha,d)}{\Longrightarrow} \vec{S}'$ where node d has been inserted after node α, changing its status and removing insertion a from I^e due to some constraint.

Table 1 Operations on a Sequence Variable `Sq` with domain $\langle \vec{S}, I, P, E \rangle$.

Operation	Description	Complexity
`isBound(Sq)`	return true iff $\|P\| = 0$	$\Theta(1)$
`is{Member/Possible/Excluded}(Sq, x)`	return true iff $x \in \{S/P/E\}$	$\Theta(1)$
`get{Member/Possible/Excluded}(Sq)`	enumerate over $\{\vec{S}/P/E\}$	$\Theta(\|\{\vec{S}/P/E\}\|)$
`succ(Sq, x)`	return $q \mid x \to q$	$\Theta(1)$
`pred(Sq, x)`	return $p \mid p \to x$	$\Theta(1)$
`insert(Sq, p, x)`	insert x into Sq such that $p \to x$ holds	$\Theta(P)$
`exclude(Sq, x)`	exclude x from Sq	$\Theta(P)$
`nMemberInserts(Sq, x)`	return $n_s^x = \|I^x \cap S\|$	$\Theta(1)$
`nPossibleInserts(Sq, x)`	return $n_p^x = \|I^x \cap P\|$	$\Theta(1)$
`getMemberInserts(Sq, x)`	enumerate over $I^x \cap S$	$\Theta(\min(\|I^x\|, \|S\|))$
`getPossibleInserts(Sq, x)`	enumerate over $I^x \cap P$	$\Theta(\min(\|I^x\|, \|P\|))$
`canInsert(Sq, p, x)`	return true iff $p \in I^x$	$\Theta(1)$
`removeInsert(Sq, p, x)`	remove p from I^x	$\mathcal{O}(P)$

- The sequence is bound, that is the set of possible nodes is empty;
- A node has been inserted / excluded and this node is provided as a parameter of the event to allow incremental updates;
- The number of elements within a set I^x has changed, and the corresponding node is provided as a parameter of the event to allow incremental updates.

In the next section, we describe the filtering algorithms of some important constraints on the sequence variables.

4 Global constraints

This section defines and explains the filtering algorithm for some constraints on the Sequence variables. Some were already introduced in [26] but the filtering algorithms are adapted to reflect the removal of the required set. Some constraints reason over a list of values, written $[x]$ when x is a list, or a matrix of values, denoted by $[[x]]$.

Dependency

Despite the removal of the required set, one might still want to require a particular node to be inserted in one specific sequence. The `Dependency` constraint allows this. This constraint takes a list Dep of nodes as parameter that must all be member of the sequence or excluded from it. It is formally defined as

$$\texttt{Dependency}(Sq, [Dep]) \equiv \left\{\overrightarrow{S} \in D(Sq) \mid Dep \cap S \neq \phi \Leftrightarrow Dep \cap S = Dep\right\}. \tag{8}$$

Filtering. The filtering is triggered when a node $d \in Dep$ is either excluded or inserted into the sequence. If it is excluded, it excludes all other nodes $d' \in Dep \mid d \neq d'$. If it is inserted, it ensures that excluding any other node $d' \in Dep \mid d \neq d'$ results in a failure. The complexity of this filtering is $\mathcal{O}(|Dep|)$.

Disjoint

This constraint ensures that every node $x \in \mathcal{X}$ can be inserted once and only once across all sequences $Sq \in SQ$:

$$\begin{aligned}&\texttt{Disjoint}([SQ], \mathcal{X}) \equiv \\ &\forall Sq, Sq' \in [SQ], \forall x \in \mathcal{X}, Sq \neq Sq' \implies x \in Sq \implies x \notin Sq'\end{aligned} \tag{9}$$

Filtering. The filtering is triggered when a node $i \in \mathcal{X}$ is inserted in a Sequence $Sq \in SQ$. It excludes i from all other Sequences $Sq' \in SQ \mid Sq \neq Sq$. As the constraint can be notified of the value of i when an insertion occurs through the hookups events, we only need to iterate over SQ for checking the consistency. The complexity of the filtering when a single node is inserted is therefore $\mathcal{O}(|SQ|)$.

This constraint can optionally enforce that nodes must be inserted in exactly one of the sequences: $\forall x \in \mathcal{X}\ \exists Sq \in SQ \mid x \in D(Sq)$. If this is the case, the constraint fails if a mandatory node is excluded from all sequences .

Precedence

For some applications, visiting a set of nodes in a specific order is important and those nodes must all be inserted or not at all. This constraint is similar to the `Dependence` constraint but is done over an ordered set $\overrightarrow{D}$ of nodes, ensuring that the order in the set appears within the sequence if some node $n \in \overrightarrow{D}$ belongs to the Sequence. It is formally defined as

$$\texttt{Precedence}(Sq, \overrightarrow{D}) \equiv \left\{\overrightarrow{S} \in D(Sq) \mid \overrightarrow{S} \cap \overrightarrow{D} \neq \phi \implies \forall i, j \in \overrightarrow{D} : i \overset{\overrightarrow{D}}{\prec} j \implies i \overset{\overrightarrow{S}}{\prec} j\right\} \tag{10}$$

Filtering. The filtering is triggered whenever a node is inserted into the sequence or excluded from it. It iterates over the nodes in $\overrightarrow{D}$ and ensures that they appear in the same order in $\overrightarrow{S}$ if they belong to $D(Sq)$. It then removes the insertions that would prevent the order from being respected:

$$\forall i,j \in \overrightarrow{D}\ \forall p \in \overrightarrow{S} \mid \left(i \overset{\overrightarrow{D}}{\prec} j \wedge p \overset{\overrightarrow{S}}{\prec} i \implies p \notin I^j \right) \wedge \left(i \overset{\overrightarrow{D}}{\prec} j \wedge j \overset{\overrightarrow{S}}{\prec} p \implies p \notin I^i \right) \tag{11}$$

Furthermore, if some node in $\overrightarrow{D}$ is excluded from the Sequence, all nodes from $\overrightarrow{D}$ are excluded as well. The time complexity is $\mathcal{O}(|D| \cdot |S|)$ as the constraint considers the insertions $I^x \cap S$ for every node $x \in \overrightarrow{D}$.

Transition Times

The `TransitionTimes` constraint is used for problems where nodes are related to a time window and where transitions from one node to another take a certain duration specified in a distance matrix. This constraint ensures that the order defined by the successor set $\overrightarrow{S}$ is feasible: all nodes in S must be visited within their time window.

More formally, each node $x \in \mathcal{X}$ is attached to a time window variable $[start_x]$ and a duration $duration_x$. A matrix $trans \in R^{n \times n}$ with n nodes defines the transition times between elements and satisfies the triangular inequality. The sum of transition times when following the path described by the sequence is defined by a variable $transitionTime$. The constraint is defined as

$$\texttt{TransitionTimes}(Sq, [start], [duration], [[trans]], transitionTime) \equiv$$
$$\left\{ \overrightarrow{S} \in D(Sq) \middle| \begin{array}{c} \forall i,j \in \overrightarrow{S}, i \overset{\overrightarrow{S}}{\prec} j \implies start_i + duration_i + trans_{i,j} \leq start_j \\ transitionTime = \sum_{i,j \in \overrightarrow{S} \mid i \rightarrow j} trans_{i,j} \end{array} \right\} \tag{12}$$

We consider that waiting at a given node (i.e. reaching it before its time window without beginning the task related to it) is possible, which is why (12) uses inequalities.

Filtering. The pseudo code for the filtering is shown in Algorithm 1. It first ensures that a Sequence respects its time windows: an iteration over $\overrightarrow{S}$ is done, updating the bounds for $start_i$ $\forall i \in \overrightarrow{S}$ (line 2). No time window update is done for nodes $\notin \overrightarrow{S}$. Afterwards, it computes the current length of the sequence as the sum of transitions between elements in S and uses it to update the bounds of $transitionTime$ (line 4). Only the lower bound is updated, as we could exclude all remaining nodes in P and still get a valid solution. Then, the algorithm starts removing invalid insertions. An insertion $p \in I^x \cap S$ for a node $x \in P$ is invalid if reaching x through p would violate its time window (line 8), prevent the current successor $q \mid p \rightarrow q$ of p to be reached within its own time window (line 13) or exceed the maximum traveled distance (line 17). Line 12 uses a max because reaching a node before its time window is possible: if $reaching_x < \min(start_x)$, the departure occurs at $\min(start_x)$, otherwise it happens at $reaching_x$.

The time complexity of this filtering is $\mathcal{O}(|P| \cdot |S|)$. However, the effective complexity is slightly lower as $I^x \cap S$ is retrieved in $\Theta(\min(|S|, |I^x|))$. A similar pruning can also be defined for predecessors $p \in I^x \cap P$ of $x \in P$, ensuring that doing the transition from p to x would not exceed $start_x$. Because we do not reason over a set of Required nodes as in [26], we do not need to ensure that a valid transition exists among those Required nodes, removing the NP-complete problem of checking such transition.

Algorithm 1 `TransitionTimes`$(Sq = \langle \overrightarrow{S}, I, P, E\rangle, [start], [duration], [[trans]], transitionTime)$ filtering.

```
for $i \in \overrightarrow{S}$ do
    update time windows $start_i$
$length \leftarrow$ current distance of the sequence
$\min(transitionTime) \leftarrow length$
for $x \in P$ do
    for $p \in I^x \cap \overrightarrow{S}$ do
        $reaching_x \leftarrow \min(start_p) + duration_p + trans_{p,x}$
        if $reaching_x > \max(start_x)$ then
            remove $p$ from $I^x$
        else
            $q \leftarrow succ(Sq, p)$
            $reaching_q \leftarrow \max(reaching_x, \min(start_x)) + duration_x + trans_{x,q}$
            if $reaching_q > \max(start_q)$ then
                remove $p$ from $I^x$
            else
                $detour \leftarrow trans_{p,x} + trans_{x,q} - trans_{p,q}$
                if $detour + length > \max(transitionTime)$ then
                    remove $p$ from $I^x$
```

Cumulative

Common variations of VRP include pickup and delivery occurring at nodes, consuming a certain amount of load available in a vehicle. By analogy to scheduling problems, this constraint is called the `Cumulative` constraint: when providing a set of activity consuming a certain load $load_x$, it ensures that a maximum capacity is never exceeded and filters insertions that would exceed the available capacity. As our filtering is close to the one presented in [26] but more enhanced, we will borrow their notation.

More specifically, let us define an *activity* i as a pair of nodes (s_i, e_i) for its start (pickup) and end (delivery), respectively. The set of all activities is written A. An activity $i \in A$ consumes a certain load $load_i$ during its execution and can be in one of three states with respect to a Sequence Variable: *fully inserted* if $s_i \in S \wedge e_i \in S$, *non-inserted* if $s_i \notin S \wedge e_i \notin S$, and *partially inserted* otherwise (the pickup or the delivery is inserted but not both). The `Cumulative` constraint with a maximum capacity C, with starts $start$ and corresponding ends end is defined as

$$\texttt{Cumulative}(Sq, [start], [end], [load], C) \equiv$$

$$\left\{ \overrightarrow{S} \in D(Sq) \mid \forall e \in \overrightarrow{S}, \sum_{i \in A \mid start_i \preceq e \preceq end_i} load_i \leq C \right\} \tag{13}$$

Checking. The checking consists of verifying that an optimistic load profile does not exceed the vehicle capacity. We introduce two sets of values that represent the accumulated capacity at each node visited in the order of the partial sequence instead of one as in [26]. This allows computing a more realistic load profile and filtering more insertion points. Those two sets are denoted $C^b = \{C^b_x \; \forall x \in \overrightarrow{S}\}$ for the accumulated capacity just *before* visiting a given node and $C^a = \{C^a_x \; \forall x \in \overrightarrow{S}\}$ for the accumulated capacity just *after* leaving a given node.

The computing of those values is presented in Algorithm 2. It looks at the positions of the start and end of fully inserted activities, and increases C^a from the start until the node before the end node (line 7). For C^b, it is increased from the node after the start until the end node, included (line 5). When encountering a partially inserted activity i, our optimistic load profile considers that a sequence can be formed where $start_i \to end_i$ and thus only increases the value of C^a (line 10) or C^b (line 12) at one node. This setting for the load profile implies $C^b_s < C^a_s$ for every inserted start s and $C^b_e > C^a_e$ for every inserted end e. An example load profile is shown in Figure 2. Note that we do not necessarily have $C^a_i = C^b_j \mid i \to j$, as illustrated in Figure 3.

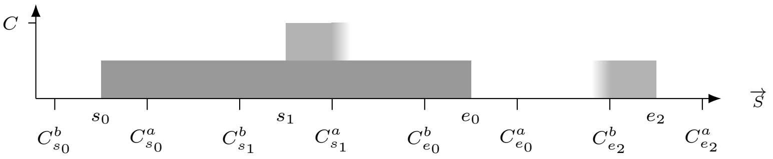


Figure 2 Load profile for the `Cumulative` constraint with $C = 2$ and $\vec{S} = \{\alpha, s_0, s_1, e_0, e_2, \omega\}$. Each activity has a load of 1, activity 0 (s_0, e_0) is fully inserted (dark gray) and both activity 1 and 2 are partially inserted (light gray). e_1 is considered to be inserted right after s_1, whose load only affects $C^a_{s_1}$. For activity 2, s_2 is consider to be inserted before e_2, affecting the value $C^b_{e_2}$.

Algorithm 2 `LoadProfile`$(Sq, start, end, load, C)$ computation.

```
Input  : $start, end, load$: start, end and load of activities, $C$: capacity,
         $Sq = \langle \vec{S}, I, P, E \rangle$: Sequence Variable.
Output : $C^b, C^a$: capacity before arriving at a node and after leaving a node,
         respectively.
1  $C^b, C^a \leftarrow 0$
2  for $i \mid start_i \in S \vee end_i \in S$ do
3      if $start_i \in S \wedge end_i \in S$ then
4          for $x \in \vec{S} \mid start_i \prec x \preceq end_i$ do
5              $C^b_x \leftarrow C^b_x + load_i$
6          for $x \in \vec{S} \mid start_i \preceq x \prec end_i$ do
7              $C^a_x \leftarrow C^a_x + load_i$
8      else
9          if $start_i \in S$ then
10             $C^a_{start_i} \leftarrow C^a_{start_i} + load_i$
11         else
12             $C^b_{end_i} \leftarrow C^b_{end_i} + load_i$
13 return $C^b, C^a$
```

Filtering. The filtering is triggered whenever new elements are inserted into the sequence. It uses the load profile computed during the checking to filter two cases: the partially inserted activities first and the non-inserted activities afterwards.

The partially inserted activities are considered first: we remove the insertions points for their non-inserted node that would cause the maximum capacity to be exceeded. A filtering example for removing insertions for starts whose corresponding end is inserted is shown in Algorithm 3. We iterate over the sequence in backward order (line 9) and compute the capacity occurring at the node (line 6 and 10). As soon as the maximum capacity would be exceeded if the start was inserted there, we remove the corresponding insertions (line 7 and 11). We inspect both the capacity before arriving at a node (line 7) and when leaving it (line 11) to detect invalid insertions. A load profile example where such filtering is used is shown in Figure 3. It detects that the starts of partially inserted activities cannot be inserted everywhere in the sequence. This detection was not possible using the load profile from [26], illustrated in Figure 4: it only includes the capacity when leaving the node, which is always zero when no start is inserted. In this case, their algorithm produces an empty profile, which can be enhanced and more representative, as in Figure 3.

For the non-inserted activities, we use a similar pruning as [26]: we look at every possible insertions for the start of the activities and see if a matching end can be found. Start positions that cannot be closed and end positions for which no corresponding start can be found are removed. The time complexity is dominated by the complexity to check all the activities, which is $\mathcal{O}(|S| \cdot |A|)$.

Algorithm 3 `CumulFiltering`$(Sq, start, end, load, C, C^b, C^a)$ for partially inserted activities with end inserted.

```
   Input : $Sq = \langle \overrightarrow{S}, I, P, E \rangle$: Sequence Variable, $start, end, load$: start, end and load of
           activities, $C$: capacity, $C^b$: minimum capacity before reaching a node, $C^a$
           minimum capacity after leaving a node.
 1 for $i \mid start_i \notin S \wedge end_i \in S$ do
 2     $current \leftarrow (x \in \overrightarrow{S} \mid x \rightarrow end_i)$
 3     if $C^a_{current} + load_i > C$ then
 4         return failure
 5     while $current \neq \alpha$ do
 6         if $C^b_{current} + load_i > C$ then
 7             remove all nodes $x \in \overrightarrow{S} \mid x \prec current$ from $I^{start_i}$
 8             break
 9         $current \leftarrow (x \in \overrightarrow{S} \mid x \rightarrow current)$
10         if $C^a_{current} + load_i > C$ then
11             remove all nodes $x \in \overrightarrow{S} \mid x \preceq current$ from $I^{start_i}$
12             break
13 return success
```

5 Experimental Results

The experiments reported in this section were conducted using two Intel(R) Xeon(R) CPU E5-2687W with 128GB of RAM. The Sequence Variable was implemented in MiniCP solver [21]. The source code is available for the readers in this anonymous repository [1], or by contacting the authors directly.

5.1 Dial-A-Ride Problem

We consider the problem described in [9, 15] and borrow the notations from [15]. This problem consists of m vehicles that must process n requests. Each request has a maximum ride time L, the vehicles have a maximum route duration D and the planning time is defined by a value T, representing the time at which the vehicles must be returned back to their

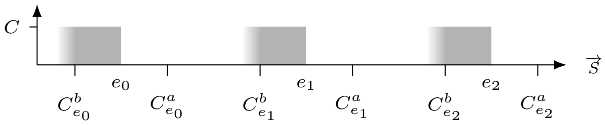


Figure 3 Load profile for a Sequence Variable with $\overrightarrow{S} = \{\alpha, e_0, e_1, e_2, \omega\}$, where only ends are inserted. Thanks to the computation of C^b in addition to C^a and the use of Algorithm 3, we can remove invalid insertions when considering the starts whose corresponding ends are inserted. This is the case for the start of activity 2 s_2 which cannot be inserted right after e_0, as the capacity before arriving at node e_1 would be exceeded. This case would have not been detected using only C^a, resulting in a load profile similar to Figure 4.

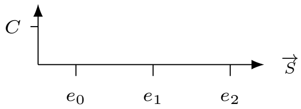


Figure 4 Load profile from [26] for a Sequence Variable with $\overrightarrow{S} = \{e_0, e_1, e_2\}$, where only ends are inserted. Because the accumulated capacity at each node is computed only when leaving a node, partially inserted activities might not contribute to the profile. In this case, it does not allow to detect that trying to insert s_2 after e_0 is invalid.

origin. Each request i consists of an associated load and 2 nodes: a pickup $pickup_i$ and delivery $drop_i$ that must be visited one before the other. Each node i has an associated service duration $d_j \geq 0$ and belongs to one of two categories: *non-critical nodes*, having a time window $[0, T]$ and *critical nodes* having a tighter time window $[s_i, e_i]$ where $s_i \neq 0 \vee e_i \neq T$. Each activity is composed of exactly one critical vertex and one non-critical vertex and the set of all critical vertices is CV. The nodes define a complete graph: there is always a transition from one node to another.

A solution for the DARP consists of finding a route such that all vehicles begin and end at the depot; all requests are serviced; the maximum capacity of a vehicle is never exceeded; the pickup and corresponding delivery of a request are serviced by the same vehicle; for all requests i the difference between the arrival time at a $drop_i$ and the departure from $pickup_i$ never exceeds L; each node is visited within its time window. The objective consists of minimizing the routing cost: the sum of traveled distance by each vehicle.

This problem can be modeled easily by introducing one Sequence Variable per vehicle. Only a few constraints are required, the most important ones being a `Disjoint` constraint to ensure that nodes are visited once, a `TransitionTimes` to prevent visits of nodes outside of their time window and a `Cumulative` to respect the maximum capacity of each vehicle. We compare our results with [15], a state-of-the art approach for DARP and we use a similar branching strategy, shown in Algorithm 4.

We begin by computing the number of insertions for every request (line 7) as the product between the insertions for its critical node and for its non-critical node. This can be retrieved in constant time for one node x through n^x_s, introduced in section 3.3. We then select the request having the least possible insertions (line 8) and branch on every pair of insertions for its vertices (line 13). Those branching decisions are ordered by increasing value of a heuristic h for inserting a node $x \in P$ between nodes $i, j \in \overrightarrow{S} \mid i \rightarrow j$. This heuristic is defined in equations (14)-(16), and is similar to [15].

$$h(x,i,j) = \alpha \cdot costIncrease(x,i,j) - \beta \cdot slack(x,i,j) \tag{14}$$
$$costIncrease(x,i,j) = dist_{i,x} + dist_{x,j} - dist_{i,j} \tag{15}$$
$$slack(x,i,j) = \max(time_j) - \min(time_i) - dist_{i,x} - dist_{x,j} \tag{16}$$

Where $time_x$ is an integer variable denoting the serving time of node x and $dist_{i,j}$ the distance between nodes i and j. The values for α and β were kept from [15] and are set to 80 and 1, respectively. We also use Large Neighborhood Search with First Feasible Probabilistic

Algorithm 4 Branching for DARP with a set SQ of Sequence Variables.

```
if *no unassigned requests left* then
    return solution
for $r \in$ unassigned request do
    $nInsert_r \leftarrow 0$
    $cv_r, ncv_r \leftarrow$ critical node and non-critical node from request $r$
    for $S \in SQ$ do
        $nInsert_r \leftarrow nInsert_r + nMemberInserts(S, cv_r) \cdot nMemberInserts(S, ncv_r)$
$r \leftarrow$ argmin $\{nInsert_r \mid \forall r \in$ unassigned request$\}$
$branching \leftarrow \{\}$
for $S \in SQ$ do
    for $p_{cv} \in getMemberInserts(S, cv_r)$ do
        for $p_{ncv} \in getMemberInserts(S, ncv_r)$ do
            $branching \leftarrow branching + (insert(S, cv_r, p_{cv}), insert(S, ncv_r, p_{ncv}))$
sort $branching$ by increasing order of heuristic
return $branching$
```

Acceptance from [15]. For a fair comparison, we have implemented the COMET source code provided by the authors of [15] in Java. Although not able to run it in COMET, we could obtain solution quality similar to the ones reported in [15] with the translated source-code. It is worth mentioning that the code for [15] is not generic, but custom and optimized for this sole problem. The filtering of the insertions is done during the search procedure rather than relying on the generic constraints executed in the fix-point of the solver.

We first compare the number of failures and solutions found using the exact search described in [15] without LNS with the Sequence Variable implementation. This comparison was made on a small instance with 2 vehicles and 20 requests and the corresponding results are reported in Table 2. We observe that the search from [15] finds all solutions to the instance in less time compared to the Sequence approach, which is ≈ 1.34 times slower. However, the number of failures is halved using Sequence Variables, resulting in a doubled ratio of solutions found per failure. This means that the approach performs better at removing invalid candidates to insert into the routes, although its filtering is slower.

Table 2 Statistics for finding all solutions on an instance with $m = 2$ vehicles and $n = 20$ requests, without using LNS. Choices refers to the number of branching decisions created during the search. Best result for each metric are shown in bold.

Statistic	Tree Search [15]	Tree Search with **Sequence**
Time [s]	**974.545**	1307.447
Choices	153 864 380	**120 593 739**
Failures	70 033 356	**35 751 093**
Solutions	**66 700 800**	**66 700 800**
Failures / choices	0.455	**0.296**
Solutions / choices	0.434	**0.553**
Solutions / failure	0.952	**1.866**

The next experiment compares the different approaches against instances with more requests and vehicles. It also includes a baseline comparison with a CP Optimizer model described in [26]. The solutions found are reported in Table 3 when an initial solution was provided. From the results, we see that our Sequence Variable does obtain results competitive with the approach from [15] with a slight advantage on smaller instances but not on larger ones. For some instances such as the one with $m = 8$ vehicles and $n = 108$ requests, finding a feasible solution is hard. This is why the Sequence Variables from CP Optimizer using a black-box search that is not specific to this problem cannot always find a feasible solution. However, even when providing an initial solution, CP Optimizer sometimes fails to improve it, whereas our approach is able to get even better results by using it.

Table 3 Comparison between our LNS-FFPA implementation from [15] (LNS-FFPA), our Sequence Variable implementation (Sequence) and the model using Sequence variables from CPoptimizer (CPO). 10 runs per solver were done on each instance, the best results are shown in bold. The left graph was produced when no initial solution was given, and the right when it was provided. Time-outs or no improving solution found are indicated by “t/o”.

15 minutes run - no initial solution provided							
class a		**LNS-FFPA**		**Sequence**		**CPO**	
m	n	Mean	Best	Mean	Best	Mean	Best
3	24	191.59	191.40	**190.99**	**190.79**	198.19	198.19
4	36	**291.71**	**291.71**	294.48	292.75	313.33	313.33
5	48	308.26	305.98	**308.02**	**305.48**	t/o	t/o
6	72	531.59	522.00	**527.80**	**518.94**	t/o	t/o
7	72	553.26	546.63	**551.45**	**544.64**	t/o	t/o
8	108	**741.37**	**719.24**	780.45	758.63	t/o	t/o
9	96	625.09	616.47	**622.86**	**612.74**	t/o	t/o
10	144	**949.72**	**922.32**	1005.12	951.33	t/o	t/o
11	120	**696.33**	**683.64**	715.10	692.71	t/o	t/o
13	144	**878.10**	**863.15**	913.60	899.12	t/o	t/o
Avg.		**576.70**	**566.25**	590.99	576.71	t/o	t/o

15 minutes run - initial solution provided							
class a		**LNS-FFPA**		**Sequence**		**CPO**	
m	n	Mean	Best	Mean	Best	Mean	Best
3	24	191.76	191.40	**190.89**	**190.21**	196.11	196.00
4	36	**291.71**	**291.71**	294.72	292.72	318.97	318.97
5	48	308.95	306.97	**307.09**	**304.38**	327.37	327.00
6	72	532.55	524.97	**531.84**	**519.76**	579.79	579.77
7	72	**554.57**	550.42	554.65	**548.72**	614.02	614.00
8	108	**752.29**	**742.08**	794.86	755.00	924.04	923.86
9	96	**622.19**	614.65	625.68	**611.15**	740.26	740.26
10	144	**950.16**	**929.31**	1011.42	962.21	t/o	t/o
11	120	**699.32**	**687.99**	718.58	709.49	861.74	861.73
13	144	**878.33**	**864.81**	901.71	874.56	1042.82	1042.82
Avg.		**578.18**	**570.43**	593.14	576.82	t/o	t/o

5.2 Patient Transportation Problem

This problem, described in [5], is an extension of the Dial-A-Ride problem introduced in Section 5.1 with a few additional constraints. It considers the transport of patients to a hospital (described as one activity) and possibly back to a given location (another activity) by using a limited number of vehicles. The trip to the hospital must therefore always occur before the return trip and some patients can only be transported in a particular type of vehicle (patients in wheelchairs for instance). The objective consists of maximizing the number of transported patients.

We introduce one Sequence Variable per vehicle. We then use a `Cumulative` constraint to ensure that a vehicle never exceeds its maximum capacity and serves each activity as well as a `Precedence` constraint to guarantee that the trip to the hospital occurs before the transportation back home. As activities must be serviced within a specific time window, we use a `TransitionTimes` constraint and finally a `Disjoint` constraint to ensure that the patients are serviced at most once. For cases where a particular patient i can only be transported in a given type of vehicle t, we simply exclude all nodes n related to i from Sequence Variables whose related vehicle type is different from t. Our search and LNS uses works similarly to the one from Section 5.1, by inserting all nodes related to a patient (for their forward and possibly backward trip) before trying to serve another patient.

The comparison between our model and the results from [5] for the biggest available instances are reported in Table 4. We have used the same time-out as the one reported in their paper (30 minutes) and run their model on our setup, finding better solutions than the ones they reported. We observe that we are able to improve the number of serviced patients on the most difficult instances by using Sequence Variables.

Table 4 Experimental results for the Patient Transportation Problem. $|H|$, $|V|$, $|R|$ are the number of hospitals, vehicles and requests, respectively. The objective is the number of patients serviced (Sol). SCHED+MSS refers to the best model from [5] while our own model is denoted as Sequence. Best results are shown in bold, the time-out was set to 30 minutes.

Instances					SCHED+MSS	Sequence
Difficulty	Name	$\|H\|$	$\|V\|$	$\|R\|$	Sol	Sol
Easy	RAND-E-8	32	12	128	**128**	**128**
Easy	RAND-E-9	36	14	144	**144**	143
Easy	RAND-E-10	40	16	160	**158**	156
Medium	RAND-M-8	64	8	128	89	**91**
Medium	RAND-M-9	72	8	144	89	**93**
Medium	RAND-M-10	80	9	160	109	**113**
Hard	RAND-H-8	128	8	128	77	**87**
Hard	RAND-H-9	144	8	144	78	**84**
Hard	RAND-H-10	160	8	160	76	**84**

5.3 Traveling Salesman Problem With Time Windows

TSPTW is a variant of the Traveling Salesman Problem (TSP) where all the customers must be visited within given time windows. Even finding a feasible solution was proved NP-complete [25]. As only one vehicle is available, the problem is modeled with a single Sequence Variable, the `TransitionTimes` constraint as well as a `Disjoint` constraint on the variable with the option that all nodes must be inserted.

We use LNS to find better solutions over time. The relaxation used by LNS starts from an initial solution and consists of removing a set C of n consecutive nodes from the solution after a given node i. Those nodes are then only allowed to be inserted after node i or after another node in C. To achieve this, we remove the insertions $(p, q) \mid (p \neq i \vee p \notin C \vee q \notin C)$ from the sets of insertions I. Nodes not belonging to C are ordered according to their previous best found ordering.

Our LNS, described in Algorithm 5 uses the same structure as the one from [15]. It starts from an initial solution *initSol* and relaxes an increasing number of nodes $n = i + j$ (line 7) from it. This process is done *numIter* times before increasing the number of relaxed nodes. $minSize, maxSize$ and *range* provide bounds for the number of nodes that needs to be relaxed. During our experiments, we have set $minSize = 10$, $maxSize =$ number of nodes in the problem, $range = 5$ and $numIter = 300$. The branching procedure at line 8 uses a similar branching as the one from the DARP: the non-inserted node x with the least number of member insertions $n_s^x = |I^x \cap S|$ is selected and branched on according to a heuristic that is the same as equation (14).

We tested the model on three sets of instances from [20], referred to as *OhlmannThomas*, *AFG* and *GendreauDumasExtended* and adapted from [22, 11] for the first set, [2] for the second set and from [13, 11] for the third set. The number of nodes in those instances varies from 20 to 232. The LNS was initialized from the best known solutions reported on [20]. The improved solutions were tested using the checker from [20] to ensure their feasibility as well as their cost.

A set of 20 of the 25 instances from the OhlmannThomas set could be improved, 10 of the 50 instances from the AFG set and 2 of the 130 instances from the GendreauDumasExtended set. The new objective solutions as well as the solving time to reach them are reported in Table 5. From our experiments we observe that we converge to a new solution sometimes rapidly (6 new best solutions are reached in less than 5 seconds and not improved afterwards) whereas some instances benefit more from the increasing number of nodes relaxed in the LNS and are still improved after a longer period of time.

Algorithm 5 LNS($Sq = \langle \overrightarrow{S}, I, P, E \rangle, initSol, minSize, maxSize, range, numIter, dist, timeLimit$).

```
$bestSol \leftarrow initSol$
for $i \in \{minSize \ldots (maxSize - range)\}$ do
    if $i$ = $maxSize$-$range$ then
        $i \leftarrow minSize$
    for $j \in \{0 \ldots range - 1\}$ do
        for $k \in \{1 \ldots numIter\}$ do
            relax(i + j) consecutive nodes from $bestSol$
            $sol \leftarrow optimize(dist)$
            if the solution has been improved then
                $bestSol \leftarrow sol$
            if $timeLimit$ is reached then
                return $bestSol$
```

Table 5 Improved routing cost values found for the TSPTW instances. Previous best objective values (Previous) are retrieved from [20]. New best objective values discovered are indicated (New) as well as the time to reach them. The time-out was set to 5 minutes. The Table on the left shows the values for the OhlmannThomas instances, the top right for the AFG instances and the bottom right for the GendreauDumasExtended.

Instance	Previous	New	Time [s]
n150w120.001	735	734	0.50
n150w120.002	683	679	290.86
n150w120.003	748	747	41.45
n150w120.005	692	689	7.84
n150w140.001	767	762	134.96
n150w140.002	757	755	34.28
n150w140.003	620	613	64.28
n150w140.004	677	676	12.94
n150w140.005	665	663	167.10
n150w160.001	708	706	2.64
n150w160.002	712	711	162.35
n150w160.003	610	608	0.49
n200w120.001	801	799	194.56
n200w120.002	725	722	12.12
n200w120.003	885	880	51.44
n200w120.005	843	841	202.69
n200w140.001	837	834	35.23
n200w140.002	768	765	14.77
n200w140.003	764	758	298.95
n200w140.005	827	822	33.21

Instance	Previous	New	Time [s]
rbg132.2	8200	8194	37.76
rbg132	8470	8468	0.76
rbg201a	12 967	12 948	152.53
rbg233.2	14 549	14 523	24.20
rbg092a	7160	7158	2.70
rbg152.3	9797	9796	0.41
rbg193.2	12 167	12 159	242.54
rbg193	12 547	12 538	55.57
rbg233	15 031	14 994	264.70
rbg172a	10 961	10 956	113.83

Instance	Previous	New	Time [s]
n80w120.005	597	591	9.09
n100w160.005	587	586	19.59

6 Conclusions and future work

This paper introduced a simplified version of the Sequence domain introduced in [26] as a flexible and effective approach for modeling and solving VRP with CP. The filtering algorithms for constraints imposing time windows and vehicle capacity are described. Experimental results on three problems show that our models are competitive with existing sequence based approaches while being effective enough to discover new best solutions to a well-studied problem such as the TSPTW. Our proposed filtering algorithms are relatively simple and could most certainly be improved. We plan to enhance them and study the use of Sequence Variables in more vehicle routing problems, as well as scheduling problems.

References

1 MiniCP Sequences - Anonymous GitHub, September 2021. [Online; accessed 26. Feb. 2022]. URL: `https://anonymous.4open.science/r/minicp-sequences-5EE3/README.md`.

2 Norbert Ascheuer. Hamiltonian path problems in the on-line optimization of flexible manufacturing systems. Technical report, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 1996.

3 Bruno De Backer, Vincent Furnon, Paul Shaw, Philip Kilby, and Patrick Prosser. Solving vehicle routing problems using constraint programming and metaheuristics. *Journal of heuristics*, 6(4):501–523, 2000.

4 Russell Bent and Pascal Van Hentenryck. A two-stage hybrid local search for the vehicle routing problem with time windows. *Transportation Science*, 38(4):515–530, 2004.

5 Quentin Cappart, Charles Thomas, Pierre Schaus, and Louis-Martin Rousseau. A constraint programming approach for solving patient transportation problems. In John Hooker, editor, *Principles and Practice of Constraint Programming*, pages 490–506, Cham, 2018. Springer International Publishing.

6 IBM Knowledge Center. Interval variable sequencing in CP Optimizer, March 2021. [Online; accessed 13. Jan. 2022]. URL: `https://www.ibm.com/docs/en/icos/12.9.0?topic=concepts-interval-variable-sequencing-in-cp-optimizer`.

7 IBM Knowledge Center. Search API for scheduling in CP Optimizer, March 2021. [Online; accessed 13. Jan. 2022]. URL: `https://www.ibm.com/docs/en/icos/12.9.0?topic=c-search-api-scheduling-in-cp-optimizer#85`.

8 Jean-François Cordeau and Gilbert Laporte. The dial-a-ride problem: models and algorithms. *Annals of operations research*, 153(1):29–46, 2007.

9 Jean-François Cordeau and Gilbert Laporte. A tabu search heuristic for the static multi-vehicle dial-a-ride problem. *Transportation Research Part B: Methodological*, 37:579–594, July 2003. `doi:10.1016/S0191-2615(02)00045-0`.

10 Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-sets for domain implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, 2013.

11 Yvan Dumas, Jacques Desrosiers, Eric Gelinas, and Marius M Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations research*, 43(2):367–371, 1995.

12 Interreg Europe. Demand-responsive transport. `https://www.interregeurope.eu/sites/default/files/2021-12/Policy%20brief%20on%20demand%20responsive%20transport.pdf`, June 2018.

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

14 Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1:191–244, March 1997.

15 Siddhartha Jain and Pascal Van Hentenryck. Large neighborhood search for dial-a-ride problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 400–413. Springer, 2011.

16 Philippe Laborie and Jerome Rogerie. Reasoning with conditional time-intervals. In *FLAIRS conference*, pages 555–560, 2008.

17 Philippe Laborie, Jerome Rogerie, Paul Shaw, and Petr Vilím. Reasoning with conditional time-intervals. part ii: An algebraical model for resources. In *FLAIRS Conference*, 2009.

18 Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. Ibm ilog cp optimizer for scheduling. *Constraints*, 23(2):210–250, April 2018. `doi:10.1007/s10601-018-9281-x`.

19 Chang Liu, Dionne M. Aleman, and J. Christopher Beck. Modelling and solving the senior transportation problem. In Willem-Jan van Hoeve, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 412–428, Cham, 2018. Springer International Publishing.

20 Manuel López-Ibáñez. Instances for the TSPTW, September 2020. [Online; accessed 15. Feb. 2022]. URL: `https://lopez-ibanez.eu/tsptw-instances`.

21 L. Michel, P. Schaus, and P. Van Hentenryck. Minicp: a lightweight solver for constraint programming. *Mathematical Programming Computation*, 13(1):133–184, 2021. `doi:10.1007/s12532-020-00190-7`.

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

23 Laurent Perron and Vincent Furnon. Or-tools. URL: `https://developers.google.com/optimization/`.

24 Laurent Perron and Vincent Furnon. Or-tools sequence var. URL: `https://developers.google.com/optimization/reference/constraint_solver/constraint_solver/SequenceVar`.

25 Martin WP Savelsbergh. Local search in routing problems with time windows. *Annals of Operations research*, 4(1):285–305, 1985.

26 Charles Thomas, Roger Kameugne, and Pierre Schaus. Insertion sequence variables for hybrid routing and scheduling problems. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 457–474. Springer, 2020.

27 Paolo Toth and Daniele Vigo. *The Vehicle Routing Problem.* SIAM, 2002.

CSP Beyond Tractable Constraint Languages

Jan Dreier
Algorithms and Complexity Group, TU Wien, Austria

Sebastian Ordyniak
Algorithms and Complexity Group, University of Leeds, UK

Stefan Szeider
Algorithms and Complexity Group, TU Wien, Austria

Abstract

The constraint satisfaction problem (CSP) is among the most studied computational problems. While NP-hard, many tractable subproblems have been identified (Bulatov 2017, Zuk 2017). Backdoors, introduced by Williams, Gomes, and Selman (2003), gradually extend such a tractable class to all CSP instances of bounded distance to the class. Backdoor size provides a natural but rather crude distance measure between a CSP instance and a tractable class. Backdoor depth, introduced by Mählmann, Siebertz, and Vigny (2021) for SAT, is a more refined distance measure, which admits the parallel utilization of different backdoor variables. Bounded backdoor size implies bounded backdoor depth, but there are instances of constant backdoor depth and arbitrarily large backdoor size. Dreier, Ordyniak, and Szeider (2022) provided fixed-parameter algorithms for finding backdoors of small depth into the classes of Horn and Krom formulas.

In this paper, we consider backdoor depth for CSP. We consider backdoors w.r.t. tractable subproblems C_Γ of the CSP defined by a constraint language Γ, i.e., where all the constraints use relations from the language Γ. Building upon Dreier et al.'s game-theoretic approach and their notion of separator obstructions, we show that for any finite, tractable, semi-conservative constraint language Γ, the CSP is fixed-parameter tractable parameterized by the backdoor depth into C_Γ plus the domain size.

With backdoors of low depth, we reach classes of instances that require backdoors of arbitrary large size. Hence, our results strictly generalize several known results for CSP that are based on backdoor size.

2012 ACM Subject Classification Theory of computation → Fixed parameter tractability

Keywords and phrases CSP, backdoor depth, constraint language, tractable class, recursive backdoor

Digital Object Identifier 10.4230/LIPIcs.CP.2022.20

Funding *Sebastian Ordyniak*: Supported by the Engineering and Physical Sciences Research Council (EPSRC, project EP/V00252X/1).

1 Introduction

To face the NP-completeness of the Constraint Satisfaction Problem (CSP), much effort has been spent in identifying polynomial-time solvable subproblems [5]. Tractability can be reached by

1. restricting the *constraint language* in terms of limiting the relations allowed to be used in constraints (e.g., [3, 6, 10, 30, 33]),
2. restricting the *graphical structure* of how constraints and variables interact (e.g., [7, 21, 22]), or
3. restricting both language and structure with *hybrid restrictions* (e.g., [8, 9, 11]).

Some of the considered restrictions are *gradual* in the sense that they support an infinite chain of classes $\mathcal{C}_0 \subsetneq \mathcal{C}_1 \subsetneq \mathcal{C}_2 \subsetneq \dots$ of instances, where each $\mathcal{C}_i$ can be solved in polynomial time. When the order of polynomial bound on the solving time remains the same for all

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 20; pp. 20:1–20:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the $i = 0, 1, 2, \ldots$ one speaks about *fixed-parameter tractability* (FPT) [12, 13, 15, 26, 29]. Most structural restrictions like bounded treewidth or hypertree width are gradual by definition [22, 22]. In contrast, language restrictions tend to be categorical by definition, as either an instance belongs to a class $\mathcal{C}_\Gamma$ defined by a tractable constraint language Γ or it does not.

However, by means of (strong[1]) *backdoors* introduced by Williams, Gomes and Selman [31, 32], one can build a chain $\mathcal{C}_\Gamma = \mathcal{C}_0 \subsetneq \mathcal{C}_1 \subsetneq \mathcal{C}_2 \subsetneq \ldots$ on top of such a class defined by a language. A CSP instance belongs to $\mathcal{C}_i$ if there is a set of i variables, called a backdoor, such that all possible instantiations of these variables move the instance into the base class $\mathcal{C}_\Gamma$. The size of a smallest backdoor provides a *distance measure* between the considered CSP instance and the base class.

The size of a smallest backdoor is a fundamental but still rather crude distance measure. Samer and Szeider [28] therefore proposed *backdoor trees*, where one counts the number of leaves of a decision tree ranging over all the variables of a backdoor; Ordyniak et al. [27] obtained further fixed-parameter tractability results for backdoor trees. A backdoor of size k over a Boolean domain can yield backdoor trees between $k + 1$ and 2^k leaves, and so it is more efficient to minimize the number of leaves than the size. Very recently, in the context of SAT, Mählmann, Siebertz, and Vigny [23] proposed the concept of *backdoor depth*, which extend backdoor trees by adding nodes where the tree branches into connected components. The advantage of considering backdoors of small depth relies on the observation that if an instance decomposes into multiple components, then each component can be treated independently. This way, one is allowed to use in total an unbounded number of backdoor variables. However, as long as the *depth* of the extended decision tree is bounded, one can still utilize it for efficiently solving the instance. In the context of graphs, similar ideas are used in the study of tree-depth [24, 25] and elimination distance [4, 16].

The challenging algorithmic question is to find a backdoor of small depth into a fixed base class, if it exists. Mählmann et al. [23] gave an FPT algorithm for SAT with respect to the base class Null consisting of formulas without variables; any bounded-depth backdoor into that class must contain all the variables of the instance. Already for this simple base class, there are instances of bounded backdoor depth that cannot be efficiently solved by other known methods. Previously [14], we extended this FPT result to bounded-depth backdoors into the classes of Horn (CNF formulas where each clause contains at most one positive literal), dual Horn (each clause contains at most one negative literal), and Krom formulas (each clause contains at most two literals).

Contribution. In this paper, we provide the first positive algorithmic results for utilizing backdoors of bounded depth for CSP. Our main technical result covers all base classes $\mathcal{C}_\Gamma$ described by a finite semi-conservative constraint language Γ. As our main result, we show the following (a formal statement is Corollary 20).

> *For any finite, tractable, semi-conservative constraint language Γ, the CSP is fixed-parameter tractable parameterized by the smallest depth of a backdoor into $\mathcal{C}_\Gamma$ plus the domain size of the instance.*

Thus, we indeed have a chain $\mathcal{C}_\Gamma = \mathcal{C}_0 \subsetneq \mathcal{C}_1 \subsetneq \mathcal{C}_2 \subsetneq \ldots$ on top of any such class $\mathcal{C}_\Gamma$, where $\mathcal{C}_i$ contains instances with a backdoor of depth i, and where the order of the polynomial-time algorithm for solving $\mathcal{C}_i$ is of the same order as the polynomial that bounds the solving time for $\mathcal{C}_\Gamma$.

[1] We focus only on strong backdoors and do not consider weak backdoors.

Backdoor depth can capture and exploit structure in CSP instances that is not captured by any other known method. In the following, we list here some known CSP parameters that admit fixed-parameter tractable CSP solving. For each of these parameters, there are CSP instances for which the parameter can be arbitrarily large, but where Γ-backdoor depth is bounded by a constant:

- backdoor size [20];
- backdoor depth for SAT [14, 23];
- backdoor size into heterogeneous and scattered base classes [19, 20];
- backdoor treewidth [18].

We closely follow the approach we introduced in recent work on SAT [14]. On a high level, we construct backdoors by simultaneously computing an upper bound in the form of an approximate backdoor and a lower bound, using so-called *obstructions*, i.e., parts of the instance that can be proven to be “far away” from the base class. As in our work on SAT, we use two types of obstructions:

1. a slightly modified version of the obstructions trees that have been introduced by Mählmann et al. [23] and
2. a new variant of separator obstructions that we introduced for SAT [14] to allow the handling of base classes that admit arbitrary long paths (in the incidence graph of a CNF formula).

This new variant of separator obstructions is tailor-made for CSP and base classes defined via finite constraint languages. It allows us to improve the algorithm’s efficiency, from a triple-exponential run-time dependency to a double-exponential run-time dependence on backdoor depth.

We present our results using the *game-theoretic framework* for backdoor depth that we introduced for SAT [14], which greatly simplifies the presentation of our algorithm.

Due to space constraints, we omit proofs of some technical claims, marked ($\star$).

2 Preliminaries

2.1 CSP

Let D be a set and n and n' be non-negative integers. An n-ary relation on D is a subset of D^n. For a tuple $t \in D^n$, we denote by $t[i]$, the i-th entry of t, where $1 \leq i \leq n$. For two tuples $t \in D^n$ and $t' \in D^{n'}$, we denote by $t \circ t'$, the concatenation of t and t'.

An instance of a *constraint satisfaction problem* (CSP) I is a triple $\langle V, D, C\rangle$, where V is a finite set of variables over a finite set (domain) D, and C is a set of constraints.We assume that D is given explicitly as a list of all domain values. A *constraint* $c \in C$ consists of a *scope*, denoted by $V(c)$, which is an ordered list of a subset of V, and a relation, denoted by $R(c)$, which is a $|V(c)|$-ary relation on D; $|V(c)|$ is the *arity* of c. To simplify notation, we sometimes treat ordered lists without repetitions, such as the scope of a constraint, like sets. For a variable $v \in V(c)$ and a tuple $t \in R(c)$, we denote by $t[v]$, the i-th entry of t, where i is the position of v in $V(c)$. For a CSP instance $I = \langle V, D, C\rangle$ we sometimes denote by $V(I)$, $D(I)$, and $C(I)$, its set of variables V, its domain D, and its set of constraints C, respectively. We usually assume, w.l.o.g, that each variable in $V(I)$ appears in the scope of at least one constraint in $C(I)$. The *size* $|I|$ of a CSP instance I is the sum of the sizes of its constraints, where the size of a constraint of arity a with t tuples and domain size δ is $at \log \delta$. A *solution* to a CSP instance I is a mapping $\tau : V \to D$ such that $\langle \tau(v_1), \ldots, \tau(v_{|V(c)|})\rangle \in R(c)$ for every $c \in C$ with $V(c) = \langle v_1, \ldots, v_{|V(c)|}\rangle$. A CSP instance is *satisfiable* if and only if it has at least one solution.

Let $V' \subseteq V$ and $\tau : V' \to D$. For a constraint $c \in C$, we denote by $c[\tau]$, the constraint whose scope is $V(c) \setminus V'$ and whose relation contains all $|V(c[\tau])|$-ary tuples t such that there is a $|V(c)|$-ary tuple $t' \in R(c)$ with $t[v] = t'[v]$ for every $v \in V(c[\tau])$ and $t'[v] = \tau(v)$ for every $v \in V' \cap V(c)$. We denote the assignment $\tau : \{x\} \to D$ with $\tau(x) = q$ simply by $x = q$.

A constraint $c \in C(I)$ of arity a is *tautological* if it contains all the $|D|^a$ possible tuples. Obviously, removing a tautological constraint from a CSP instance does not change its satisfiability. We denote by $I[\tau]$ the CSP instance with variables $V \setminus V'$, domain D, and constraints $C[\tau]$, where $C[\tau]$ contains all *non-tautological* constraints $c[\tau]$ for every $c \in C$. We would like to point out that the removal of tautological constraints is important in the context of backdoor depth as it makes the notion more powerful.

Let $\tau_1 : V_1 \to D$ and $\tau_2 : V_2 \to D$ be two assignments. We say that the two assignments are *compatible* if $\tau_1(v) = \tau_2(v)$ for every $v \in V_1 \cap V_2$. Moreover, if τ_1 and τ_2 are compatible, we denote by $\tau_1 \cup \tau_2$ the assignment $\tau : V_1 \cup V_2 \to D$ given by $\tau(v) = \tau_1(v)$ if $v \in V_1$ and $\tau(v) = \tau_2(v)$ if $v \in V_2$.

The *incidence graph* of a CSP instance I is the bipartite graph G_I whose vertices are the variables and constraints of I, and where a variable x and a constraint c are adjacent if and only if $x \in V(c)$. Via incidence graphs, graph theoretic concepts directly translate to CSP instances. For instance, we say that I is connected if G_I is connected, and I' is a *connected component* of I if $D(I') = D(I)$, and where $V(I')$ and $C(I')$ are maximal subsets of $V(I)$ and $C(I)$, respectively, such that G_I is connected. $\mathrm{Conn}(I)$ denotes the set of connected components of I. Occasionally, we will also consider the *primal graph* of a CSP instance I, which has as vertex set $V(I)$, and has pairs of variables adjacent if they appear together in the scope of a constraint.

A *constraint language* Γ over a domain D is a set of relations over D. $D(\Gamma)$ is the set of all the elements appearing in the relations in Γ. We denote by $\mathrm{arity}(\Gamma)$ the maximum arity of any relation in Γ. $\mathcal{C}_\Gamma$ denotes the class of CSP instances I with the property that for each $c \in C(I)$ we have $R(c) \in \Gamma$. $\mathrm{CSP}(\Gamma)$ refers to the CSP with instances restricted to $\mathcal{C}_\Gamma$. A constraint language Γ is *tractable* if $\mathrm{CSP}(\Gamma)$ can be solved in polynomial time, Γ is *linear-time tractable* if $\mathrm{CSP}(\Gamma)$ can be solved in linear time.

Γ is *semi-conservative* (or *1-conservative*) [1, 17], if for each $q \in D(\Gamma)$ one can express with Γ the unary constraint $x = q$; more precisely, there is a satisfiable instance I_q of $\mathrm{CSP}(\Gamma)$ and a variable $x \in V(I_q)$ such that for each solution τ of I_q we have $\tau(x) = q$. Semi-conservative constraint languages are very natural, as one would expect in any reasonable practical settings that the unary relations are present. Indeed, some authors (e.g., [10]) even define the CSP so that every variable can have its own set of domain values, making (semi-)conservativeness a built-in property.

A constraint language Γ is *closed under assignments* if for every constraint c with $R(c) \in \Gamma$ and every assignment τ, it holds that $R(c[\tau]) \in \Gamma$. For a constraint language Γ we denote by Γ^* the smallest constraint language that contains Γ and is closed under assignments.

▶ **Lemma 1** ([17]). *If a semi-conservative constraints language Γ is tractable, then Γ^* is also tractable.*

We would like to point out that the original definition of a backdoor by Williams et al. [31, 32] assumes the base class to be closed under assignments. Hence, it is natural to assume this property in the context of backdoor depth, directly or indirectly by means of semi-conservativeness of the considered language.

2.2 Backdoors

Backdoors are defined relative to some fixed *base class* $\mathcal{C}$ of instances of the problem under consideration (i.e., CSP), for which satisfiability and membership in $\mathcal{C}$ are polynomial-time decidable. In the context of CSP, we define a *$\mathcal{C}$-backdoor set* of a CSP instance I as a set $B \subseteq V(I)$ of variables such that $I[\tau] \in \mathcal{C}$ for every $\tau : B \to D(I)$. For a constraints language Γ, we usually denote the base class $\mathcal{C}_\Gamma$ by Γ itself. Thus, for example, instead of $\mathcal{C}_\Gamma$-backdoors, we talk of Γ-backdoors. If we know a $\mathcal{C}$-backdoor set B of I, we can reduce the satisfiability of I to the satisfiability of $|D(I)|^{|B|}$ CSP instances in $\mathcal{C}$. The challenging problem is to find a $\mathcal{C}$-backdoor set of a given instance that reduces the satisfiability problem to instances from $\mathcal{C}$.

3 Backdoor Depth

Component backdoor trees generalize backdoor trees as considered by Samer and Szeider [28] by allowing an additional type of nodes, *component nodes*, where the current instance is split into connected components. More precisely, let $\mathcal{C}$ be a class of CSP instances (called the base class) and I a CSP instance. A *component $\mathcal{C}$-backdoor tree for I* is a pair (T, φ), where T is a rooted tree and φ is a mapping that assigns each node t a CSP instance $\varphi(t)$ such that the following conditions are satisfied:

1. For the root r of T, we have $\varphi(r) = I$.
2. For each leaf ℓ of T, we have $\varphi(\ell) \in \mathcal{C}$.
3. For each non-leaf t of T, there are two possibilities:
   a. $D(I) = \{q_1, \dots, q_\delta\}$ and t has exactly δ children $t_1, \dots, t_\delta$ where for some variable $x \in V(\varphi(t))$ we have $\varphi(t_i) = \varphi(t)[x = q_i]$; in this case we call t a *variable node.*
   b. $\mathrm{Conn}(\varphi(t)) = \{I_1, \dots, I_k\}$ for $k \geq 2$ and t has exactly k children $t_1, \dots, t_k$ with $\varphi(t_i) = I_i$; in this case we call t a *component node.*

Thus, a backdoor tree as considered by Samer and Szeider [28] is just a component backdoor tree without component nodes. The *depth* of a backdoor is the largest number of variable nodes on any root-leaf path in the tree. The *$\mathcal{C}$-backdoor depth* $\mathrm{depth}_\mathcal{C}(I)$ of an instance I into a base class $\mathcal{C}$ is the smallest depth over all component $\mathcal{C}$-backdoor trees of I. If $\mathcal{C}$ is defined in terms of a constraint language Γ, we simply write $\mathrm{depth}_\Gamma(I)$.

Alternatively, we can define $\mathcal{C}$-backdoor depth recursively as follows:

$$\mathrm{depth}_\mathcal{C}(I) := \begin{cases} 0 & \text{if } I \in \mathcal{C}; \\ 1 + \min_{x \in V(I)} \max_{a \in D(I)} \mathrm{depth}_\mathcal{C}(I[x = a]) & \text{if } I \notin \mathcal{C} \text{ and } I \text{ is connected;} \\ \max_{I' \in \mathrm{Conn}(I)} \mathrm{depth}_\mathcal{C}(I') & \text{if } I \notin \mathcal{C} \text{ and } I \text{ is not connected.} \end{cases}$$

▶ **Lemma 2.** *ategory=ssrbd,normal]lemma Let Γ be a constraint language such that $\mathcal{C}_\Gamma$ can be solved in time $\mathcal{O}(n^c)$ for some constant $c \geq 1$ and input size n. Assume we are given a CSP instance I whose size is m, $\delta = |D(I)|$, and a component Γ-backdoor tree (T, φ) of I of depth d. Then, we can solve I in time $\mathcal{O}((\delta^d m)^c)$.*

Proof. We start by showing that $\sum_{\ell \in L(T)} |\varphi(\ell)| \leq \delta^d m$, where $L(T)$ denotes the set of leaves of T, using induction on d and m. The statement holds if $d = 0$ or $m \leq 1$. We show that it also holds for larger d and m. If the root is a variable node, then it has δ children $c_1, \dots, c_\delta$, and the subtree rooted at any of these children represents a component backdoor tree for the CSP instance $\varphi(c_i)$ of depth $d - 1$. Therefore, by the induction hypothesis, we obtain that $s_i = \sum_{\ell \in L(T_i)} |\varphi(\ell)| \leq \delta^{d-1} m$, for every subtree T_i rooted at c_i. Consequently, $\sum_{\ell \in L(T)} |\varphi(\ell)| = \sum_{1 \leq i \leq \delta} s_i \leq \delta\delta^{d-1} m = \delta^d m$, as required. If, on the other hand, the root is a component node, then its children, say $c_1, \dots, c_k$, are labeled with CSP instances of sizes $m_1 + \dots + m_k = m$. Therefore, for every subtree T_i of T rooted at c_i, we have that T_i is a component backdoor tree of depth d for $\varphi(c_i)$, which using the induction hypothesis implies that $\sum_{\ell \in L(T_i)} |\varphi(\ell)| \le \delta^d m_i$. Hence, we obtain $\sum_{\ell \in L(T)} |\varphi(\ell)| \le \delta^d m$ in total.

To solve the CSP instance I, we first solve all CSP instances associated with the leaves of T. Because, as shown above, their total size is at most $\delta^d m$, this can be achieved in time $\mathcal{O}((\delta^d m)^c)$, because $\mathcal{C}_\Gamma$ can be solved in time $\mathcal{O}(n^c)$ for some constant $c \ge 1$ and input size n. Let us call a leaf true/false if and only it is labeled by a satisfiable/unsatisfiable CSP instance, respectively. We now propagate the truth values upwards to the root, considering a component node as the logical *and* of its children, and the a variable node as the logical *or* of its children. I is satisfiable if and only if the root of T is true. We can carry out the propagation in time linear in the number of nodes of T, which is linear in the number of leaves of T, i.e., at most $\delta^d m$. ◀

4 Technical Overview

On a high level, the approach of our algorithm is similar to the approach we employed for SAT [14]. The critical difference lies in the exact definition of separator obstructions in Section 5, which we adapt to CSP and base classes defined via finite constraint languages. Apart from lifting the approach from SAT to CSP, our tailor-made separator obstructions also allow us to obtain a more efficient algorithm. As our first order of business, we state an equivalent formulation of backdoor depth using *connector-splitter games*, as we previously introduced for SAT [14], allowing us to greatly simplify the exposition of our algorithm.

▶ **Definition 3.** *Let Γ be a finite constraint language that is closed under assignments and let $I = \langle V, D, C \rangle$ be a CSP instance. We denote by* $\textsc{Game}(I, \Gamma)$ *the so-called* Γ-backdoor depth game *on I. The game is played between two players, the* connector *and the* splitter*. The* positions *of the game are CSP instances. At first, the connector chooses a connected component of I to be the* starting position *of the game. The game is over once a position in the base class $\mathcal{C}$ is reached. We call these positions the* winning positions *(of the splitter). In each* round *the game progresses from a* current position *J to a* next position *as follows.*

- *The splitter chooses a variable $v \in V(J)$.*
- *The connector chooses an assignment $\tau \colon \{v\} \to D$ and a connected component J' of $J[\tau]$. The* next position *is J'.*

In the (unusual) case that a position J contains no variables anymore but J is still not in $\mathcal{C}_\Gamma$, the splitter looses. For a position J, we denote by τ_J the assignment of all variables assigned up to position J.

The following observation follows easily from the definitions of the game and the fact that the (strategy) tree obtained by playing all possible plays of the connector against a given strategy for the splitter forms a component backdoor tree and vice versa. In particular, the splitter choosing a variable v at position J corresponds to a variable node and the subsequent choice of the connector for an assignment τ of v and a component of $J[\tau]$ corresponds to a component node (and a subsequent variable or leaf node) in a component backdoor tree.

▶ **Observation 4.** *The splitter has a strategy for the game* $\textsc{Game}(I, \Gamma)$ *to reach within at most d rounds a winning position if and only if I has a Γ-backdoor of depth at most d.*

Backdoor depth games mean that we no longer have to explicitly construct a backdoor. Instead, in Section 6, we compute winning strategies for the splitter, which appear to be easier to reason about. Such a strategy can then be automatically converted into a backdoor algorithm (Lemma 6).

We start by describing these so called *splitter-algorithms* and how they can be turned into an algorithm to compute backdoor depth. The algorithms will have some auxiliary internal state that they modify with each move. Formally, a *splitter-algorithm* for a game $\textsc{Game}(I, \Gamma)$, where Γ is a finite constraint language that is closed under assignments, is a procedure that

- gets as input a (non-winning) position J of the game, together with an internal state
- and returns a valid move for the splitter at position J, together with an updated internal state.

Suppose we have a game $\textsc{Game}(I, \Gamma)$ and some additional input S. For a given strategy of the connector, the splitter-algorithm plays the game as one would expect: In the beginning, an internal state is initialized with S (if no additional input is given, it is initialized empty). Whenever the splitter should make its next move, the splitter-algorithm is queried using the current position and internal state and afterwards the internal state is updated accordingly.

▶ **Definition 5.** *We say a splitter-algorithm* implements a strategy to reach for a game $\textsc{Game}(I, \Gamma)$ and input S within at most d rounds a position and internal state with some property *if and only if initializing the internal state with S and then playing $\textsc{Game}(I, \Gamma)$ according to the splitter-algorithm leads – no matter what strategy the connector is using – after at most d rounds to a position and internal state with said property.*

Using the following observation converts splitter-algorithms into algorithms for backdoors. It builds backdoors by always trying out all the next moves of the connector.

▶ **Lemma 6** ($\star$). *Assume there exists a function $f(d, \Gamma)$ and a splitter-algorithm that implements a strategy to reach for each game $\textsc{Game}(I, \Gamma)$ and non-negative integer d within at most $f(d, \Gamma)$ rounds either a winning position or (an internal state representing) a proof that the Γ-backdoor depth of I is larger than d.*

Further assume this splitter-algorithm always takes at most $\mathcal{O}(|I|)$ time to compute its next move. Then there exists an algorithm that, for a CSP instance I, a finite constraint language Γ that is closed under assignments, and a non-negative integer d in time at most $|D(I)|^{2f(d,\Gamma)}\mathcal{O}(|I|)$ either returns a component Γ-backdoor tree of depth at most $f(d, \Gamma)$ or concludes that the Γ-backdoor depth of I is larger than d.

For improved readability, we may present splitter-algorithms as continuously running algorithms that periodically output moves (via some output channel) and always immediately as a reply get the next move of the connector (via some input channel). Such an algorithm can easily be converted into a procedure that gets as input a position and internal state and outputs a move and a modified internal state: The internal state encodes the whole state of the computation, (e.g., the current state of a Turing machine together with the contents of the tape and the position of the head). Whenever the procedure is called, it “unfreezes” this state, performs the computation until it reaches its next move and then “freezes” and returns its state together with the move.

Our main result is an approximation algorithm (Theorem 19) that either concludes that there is no backdoor of depth d, or computes a component backdoor tree of depth at most $2^{\mathcal{O}(d)}$. Using Lemma 6, we see that this is equivalent to a splitter-algorithm that plays for $2^{\mathcal{O}(d)}$ rounds to either reach a winning position or a proof that the backdoor depth is larger than d.

Here and in the following, we say that a constraint is Γ-bad for a finite constraint language Γ if its relation is not in Γ; otherwise we say that the constraint is Γ-good. Note that if Γ is close under assignments, then a Γ-good constraint remains Γ-good even after assigning additional variables and a conversely a constraint that is Γ-bad in some subinstance obtained by assigning some variables is also Γ-bad in the original instance.

Our proofs of high backdoor depth come in the form of so-called *obstruction trees*, which have first been introduced by Mählmann et al. [23]. These are trees in the incidence graph of a CSP instance. Their node set therefore consists of both variables and constraints. Obstruction trees of depth d describe parts of an instance for which the splitter needs more than d rounds to win the backdoor depth game. For depth zero, we simply take a single Γ-bad constraint that is not allowed by the base class. Roughly speaking, an obstruction tree of depth $d > 0$ is built from two "separated" obstruction trees T_1, T_2 of depth $d-1$ that are connected by a path. Because the two obstruction trees are separated but in the same component, we know that for any choice of the splitter (i.e., choice of a variable v), there is a response of the connector (i.e., an assignment of v and a component) in which either T_1 or T_2 is whole. Then the splitter needs by induction still more than $d-1$ additional rounds to win the game.

▶ **Definition 7.** *Let I be a CSP instance and Γ be a constraint language that is closed under assignments. We inductively define Γ-obstruction trees T of increasing depth.*

- *Let c be a Γ-bad constraint of I. The set $T = \{c\}$ is a* Γ-obstruction tree in I of depth 0.
- *Let T_1 be a Γ-obstruction tree of depth i in I. Let β be a partial assignment of the variables in I. Let T_2 be an obstruction tree of depth i in $I[\beta]$ such that that no variable $v \in V(I[\beta])$ is contained both in a constraint of T_1 and T_2. Let further P be a path (in the incidence graph) connecting T_1 and T_2 in I. Then $T = T_1 \cup T_2 \cup V(P) \cup C(P)$ is a* Γ-obstruction tree in I of depth $i+1$.

▶ **Lemma 8** ($\star$). *Let I be a CSP instance and Γ be a constraint language that is closed under assignments. If there is a Γ-obstruction tree of depth d in I, then the Γ-backdoor depth of I is at least $d+1$.*

Our splitter-algorithm will construct obstruction trees of increasing depth by a recursive procedure (Lemma 18) that we outline now. We say a splitter-algorithm satisfies *property i* if it reaches in each game $\textsc{Game}(I, \Gamma)$ within $g_{\mathcal{C}}(i,d)$ rounds (for some function $g_{\mathcal{C}}(i,d)$) either

i) a winning position, or
ii) a position J and a Γ-obstruction tree T of depth i in I such that no variable in $var(J)$ occurs in a constraint of T, or
iii) a proof that the Γ-backdoor depth of I is at least d.

A splitter-algorithm satisfying property $d+1$ then directly implies our main result, the approximation algorithm for backdoor depth, using Lemma 8 and Lemma 6. Assume we have a strategy satisfying property $i-1$, let us describe how to use it to satisfy property i. If at any point we reach a winning position, or a proof that the Γ-backdoor depth of I is at least d, we are done. Let us assume this does not happen, so we can focus on the much more interesting case 2).

We use property $i-1$ to construct a first tree T_1 of depth $i-1$, and reach a position J_1. We use it again, starting at position J_1 to construct a second tree T_2 of depth $i-1$ that is completely contained in position J_1. Since T_1 and T_2 are in the same component of F, we can find a path P connecting them. Let β be the assignment that assigns all the variables the splitter chose until reaching position J_1. Then T_2 is an obstruction tree not only in J_1 but also in $I[\beta]$. In order to join both trees together into an obstruction of depth i, we have to show, according to Definition 7 that no variable $v \in var(I[\beta])$ occurs both in a constraint of T_1 and T_2. Since no variable in $var(J_1)$ occurs in a constraint of T_1 (property $i-1$), and T_2 was built only from J_1, this is the case. The trees T_1 and T_2 are "separated" and can be safely joined into a new obstruction tree T of depth i (details also in proof of Lemma 18).

Finally, we need to ensure is that we reach a position J such that no variable in $var(J)$ occurs in a constraint of T. This then guarantees that T is "separated" from all future obstruction trees that we may want to join it with to satisfy property $i+1$, $i+2$ and so forth.

It is important to note here, that the exact notion of "separation" between obstruction trees plays a crucial role and is one of the main differences between the approaches used by Mählmann et al. [23] and Dreier et al. [14]. The former solve the separation problem in a "brute-force" manner: If we translate their approach to the language of splitter-algorithms, then the splitter simply selects all variables that occur in a clause of T. For their base class – the class Null of formulas without variables – there are at most $2^{\mathcal{O}(d)}$ variables that occur in an obstruction tree of depth d. Thus, in only $2^{\mathcal{O}(d)}$ rounds, the splitter can select all of them, fulfilling the separation property. This completes the proof for the base class Null.

However, already for backdoor depth to Krom formulas (or equivalently backdoor depth to some finite constraint language of arity at most two), this approach cannot work since obstruction trees for Krom formulas can have arbitrarily many clauses. We solve this issue by adapting the separator obstructions in [14] from SAT to CSP. We also exploit the fact that our base classes have bounded arity (in contrast to, e.g., the class of Horn formulas) to simplify their separator obstructions significantly allowing us to drop the complexity for solving CSP using backdoor depth from triple to double exponential in the backdoor depth.

5 Separator Obstructions

Obstruction trees are made up of paths, therefore, it is sufficient to separate each new path P that is added to an obstruction. Note that P can be arbitrarily long and therefore the splitter cannot simply select all variables in (constraints of) P. Instead, given such a path P that we want to separate, we will use separator obstructions to develop a splitter-algorithm (Lemma 16) that reaches in each game Game(I, Γ) within a bounded number of rounds either

i) a winning position, or
ii) a position J such that no variable in $var(J)$ occurs in a constraint of P, or
iii) a proof that the backdoor depth of I is at least d.

Informally, a separator obstruction is a sequence $\langle P_1, \dots, P_\ell\rangle$ of paths that form a tree T_ℓ together with an assignment τ of certain *important* variables occurring in T_ℓ. The variables of τ correspond to the variables chosen by the splitter-algorithm and the assignment τ corresponds to the assignment chosen by the connector. Each path P_i adds (at least one) Γ-bad constraint b_i to the separator obstruction, which is an important prerequisite to increase the backdoor depth by growing the obstruction. Moreover, by choosing the important variables and the paths carefully, we ensure that the tree T_ℓ has maximum degree at most three and that every *outside* variable, i.e., any variable that is not an important variable assigned by τ, can occur in at most four constraints of T_ℓ. Therefore assigning any outside variable can split T_ℓ in only constantly many parts. Together with the assignment τ, which we will use as a guide for the connector for the variables inside the obstruction, this will allow us to show that the connector can play in such a way that after every round at least a constant fraction of the separator obstruction remains intact. This means a large separator obstruction is a proof that the backdoor depth is larger than d.

To illustrate the growth of a separator obstruction (and motivate its definition) suppose that our splitter-algorithm is at position J of the game Game(I, Γ) and already has built a separator obstruction $X = \langle\langle P_1, \dots, P_i\rangle, \tau\rangle$ containing Γ-bad constraints $b_1, \dots, b_i$; note that τ is compatible with τ_J. If J is already a winning position, then we are done. Therefore, J has to contain a Γ-bad constraint. If no Γ-bad constraint has a path to T_i in J, then J satisfies 2) and we are also done. Otherwise, let b_{i+1} be a Γ-bad constraint in J that is closest to T_i and let P_{i+1} be a shortest path from b_{i+1} to T_i in J. Then, we extend our separator obstruction X by attaching the path P_{i+1} to T_i (and obtain the tree T_{i+1}). Our next order of business is to choose a bounded number of important variables occurring on P_{i+1} that we will add to X. Those variables need to be chosen in such a way that no outside variable can destroy too much of the separator obstruction. Apart from destroying the paths of the separator obstruction, we also need to avoid that assigning any outside variable makes too many of the Γ-bad constraints $b_1, \dots, b_{i+1}$ Γ-good. Therefore, a natural choice is all variables of b_{i+1} to X, i.e., to make those variables important. The following lemma shows that this is possible, because the number of those variables is bounded.

▶ **Lemma 9** ($\star$). *Let I be a CSP instance and Γ be a finite constraint language. If I has Γ-backdoor depth at most some integer d, then every constraint of I has arity at most $d + \text{arity}(\Gamma)$.*

The next thing that we need to ensure is that any outside variable can not destroy too many paths. Note that by choosing a *shortest* path P_{i+1}, we have already ensured that no variable occurs on more than two constraints of P_{i+1} (such a variable would be a shortcut, meaning P_{i+1} was not a shortest path). Moreover, because P_{i+1} is a shortest path from b_{i+1} to T_i, we know that every variable that occurs on T_i and on P_{i+1} must occur in the constraint c in P_{i+1} that is closest to T_i but not in T_i itself. Similarly, to how we dealt with the Γ-bad constraints, we will now add all variables that occur in c to X. This ensures that no outside variable can occur in both T_i and P_{i+1} , which (by induction over i) implies that every outside variable occurs in at most two constraints (either from T_i or from P_{i+1}). However, since removing any single constraint can still be arbitrarily bad if the constraint has a high degree in the separator obstruction, we further need to ensure that all constraints of the separator obstruction have small degree. We achieve this by adding the variables occurring in any constraint as soon as its degree (in the separator obstruction) becomes larger than two, which happens whenever the endpoint of P_{i+1} in T_i is a constraint. Finally, if the endpoint of P_{i+1} in T_i is a variable, we also add this variable to the separator obstruction to ensure that no variable has degree larger than three in T_{i+1}. This leads us to the following definition of separator obstructions (see also Figure 1 for an illustration).

▶ **Definition 10.** *A* Γ-separator obstruction *for I is a tuple $X = \langle\langle P_1, \dots, P_\ell\rangle, \tau\rangle$ satisfying the following conditions.*

- *P_1 is a shortest Γ-good path between two Γ-bad constraints b_0 and b_1 in I.*
- *For $1 \leq i \leq \ell$, let $T_i = \bigcup_{1\leq j\leq i} P_j$.*
- *For every $i \in [2, \ell]$, P_i is a shortest Γ-good path from T_{i-1} to a Γ-bad constraint b_i that is closest to T_{i-1} in $I[\tau_{i-1}]$, where for every $i \in [0, \ell]$, τ_i is the restriction of τ to the variables in V_i given below.*
- *For every $i \in [2, \ell]$, let e_i be the constraint in $P_i \setminus T_{i-1}$ that is closest to T_{i-1} and let f_i be the constraint $P_i \cap T_{i-1}$ if $P_i \cap T_{i-1}$ is a constraint, otherwise we set $f_i = e_i$. Moreover, for every $i \in [\ell]$, let B_i be the set $\{b_0, b_1, b_2, e_2, f_2, \dots, b_i, e_i, f_i\}$ of constraints and let V_i be the set of all variables occurring in any constraint in B_i. Then, τ_i is the restriction of τ to V_i and τ assigns exactly the variables in V_ℓ.*

We define the size *of X to be the number of leaves of $T = T_\ell$.*

We start by showing some simple but important properties of separator obstructions.

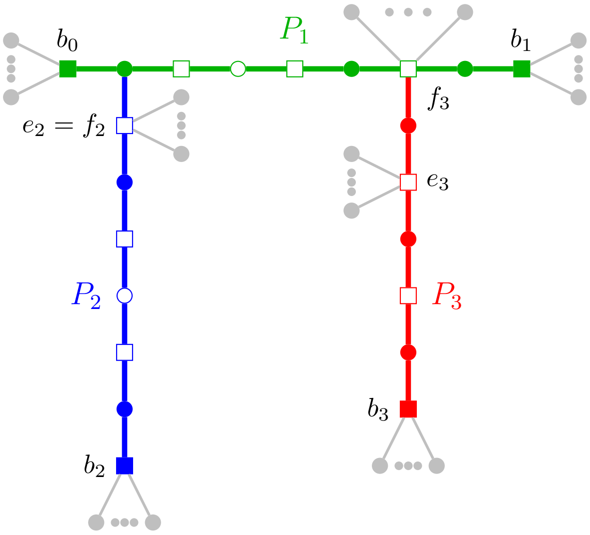


Figure 1 A separator obstruction containing three paths P_1, P_2, and P_3. The figure shows the vertices and edges of the incident graph. Variables are represented by circles and constraints are represented by rectangles. Filled variables are contained in V_3 (all other variables are not) and filled rectangles are bad constraints (all other constraints are good). Only the black variables and edges are part of the tree of the separator obstruction, grey variables and edges are not part of the tree but are part of V_3.

▶ **Lemma 11** ($\star$). *Let Γ be a finite constraint language that is closed under assignments and let $X = \langle\langle P_1, \ldots, P_\ell\rangle, \tau\rangle$ be a Γ-separator obstruction in I, then for every $i \in [\ell]$:*

(P1) *T_i is a tree with leaves $b_0, \ldots, b_i$.*

(P2) *T_i has maximum degree at most 3.*

(P3) *Every variable $v \notin V_i$ is contained in at most two constraints of T_i and moreover those constraints are consecutive in T_i.*

(P4) *Every variable $v \in V_i \setminus V_{i-1}$ is contained in most 4 constraints of T_i.*

(P5) *If β is any assignment compatible with τ that does not assign any variable in $V_i \setminus V_{i-1}$, then b_i is a Γ-bad constraint in $I[\beta]$.*

Our next aim is to show that separator obstructions – just like obstruction trees – can be employed to obtain a lower bound on the backdoor depth of a CSP instance. For this it is important to show that assigning a single variable cannot sufficiently destroy a separator obstruction.

Note that Lemma 11 already provides a first step in this direction. In particular, (P3) limits the influence of variables outside of V_ℓ to only two constraints and (P4) limits the influence of variables inside V_ℓ, at least towards the part of the separator obstruction that was constructed before the variable was added. To limit the influence of variables in V_ℓ also on the remaining part of the separator obstruction, we show that even though these variables can appear in arbitrary many constraints of the remaining part, their influence is still limited as long as we only consider CSP instances obtained by assigning those variables according to τ.

▶ **Definition 12.** *Let $X = \langle\langle P_1, \ldots, P_\ell\rangle, \tau\rangle$ be a Γ-separator obstruction for I and let β be an assignment that is compatible with τ. Moreover, let c be a constraint contained in T and let i be minimal such that c is contained in T_i. We say that c is* tainted *by β, if $V(\beta)$ contains a* *variable v in the scope of c such that $v \notin V_{i-1}$. Otherwise we say that c is* untainted *by β. Similarly, we say that a subtree T' is untainted by β if so is every constraint of T' and moreover $V(\beta)$ does not contain a variable of T'.*

▶ **Lemma 13** ($\star$). *Let Γ be a finite constraint language that is closed under assignments and let $X = \langle\langle P_1, \dots, P_\ell\rangle, \tau\rangle$ be a Γ-separator obstruction in I, let β be an assignment that is compatible with τ, and let T' be a subtree of T untainted by β. Then, $I[\beta]$ contains T'.*

▶ **Lemma 14** ($\star$). *Let T be a tree with n leaves and maximum degree g and let $R \subseteq V(T)$. Then, $T - R$ has a component containing at least $(n - |R|)/(g|R|)$ leaves of T.*

We are now ready to show our main result of this subsection, namely, that separator obstructions can be used to obtain lower bounds on the backdoor depth of a CSP instance.

▶ **Lemma 15.** *Let Γ be a finite constraint language that is closed under assignments and let I be a CSP instance. If I has a Γ-separator obstruction of size at least $n = (d + 2)(15)^d$, then I has Γ-backdoor depth at least d.*

Proof. Let $X = \langle\langle P_1, \dots, P_\ell\rangle, \tau\rangle$ be a Γ-separator obstruction for I of size at least $n = (d + 2)(15)^d$ and let $B = \{b_0, \dots, b_\ell\}$.

Consider the following strategy S for the connector in the game $\textsc{Game}(I, \Gamma)$. Suppose that we have reached position J in the game and suppose that the splitter chooses a variable v as his next move. We distinguish the following two cases:

1. If $v \notin V_\ell$, then the connector plays an arbitrary assignment α for v and chooses a component of $J[\alpha]$ containing a subtree untainted by $\tau_J \cup \alpha$ of T containing the largest subset of B among all components of $J[\alpha]$.
2. If $v \in V_\ell$, then the connector plays the assignment $\alpha(v) = \tau(v)$ for v and chooses the component of $J[\alpha]$ containing a subtree of T untainted by $\tau_J \cup \alpha$ containing the largest subset of B among all components of $J[\alpha]$.

Let J be a position reached in the game $\textsc{Game}(I, \Gamma)$ against S at round i. We show by induction on i that J contains a subtree of T untainted by τ_J containing at least $n_i = n/(15^i) - 1$ elements from B; note that because of Lemma 11 (P1) the elements of B contained in the subtree are the leaves of the subtree.

The claim clearly holds for $i = 0$ since the connector chooses the component of I containing T. Moreover, for $i > 0$ let J' be the predecessor (position) of J in $\textsc{Game}(I, \Gamma)$. By the induction hypothesis J' contains a subtree T' of T untainted by $\tau_{J'}$ containing at least $n_{i-1} = n/(15^{i-1}) - 1$ elements from B. Let v be the variable chosen by the splitter at position J' and let α be the assignment of v chosen by the connector.

If $v \notin V_\ell$, then it follows from Lemma 11 (P3) with $i = \ell$ that v is contained in at most 2 constraints of T and therefore α can taint at most 2 constraints of T. Otherwise let $1 \le i \le \ell$ be minimal such that $v \in V_i$. Assume for contradiction α taints a constraint c in $T \setminus T_i$. Then let j be minimal such that c is contained in T_j. Obviously, $j > i$. But then $v \notin V_{j-1}$, a contradiction to our choice of i. This means α cannot taint any constraints in $T \setminus T_i$. Since $1 \le i \le \ell$ is minimal with $v \in V_i$, we have $v \in V_i \setminus V_{i-1}$ and by (P4) v is contained in at most 4 constraints of T_i. This means α can taint at most 4 constraints of T_i. In total, α can taint at most 4 constraints of T and therefore also of T'. Further, since T' is untainted by τ_J and $\tau_{J'} = \tau_{J'} \cup \alpha$, the assignment τ_J taints at most 4 constraints of T'.

Moreover, because of Lemma 13 and the fact that $\tau_{J'} \cup \alpha$ is compatible with τ, it follows that every subtree of T' untainted by α is contained in some connected component of $J'[\alpha]$. Since T' has maximum degree at most 3 due to Lemma 11 (P2), we obtain from

Observation 14 that after removing the at most 4 constraints together with the variable v from J', there is a component of $J'[\alpha]$ containing a subtree of T' untainted by τ_J with at least $(n_{i-1} - 5)/(3 \cdot 5) = (n_{i-1}/15) - 1/3 \geq (n/15^{i-1} - 1)/15 - 1/3 = n/15^i - 2/5 \geq n_i$ elements of B. Since the connector will choose such a component this concludes the proof of the claim.

Therefore, we obtain that if J is a position reached after i rounds in the game $\textsc{Game}(I, \Gamma)$ against S, then J contains a subtree of T untainted by τ_J containing at least $n_i = n/(15)^i - 1$ constraints from B. In particular, this implies that if J is a position reached after d rounds against S, then J contains a subtree of T untainted by τ_J containing at least $n/(15)^d - 1 = d+1$ constraints from B. Finally, because of Lemma 11 (P5) at least one of these constraints is Γ-bad in J, which concludes the proof of the lemma. ◀

6 Winning Strategies and Algorithms

In this section, we will present our algorithmic results. In Section 4, we discussed that separator obstructions are used to separate existing obstruction trees from future obstruction trees. As all obstruction trees are built only from shortest paths, it is sufficient to derive a splitter-algorithm that takes a shortest path P and separates it from all future obstructions. By reaching a position J such that no variable in $var(J)$ occurs in a constraint of P, we are guaranteed that all future obstructions are separated from P, as future obstructions will only contain constraints and variables from J.

▶ **Lemma 16.** *Let Γ be a finite constraint language that is closed under assignments. There exists a splitter-algorithm that implements a strategy to reach for each game $\textsc{Game}(I, \Gamma)$, non-negative integer d, and shortest Γ-good path P between two Γ-bad constraints in I within at most $(3 \cdot \mathrm{arity}(\Gamma) + d)(d + 2)(15)^d$ rounds either:*

i) *a winning position, or*

ii) *a position J such that no variable in $V(J)$ is contained in a constraint of P, or*

iii) *a proof that the Γ-backdoor depth of I is larger than d.*

This algorithm takes at most $\mathcal{O}(|I|)$ time per move.

Proof. Let $X = \langle\langle P_1, \ldots, P_\ell\rangle, \tau\rangle$ be a Γ-separator obstruction for I and let τ' be a sub-assignment of τ assigning at least all variables in $V_{\ell-1}$. Then, we call $X = \langle\langle P_1, \ldots, P_\ell\rangle, \tau'\rangle$ a *partial Γ-separator obstruction* for I.

Consider the following splitter-algorithm, where for each position J of the game $\textsc{Game}(I, \Gamma)$, we additionally associate a partial Γ-separator obstruction denoted by $X(J) = \langle\langle P_1, \ldots, P_\ell\rangle, \tau_J\rangle$ with $P_1 = P$ to every position J. We set $X(S) = \langle\langle P\rangle, \emptyset\rangle$ for the starting position S of the game.

Then, the splitter-algorithm does the following for a position J in $\textsc{Game}(I, \Gamma)$. If $X(J) = \langle\langle P_1, \ldots, P_\ell\rangle, \tau_J\rangle$ and there is at least one variable in $V_\ell \setminus V_{\ell-1}$ (assuming that $V_0 = \emptyset$) that has not yet been assigned by τ_J, then the splitter chooses any such variable. Otherwise, $X(J)$ is a Γ-separator obstruction and we distinguish the following cases:

1. If there is a Γ-bad constraint in J that has a path to some vertex of T_ℓ, then let b be a Γ-bad constraint that is closest to any vertex of T_ℓ in J and let P' be a shortest Γ-good path from b to some vertex of T_ℓ in J. Note that $X = \langle\langle P_1, \ldots, P_\ell, P_{\ell+1}\rangle, \tau_J\rangle$, where $P_{\ell+1} = P'$, is a partial Γ-separator obstruction for I. The splitter now chooses any variable in $V_{\ell+1} \setminus V_\ell$ and assigns $X' = \langle\langle P_1, \ldots, P_\ell, P_{\ell+1}\rangle, \tau_{J'}\rangle$ to the position J' resulting from this move.

2. Otherwise, $X(J)$ can no longer be extended and either: (a) there is no Γ-bad constraint in J, in which case we reached a winning position, i.e., we achieved case i), or (b) every Γ-bad constraint of J has no path to T_ℓ, which implies that no variable of J is contained in a constraint of T_ℓ and therefore also of P, i.e., we achieved case ii).

This completes the description of the splitter-algorithm. Moreover, if every play against the splitter-algorithm ends after at most $(3 \cdot \text{arity}(\Gamma) + d)(d+2)(15)^d$ rounds, every position is either of type i) or type ii) and we are done.

Otherwise, there is a position J that is reached after playing at least $(3 \cdot \text{arity}(\Gamma) + d)(d+2)(15)^d$ rounds. Then, $X(J)$ has size at least $(d+2)(15)^d$ because the size of the Γ-separator obstruction increases by at least 1 after at most $3 \cdot \text{arity}(\Gamma) + d$ steps. This is because every time the Γ-separator obstruction increases by 1, we only add the at most $\text{arity}(\Gamma) + d + 1$ variables of at most one Γ-bad constraint b_i (because of Lemma 9) and the at most $2 \cdot \text{arity}(\Gamma)$ variables of at most two Γ-good constraints (e_i and f_i). Therefore, it follows from Lemma 15 that I has Γ-backdoor depth at least d.

Finally, the splitter-algorithm takes time at most $\mathcal{O}(|I|)$ per round since a Γ-bad constraint that is closest to the current Γ-separator obstruction and the associated shortest path can be found using a simple breadth-first search. ◀

Since selecting more variables can only help the splitter in archiving their goal, we immediately also get the following statement.

▶ **Corollary 17.** *Consider a finite constraint language Γ that is closed under assignments, a game $\text{Game}(I, \Gamma)$ and a position J' in this game, a non-negative integer d and shortest Γ-good path P between two Γ-bad constraints in I. There exists a splitter-algorithm that implements a strategy that continues the game from position J' and reaches within at most $(3 \cdot \text{arity}(\Gamma) + d)(d+2)(15)^d$ rounds either:*

i) *a winning position, or*

ii) *a position J such that no variable in $V(J)$ is contained in a constraint of P, or*

iii) *a proof that the Γ-backdoor depth of I is larger than d.*

This algorithm takes at most $\mathcal{O}(|I|)$ time per move.

As described at the end of Section 4, we can now construct in the following lemma obstruction trees of growing size, using the previous corollary to separate them from potential future obstruction trees.

▶ **Lemma 18** ($\star$). *Let Γ be a finite constraint language that is closed under assignments. There is a splitter-algorithm that implements a strategy to reach for a game $\text{Game}(I, \Gamma)$ and non-negative integers i and d with $1 \le i \le d$ within at most $(2^{i+1} - 1)(3 \cdot \text{arity}(\Gamma) + d)(d+2)(15)^d$ rounds either:*

i) *a winning position, or*

ii) *a position J and a Γ-obstruction tree T of depth i in I such that no variable in $V(J)$ is contained in a constraint of T, or*

iii) *a proof that the Γ-backdoor depth of I is larger than d.*

This algorithm takes at most $\mathcal{O}(|I|)$ time per move.

Given Lemma 18, the remaining results now follow easily.

▶ **Theorem 19.** *Let Γ be a finite constraint language that is closed under assignments. We can, for a given CSP instance I and a non-negative integer d, in time at most $|D(I)|^{2^{\mathcal{O}(d)}}|I|$ either:*

1. *compute a component Γ-backdoor tree of I of depth at most $2^{\mathcal{O}(d)}$, or*
2. *conclude that the Γ-backdoor depth of I is larger than d.*

Proof. An obstruction tree of depth d is a proof that the backdoor depth is higher than d, thus for the case $i = d$ the output of the splitter-algorithm in Lemma 18 after $2^{\mathcal{O}(d)}$ rounds reduces to either a winning position, or a proof that the Γ-backdoor depth of I is larger than d. The algorithm takes at most $\mathcal{O}(|I|)$ time per move. The statement then follows from Lemma 6. ◀

▶ **Corollary 20.** *Let Γ be a tractable constraint language that is finite and semi-conservative. The CSP can be solved in time $\delta^{2^{\mathcal{O}(d)}}(|I|)^{\mathcal{O}(1)}$ for instances I with $\delta = |D(I)|$ and $d = \mathrm{depth}_\Gamma(I)$.*

Proof. According to Lemma 1, the closure Γ^* of Γ is also tractable. Furthermore, Γ^* is more permissive than Γ and therefore $\mathrm{depth}_{\Gamma^*}(I) \leq \mathrm{depth}_\Gamma(I) = d$. We use Theorem 19 to compute a component Γ^*-backdoor tree of depth $2^{\mathcal{O}(d)}$ in I and then use Lemma 2 to solve I in time $\delta^{2^{\mathcal{O}(d)}}(|I|)^{\mathcal{O}(1)}$. ◀

We would like to mention a corollary of Theorem 19 that we can derive very similarly to Corollary 20. Consider the #CSP problem, which asks for the number of satisfying assignments. A constraint language is *#-tractable* if #CSP is solvable in polynomial time for instances from $\mathcal{C}_\Gamma$ [2]. The proof of Lemma 2 can easily be adapted to #CSP, as at a variable node, we have to add, and at a component node we have to multiply. Hence, we can substitute in the statement of Corollary 20 CSP with #CSP and tractable with #-tractable.

7 Conclusion

In this work, we compute backdoors of bounded depth for the CSP to base classes defined via finite semi-conservative constraint languages. Our approach via obstruction trees seems to be fundamentally limited to semi-conservative languages. However, we are optimistic that our techniques can be extended to base classes of unbounded arity. A first step in this direction has already been obtained in the context of SAT for the base class of Horn formulas [14]. In this setting, it is particularly interesting to consider tractable classes (of unbounded arity) of CSPs based on restrictions on the graphical structure [7, 21, 22], as well as hybrid restrictions [8, 9, 11].

Another interesting direction for future research, which has also been mentioned in the context of SAT [14], are the so-called scattered and heterogeneous extensions of (strong) backdoor sets [19, 20].

These extensions can be readily lifted to backdoor depth by allowing each component to be in any of a given set of (heterogeneous) tractable base classes. Interestingly, while those two notions lead to orthogonal tractable classes in the context of backdoor size, they lead to the same notion for backdoor depth. Therefore, lifting these two extensions to backdoor depth, would result in a unified and significantly more general approach. Moreover, we think that obtaining a heterogeneous version of backdoor depth seems to be particularly promising within the context of CSP. This is because, in contrast to SAT, there is a wide range of tractable classes (even of bounded arity) that can be characterized in a unified manner via algebraic properties.

References

1 Andrei A. Bulatov. Complexity of conservative constraint satisfaction problems. *ACM Trans. Comput. Log.*, 12(4):24:1–24:66, 2011. doi:10.1145/1970398.1970400.

2 Andrei A. Bulatov. The complexity of the counting constraint satisfaction problem. *J. of the ACM*, 60(5):34:1–34:41, 2013. doi:10.1145/2528400.

3 Andrei A. Bulatov. A dichotomy theorem for nonuniform CSPs. In Chris Umans, editor, *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 319–330. IEEE Computer Society, 2017. doi:10.1109/FOCS.2017.37.

4 Jannis Bulian and Anuj Dawar. Graph isomorphism parameterized by elimination distance to bounded degree. *Algorithmica*, 75(2):363–382, 2016.

5 Clément Carbonnel and Martin C. Cooper. Tractability in constraint satisfaction problems: a survey. *Constraints*, 21(2):115–144, 2016.

6 David Cohen and Peter Jeavons. The complexity of constraint languages. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume I, chapter 8, pages 245–280. Elsevier, 2006.

7 David Cohen, Peter Jeavons, and Marc Gyssens. A unified theory of structural tractability for constraint satisfaction and spread cut decomposition. In *International Joint Conferences on Artificial Intelligence (IJCAI-05)*, pages 72–77, 2005.

8 David A. Cohen, Martin C. Cooper, Páidí Creed, Dániel Marx, and András Z. Salamon. The tractability of CSP classes defined by forbidden patterns. *J. Artif. Intell. Res.*, 45:47–78, 2012.

9 David A. Cohen, Martin C. Cooper, Peter G. Jeavons, and Stanislav Zivný. Tractable classes of binary CSPs defined by excluded topological minors. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1945–1951. AAAI Press, 2015.

10 Martin C. Cooper, David A. Cohen, and Peter Jeavons. Characterising tractable constraints. *Artificial Intelligence*, 65(2):347–361, 1994. doi:10.1016/0004-3702(94)90021-3.

11 Martin C. Cooper, Philippe Jégou, and Cyril Terrioux. A microstructure-based family of tractable classes for CSPs. 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 74–88. Springer Verlag, 2015.

12 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms.* Springer, 2015.

13 Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity.* Texts in Computer Science. Springer Verlag, 2013.

14 Jan Dreier, Sebastian Ordyniak, and Stefan Szeider. SAT backdoors: Depth beats size. In *30th Annual European Symposium on Algorithms (ESA 2022)*, volume 244 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. to appear. arXiv:2202.08326.

15 Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*, volume XIV of *Texts in Theoretical Computer Science. An EATCS Series.* Springer Verlag, Berlin, 2006.

16 Fedor V Fomin, Petr A Golovach, and Dimitrios M Thilikos. Parameterized complexity of elimination distance to first-order logic properties. *arXiv preprint arXiv:2104.02998*, 2021.

17 Robert Ganian, M. S. Ramanujan, and Stefan Szeider. Discovering archipelagos of tractability for constraint satisfaction and counting. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1670–1681. SIAM, 2016.

18 Robert Ganian, M. S. Ramanujan, and Stefan Szeider. Combining treewidth and backdoors for CSP. In Heribert Vollmer and Brigitte Vallée, editors, *34th Symposium on Theoretical Aspects of Computer Science (STACS 2017)*, volume 66 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 36:1–36:17, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.STACS.2017.36.

19 Robert Ganian, M. S. Ramanujan, and Stefan Szeider. Discovering archipelagos of tractability for constraint satisfaction and counting. *ACM Transactions on Algorithms*, 13(2):29:1–29:32, 2017. Full version of a SODA'16 paper. doi:10.1145/3014587.

20 Serge Gaspers, Neeldhara Misra, Sebastian Ordyniak, Stefan Szeider, and Stanislav Zivny. Backdoors into heterogeneous classes of SAT and CSP. *J. of Computer and System Sciences*, 85:38–56, 2017. doi:10.1016/j.jcss.2016.10.007.

21 Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, 2000.

22 Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. of Computer and System Sciences*, 64(3):579–627, 2002.

23 Nikolas Mählmann, Sebastian Siebertz, and Alexandre Vigny. Recursive backdoors for SAT. In Filippo Bonchi and Simon J. Puglisi, editors, *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021, August 23-27, 2021, Tallinn, Estonia*, volume 202 of *LIPIcs*, pages 73:1–73:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.MFCS.2021.73.

24 Jaroslav Nesetril and Patrice Ossona de Mendez. Tree-depth, subgraph coloring and homomorphism bounds. *European J. Combin.*, 27(6):1022–1041, 2006.

25 Jaroslav Nešetřil and Patrice Ossona de Mendez. *Sparsity - Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and combinatorics*. Springer, 2012.

26 Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics and its Applications. Oxford University Press, Oxford, 2006.

27 Sebastian Ordyniak, Andre Schidler, and Stefan Szeider. Backdoor DNFs. In Zhi-Hua Zhou, editor, *Proceeding of IJCAI-2021, the 30th International Joint Conference on Artificial Intelligence*, pages 1403–1409, 2021. doi:10.24963/ijcai.2021/194.

28 Marko Samer and Stefan Szeider. Backdoor trees. In *AAAI 08, Twenty-Third Conference on Artificial Intelligence, Chicago, Illinois, July 13–17, 2008*, pages 363–368. AAAI Press, 2008.

29 Marko Samer and Stefan Szeider. Fixed-parameter tractability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability, 2nd Edition*, chapter 17, pages 693–736. IOS Press, 2021. doi:10.3233/FAIA201000.

30 Thomas J. Schaefer. The complexity of satisfiability problems. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing (San Diego, Calif., 1978)*, pages 216–226. ACM, 1978.

31 Ryan Williams, Carla Gomes, and Bart Selman. Backdoors to typical case complexity. In Georg Gottlob and Toby Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, IJCAI 2003*, pages 1173–1178. Morgan Kaufmann, 2003.

32 Ryan Williams, Carla Gomes, and Bart Selman. On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. In *Informal Proc. of the Sixth International Conference on Theory and Applications of Satisfiability Testing, S. Margherita Ligure - Portofino, Italy, May 5-8, 2003 (SAT 2003)*, pages 222–230, 2003.

33 Dmitriy Zhuk. A proof of CSP dichotomy conjecture. In Chris Umans, editor, *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 331–342. IEEE Computer Society, 2017. doi:10.1109/FOCS.2017.38.

Explaining Propagation for Gini and Spread with Variable Mean

Alexander Ek
Dept. of Data Science & AI, Monash University, Melbourne, Australia
CSIRO Data61, Melbourne, Australia

Andreas Schutt
CSIRO Data61, Melbourne, Australia

Peter J. Stuckey
Dept. of Data Science & AI, Monash University, Melbourne, Australia

Guido Tack
Dept. of Data Science & AI, Monash University, Melbourne, Australia

Abstract

In optimisation problems involving multiple agents (stakeholders) we often want to make sure that the solution is balanced and fair. That is, we want to maximise total utility subject to an upper bound on the statistical dispersion (e.g., spread or the Gini coefficient) of the utility given to different agents, or minimise dispersion subject to some lower bounds on utility. These needs arise in, for example, balancing tardiness in scheduling, unwanted shifts in rostering, and desired resources in resource allocation, or minimising deviation from a baseline in schedule repair, to name a few. These problems are often quite challenging. To solve them efficiently we want to effectively reason about dispersion. Previous work has studied the case where the mean is fixed, but this may not be possible for many problems, e.g., scheduling where total utility depends on the final schedule. In this paper we introduce two log-linear-time dispersion propagators – (a) spread (variance, and indirectly standard deviation) and (b) the Gini coefficient – capable of explaining their propagations, thus allowing effective clause learning solvers to be applied to these problems. Propagators for (a) exist in the literature but do not explain themselves, while propagators for (b) have not been previously studied. We avoid introducing floating-point variables, which are usually not supported by learning solvers, by reasoning about scaled, integer versions of the constraints. We show through experimentation that clause learning can substantially improve the solving of problems where we want to bound dispersion and optimise total utility and vice versa.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence; Theory of computation → Constraint and logic programming

Keywords and phrases Spread constraint, Gini index, Filtering algorithm, Constraint programming, Lazy clause generation

Digital Object Identifier 10.4230/LIPIcs.CP.2022.21

Supplementary Material *Software (Source Code)*: `https://www.github.com/aekh/chuffed`
Dataset (Instances, Models, and Results): `https://www.github.com/aekh/CP22-Extras`

Funding This research was partially funded by the Australian Government through the Australian Research Council Industrial Transformation Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Project ID IC200100009.

1 Introduction

In many real-world applications of combinatorial optimisation the statistical dispersion (sometimes called variability or spread) of the variable assignments in the solutions are important to consider. This is because some kind of balance or fairness within the solution

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 21; pp. 21:1–21:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

is desired. As an example, consider a resource allocation problem, where a set A of agents compete over a set R of resources under complex constraints. The utility that an agent $a \in A$ receives from being allocated a resource $r \in R$ may completely depend on what other resources a are also being allocated. For example, a woodworker gets little utility from being allocated nails but not a hammer, or hammer and nails but not planks. In this situation, it may be essential to balance the utilities (or metrics derived from the utilities) each agent receives. A fixed mean is often assumed in the literature, simplifying the constraint; however, fixing the mean is not possible in many contexts. Throughout this paper, we assume the mean is variable.

The **main contribution** in this paper is the development of efficient and explaining propagators, filtering the lower bound, for two kinds of dispersion measured over an array of variables X: (i) spread (i.e., variance and standard deviation), explained in Section 3; and (ii) the Gini coefficent, explained in Section 4. We only consider filtering the lower bound because usually minimisation of dispersion or keeping dispersion under some upper bound is of interest. Problems which require maximising dispersion or ensuring sufficient dispersion need filtering algorithms for the upper bound, and require other types of approximations and roundings than presented here, which we leave to future work.

A dispersion propagator can also be devised that propagates the bounds of the variables X whose dispersion is being measured. Interestingly we performed some preliminary experiments, where we implemented a naive domain consistent propagator for spread by checking each possible assignment to the variables X, and removing unsupported values. This propagator, which is the strongest possible, rarely removed values from the X variables before all but one of them were fixed. This is because the X variables act in aggregate in a large sum, and propagation is notoriously weak for reasoning about large sums. This is even more pronounced in our case because of the squared differences in the sum for variance and absolute differences in the sum for Gini. Hence, we decided not to pursue algorithms for this case, since it was clear they would rarely help.

In Sections 3 and 4 we respectively present lower bounds on spread and the Gini coefficent (Gini) and how to explain the propagation of those. We avoid introducing floating-point variables, which are usually not supported by clause learning solvers, by reasoning over appropriately scaled integer versions of the constraints. Selecting a scale is application-specific: there is a trade-off between the size of integers and fidelity.

2 Background

We briefly introduce main concepts used in this paper starting with those ones in combinatorial optimisation, statistics, and ending with real relaxations.

2.1 Combinatorial Optimisation

A constrained optimisation problem (COP) $P = (X, D, C, o)$ consists of a set of variables X, an initial domain D mapping each $x \in X$ to a set of integer values $x \mapsto D(x) \subseteq \mathbb{Z}$, a set of constraints C over the variables X and an objective function o to be minimised (w.l.o.g). An assignment θ is a mapping from each variable $x \in X$ to a value from its domain $\theta(x) \in D(x)$. An assignment θ is a *solution* of COP P if it makes all constraints $c \in C$ true. An assignment θ is an *optimal solution* if $\theta(o) \leq \theta'(o)$ for all solutions θ' of P.

An often effective way to solve COPs is the *constraint programming* (CP) solving technology [14]. In CP, each constraint $c \in C$ has a *propagator* f_c, which uses specialised algorithms and logic to reduce the domains of the variables concerning c when invoked, i.e.,

$f_c(D)(x) \subseteq D(x)$. Only guaranteed non-solutions are removed. If $f_c(D)(x) = \emptyset$ for some $x \in X$, we say that we have reached a failure because there is no solution in D. When no more inference can be made by the propagators, and we do not have a solution or failure yet, we need to search for a solution. This can be done by arbitrarily reducing the domain of a variable $x \in X$ to $D'(x) \subsetneq D(x)$, creating a binary branch point in the search tree of the constraints $x \in D'(x)$ in one child and $x \in D(x) \setminus D'(x)$ in the other.

We use $l..u$ to denote the set of integers $\{l, l+1, \ldots, u-1, u\}$, and for a decision variable x with domain $D(x)$, we use $\overline{x}$ to denote $\max(D(x))$ and $\underline{x}$ to denote $\min(D(x))$.

An often effective augmentation of CP is lazy clause generation (LCG), which combines the techniques of Boolean satisfiability solving (SAT) and CP [9]. An LCG solver connects Boolean variables concerning bounds (and fixed values, but they will not be needed here) to the integer variables. The Boolean $[\![x \geq d]\!]$ holds if the integer variable x takes a value greater than or equal to d. We use notation $[\![x < d]\!]$ for $\neg[\![x \geq d]\!]$ and $[\![x \leq d]\!]$ for $\neg[\![x \geq d-1]\!]$. An LCG solver tracks the reasons justifying all propagated domain changes happening during search. We call such reasons for each propagation an *explanation*, represented as a set of Boolean clauses, using the literals defined above, which were implied by the domains at the time of propagation and imply the propagation. In case of a failure, these explanations are used to find *nogoods*, which extract the reason for the failure as a new (previously implicit) constraint, representing an erroneous choice made earlier during search and preventing it from reoccurring. We can then backjump to the point just before this choice was made and add the new constraint explicitly to prevent making the same erroneous choice again. A popular LCG solver is Chuffed [3], which we will use and extend in this paper.

2.2 Statistical Preliminaries

Perhaps the most common statistical summary of a set of numbers is that of its central tendency, and arguably the most prevalent of those is the *arithmetic mean.* Suppose we have a *population* X of *integer* values $x_1, \ldots, x_n$. We respectively denote the *arithmetic mean* and the *sum* of these as:

$$\mu_X = \frac{1}{n}\sum\nolimits_{i=1}^{n} x_i, \qquad M_X = \sum\nolimits_{i=1}^{n} x_i.$$

For brevity, we will omit the subscript "X" when there is little room for ambiguity.

A *measure of dispersion* is another kind of statistical summary of a set of numbers. It describes how different or similar values of X are. The first measure of dispersion we consider is the *population variance*, or simply *variance.* The *variance* of X is defined as the average squared difference from the arithmetic mean, with its conventional and alternative formulations as follows:

$$\sigma_X^2 = \frac{1}{n}\sum\nolimits_{i=1}^{n} (x_i - \mu)^2, \qquad \sigma_X^2 = \left(\frac{1}{n}\sum\nolimits_{i=1}^{n} x_i^2\right) - \mu^2. \tag{1}$$

A related measure of dispersion is the *standard deviation*, which is the square-root of variance:

$$\sigma_X = \sqrt{\frac{1}{n}\sum\nolimits_{i=1}^{n} (x_i - \mu)^2},$$

Standard deviation is a monotonic transformation of variance. As such, minimising (and maximising) standard deviation is equivalent to minimising (and maximising) variance. Similarly, any constraints specified on the standard deviation can be squared to obtain the equivalent constraint on the variance. As such we only need to define a propagator for variance.

The second measure of dispersion we consider is the *Gini coefficient*. The *Gini coefficient* of X is conventionally formulated as:

$$Gini(X) = \frac{\sum_{i=1}^{n} \sum_{j=i+1}^{n} |x_i - x_j|}{n \sum_{i=1}^{n} x_i}.$$

Calculating this naïvely takes $\mathcal{O}(|X|^2)$ time, but an alternative linear-time formulation is well-known if X is sorted [4]:

$$Gini(X) = \frac{\sum_{i=1}^{n} (2i - n - 1) x_i}{n \sum_{i=1}^{n} x_i}. \tag{2}$$

The Gini coefficient, originally developed for measuring income inequality, measures how far a distribution X deviates from a totally equal distribution. Note that it is only meant to apply when $x_i > 0$, since otherwise the sum divisor can be zero.

2.3 Real Relaxation

We will examine the case of the dispersion constraints on a set of integer valued variables X. It will often be useful to relax the integrality constraint in order to create lower bounds on dispersion values.

In our lower bound calculations we relax the problem from integer variables to real variables.

▶ **Definition 1.** *An $\mathbb{R}$-assignment of variables X is a mapping θ from X to $\mathbb{R}$ such that $\underline{x} \leq \theta(x) \leq \overline{x}, \forall x \in X$.*

We further make use of ν-centred assignments as defined by [13].

▶ **Definition 2.** *Given a real value ν, an $\mathbb{R}$-assignment θ is ν-*centred *if*

$$\theta(x) = \begin{cases} \overline{x} & \textit{if } \overline{x} \leq \nu \\ \underline{x} & \textit{if } \underline{x} \geq \nu \\ \nu & \textit{otherwise} \end{cases}$$

We denote the ν-centred assignment θ_ν.

3 The Spread Constraint

The spread constraint [13] as initially introduced was defined as `spread`(X,μ,σ,$\tilde{x}$), where μ is constrained to be the mean of the array of (say n) variables $X = [x_1, \ldots, x_n]$, σ the standard deviation, and $\tilde{x}$ the median. We will examine a slightly different form.

Let `spread`(X,M,$n2V$) be defined to constrain M to be the sum of X, i.e., $\sum_{i=1}^{n} x_i$, and $n2V$ to be equal to n^2 times the variance of X, i.e., $n^2\sigma^2$. The advantage of this form is that each of the variables takes integer values. By making use of the alternate expression of variance, the following equations hold:

$$n2V = n \sum_{i=1}^{n} x_i^2 - M^2 \quad \wedge \quad M = \sum_{i=1}^{n} x_i, \tag{3}$$

Since all the variables involved are integer, we can use this within a solver that does not support floating-point variables, without difficulty.

But when n is large, the size of the $n2V$ may often push the boundaries of integers that most solvers can deal with. Thus, we also introduce a relaxed version of the spread constraint that works with a given fixed positive scaling factor s defined as follows.

spread(X,M,v,s) constrains M to be the sum of X and $v = \lfloor \sigma^2 \cdot s \rfloor$, where σ^2 is the variance of X scaled (by s) and rounded down. For instance, $s = 1$ means whole number and $s = 100$ means percentage. Note when this is used as a lower bound it is always correct. The benefit of this relaxed form of the constraint is that it, again, only involves integer variables.

We can define this as a decomposition in MiniZinc using Equation (3), as follows:

```
predicate spread(array[int] of var int:X, var int:M, var int:v, int:s) =
    let { int: n=length(X);
        array[int] of var 0..infinity: sq = [x * x | x in X];
        var 0..n*ub(sum(sq)): v_ = n*sum(sq);
        var 0..max(lb(M)*lb(M),ub(M)*ub(M)): Msq = M*M;
        var 0..ub(v_): v__ = v_ - Msq;
        var 0..ub(v__)*s: v___ = v__*s; }
    in v = v___ div (n*n) /\ M = sum(X);
```

Note that lb and ub return the best known, *at compile time*, lower and upper bound respectively of the argument expression. Note that the first version we defined is implemented by spread(X,M,$n2v$) = spread(X,M,$n2v$,n^2).

3.1 Lower Bound on Variance

In this section we present a log-linear-time propagator for filtering the lower bound of the variance variable using binary chop.

We define a real-value relaxed lower bound formula for the variance:

$$LBV(X, M) = \min_{m=\underline{M}}^{\overline{M}} LBV'(X, \frac{m}{n}), \qquad \text{where} \quad LBV'(X, \nu) = \frac{1}{n} \sum\nolimits_{i=1}^{n} (\theta_\nu(x_i) - \nu)^2$$

Essentially, we pick the ν-centred assignment with the lowest variance. From a previous result by [13] we know that a variance-minimal solution to the spread constraint must be a ν-centred assignment.

▶ **Lemma 3.** *$LBV(X, M)$ is a lower bound on σ^2, i.e., $LBV(X, M) \leq \sigma^2$.*

Proof. Let $Y = \{y_1, \ldots, y_n\}$ be an arbitrary assignment of the variables in X where $y_i \in D(x_i)$ and $\nu = M_Y/n$ be the mean of Y where $M_Y = \sum_{i=1}^{n} y_i$. Now, we only need to prove that $LBV'(X, \nu) \leq \frac{1}{n} \sum_{i=1}^{n} (y_i - \nu)^2$, i.e., the variance of the ν-centred assignment is less than or equal to the variance of the assignment Y. We do so by proving $(\theta_\nu(x_i) - \nu)^2 \leq (y_i - \nu)^2$ for every i. By construction of the ν-centred assignment (see Definition 2), for any i, where $1 \leq i \leq n$, it holds either that $y_i \leq \theta_\nu(x_i) \leq \nu$ for the case $\overline{x}_i \leq \nu$, that $\nu \leq \theta_\nu(x_i) \leq y_i$ for the case $\underline{x}_i \geq \nu$, or that $\nu = \theta_\nu(x_i)$ (i.e., $(\theta_\nu(x_i) - \nu)^2 = 0$ otherwise). Thus, it holds $(\theta_\nu(x_i) - \nu)^2 \leq (y_i - \nu)^2$ in all cases. Hence, the lemma holds. ◀

We can calculate LBV in log-linear time, because LBV' is convex in ν.

▶ **Lemma 4.** *$LBV'(X, \nu)$ is a convex function in ν.*

Proof. Since a sum of convex functions is convex, we only need to prove for each i, where $1 \leq i \leq n$, that the function $f(\nu) = (\theta_\nu(x_i) - \nu)^2$ is convex in ν. By construction of θ_ν (see Definition 2), the difference $\theta_\nu(x_i) - \nu$ is strictly decreasing until $\nu = \underline{x}_i$, then zero until $\nu = \overline{x}_i$, and strictly decreasing again. Since the difference is squared, the function f is a quadratic function with a flat bottom of some length. Hence, it is convex and $LBV'(X, \nu)$ as well. ◀

Algorithm 1 Only run if not all fixed. $n = |X|$.

```
procedure SpreadLb($X$, $M$, $v$, $s$)
    $L,\ R \leftarrow \underline{M},\ \overline{M}$
    while $L < R$ do
        $m_L \leftarrow L + \lfloor (R - L)/2 \rfloor$
        $m_R \leftarrow m_L + 1$
        $V_L,\ V_R \leftarrow LBV'(X, m_L/n),\ LBV'(X, m_R/n)$
        if $\min(V_L, V_R) = 0$ then return true
        if $V_L = V_R$ then
            $V,\ m \leftarrow V_L,\ m_L$
            break
        if $V_L < V_R$ then
            $V,\ m,\ R \leftarrow V_L,\ m_L,\ m_L$
        else if $V_L > V_R$ then
            $V,\ m,\ L \leftarrow V_R,\ m_R,\ m_R$
    $Vs \leftarrow \lfloor V \times s \rfloor$
    if $Vs \le \underline{v}$ then return true
    $EX \leftarrow \{[\![x_i \le \overline{x}_i]\!] \mid \overline{x}_i < m/n\} \cup \{[\![x_i \ge \underline{x}_i]\!] \mid m/n < \underline{x}_i\}$
    if $m = \overline{M}$ then $EX \leftarrow EX \cup \{[\![M \le \overline{M}]\!]\}$
    if $m = \underline{M}$ then $EX \leftarrow EX \cup \{[\![M \ge \underline{M}]\!]\}$
    if $Vs > \overline{v}$ then
        return $[\![v < Vs]\!] \wedge \bigwedge_{l \in EX} l \rightarrow \mathit{false}$
    else
        return $\bigwedge_{l \in EX} l \rightarrow [\![v \ge Vs]\!]$
```

Hence, we can use binary search on the values m in M to find a minimum value for LBV'. Thus, only the mean values $\nu = \frac{m}{n}$ with $\underline{M} \le m \le \overline{M}$ have to be considered. This follows from the proof of Lemma 3 since the use of ν value is set to the mean of an arbitrary assignment.

3.2 Algorithm and Clause Learning Explanations

Algorithm 1 defines our approach to finding and asserting a lower bound on (scaled) variance. Note that when all $x \in X$ are fixed, we can simply calculate the actual variance in $\mathcal{O}(n)$ time.[1] It returns a clause which is a consequence of the constraint, and whose right-hand side gives the new propagation of the lower bound, otherwise it returns the vacuous *true* clause.

The algorithm searches through the integer range $L..R$ of values for the sum m that results in the minimum value for $LBV'(X, m/n)$. We compute the values V_L of LBV' for the (integer) midpoint m_L of $L..R$ and V_R for the position one to the right of m_L, namely m_R. If either of V_L or V_R give 0, then the overall variance bound is 0 and hence we return, since no propagation is possible. If V_L and V_R are equal, then we have found the lowest value and break from the loop, since two variance-equal and neighbouring points can only occur if they are both minima – this follows from convexity. Otherwise, we use the relative values of V_L and V_R to decide which half of the interval to keep. We store the best value found V and the sum value m where it occurs and update the appropriate bound L or R. If the interval ever shrinks to a singleton, then the loop exits. We compute the lower bound Vs on the scaled version of the variance v. If the lower bound is already subsumed, then we return a trivial clause *true*. Otherwise, we collect the literals that will appear in the explanation for the bound change or failure in EX. We collect all the lower and upper bounds that appear in

[1] In this case we still scale and round down, for consistency.

the ν-centred valuation, but not the bounds of the overlapping variables, in the explanation. If the optimal m value resides at one of the extremes of the sum M, then we collect the appropriate literal. Finally, if the new bound will cause unsatisfiability, then we add in the current upper bound for v and return an explanation for failure. Otherwise, we return an explanation for the new bound $v \geq Vs$.

This algorithm runs in $\mathcal{O}(n \log d)$ time, where $n = |X|$ and $d = \overline{M} - \underline{M}$. We can assume this algorithm is $\mathcal{O}(n \log n)$ when d is of size $O(n^k)$ for any reasonably small constant k.

▶ **Lemma 5.** *Algorithm 1 returns a clause which is a consequence of* `spread`*(X,M,v,s).*

Proof. The result clearly holds for the trivial clause *true*. The correctness of the value $V = LBV'(X, m/n)$ computed by Algorithm 1 follows from Lemma 3 and Lemma 4. We now show that the explanation collected in EX is correct, i.e., $EX \rightarrow s \times \sigma^2 \geq \lfloor s \times V \rfloor$ is universally true, which is the clausal explanation in both non-trivial cases. Assume to the contrary that there is a solution η of `spread`(X,M,v,s) satisfying EX that has smaller variance. We can assume η is ν-centred for some ν and bounds on X (not necessarily the current bounds) [13]. But then η must set all x_i variables not appearing in EX to ν, otherwise there is an assignment allowed by EX which would be better. But η as a function of ν is identical to $LBV'(X, \nu)$ around the minima m/n. That is, all variables with upper bound below m/n are set to their upper bound, which is part of EX, and similarly for variables with lower bound above m/n, and the remaining variables take value ν. If m is not the upper or lower bound of the sum M, then m/n is a local (and global) minima of $LBV'(X, \nu)$, and hence also of η. If m is the upper or lower bound of M, then this bound is included in EX, and again m/n is the minima for $LBV'(X, \nu)$ and η. Contradiction. ◀

▶ **Example 6.** Consider an execution of the algorithm where $X = [x_1, x_2, x_3, x_4]$ with current domains $[0, 0..4, 2..3, 4..6]$, M has domain 6..10, v has domain 0..400 and $s = 100$. We start with $L = 6$, $R = 10$. Then $m_L = 8$ and $m_R = 9$. We compute $V_L = LBV'(X, 2) = 2$ and $V_R = LBV'(X, 2.25) = 2.015625$. We set $V = V_L$, $m = 8$ and $R = 8$. We compute $m_L = 7$ and $m_R = 8$, and compute $V_L = LBV'(X, 1.75) = 2.0123$ and $V_R = LBV'(X, 2) = 2$. We set $V = V_R$, $m = 8$ and $L = 8$. We exit the loop, computing $Vs = 200$. We collect the explanation $EX = \{[\![x_1 \leq 0]\!], [\![x_4 \geq 4]\!]\}$ returning the explanation clause $[\![x_1 \leq 0]\!] \wedge [\![x_4 \geq 4]\!] \rightarrow [\![v \geq 200]\!]$. Note that we do not collect $[\![x_3 \geq 2]\!]$. This is because if the domain of x_3 is extended arbitrarily much in either or both directions, then it would still overlap with 2, and if it were reduced arbitrarily much from either or both directions, then the resulting minimal variance would increase. Thus, the lower bound holds regardless of the value of x_3. ◀

4 The Gini Coefficient

Suppose we have an array X of n variables $x_1, \ldots, x_n$. Define the Gini constraint `gini(`X`,` M`,` g`,` s`)` to constrain M to be the sum of X (i.e., $\sum_{i=0}^{n} x_i$) and $g = \lfloor Gini(X) \times s \rfloor$ to be the Gini coefficient variable expressed as an integer scaled by s and rounded down. We can define this as a decomposition in MiniZinc as follows:

```
predicate gini(array[int] of var int:X, var int:M, var int:g, int:s) =
  let { int: n=length(X);
        int: range_size = ub_array(X) - lb_array(X);
        array[int] of var 0..range_size: diffs
            = [ abs(X[i]-X[j]) | i,j in index_set(X) where i<j];
        var 0..s*(n div 2 + 1)*(n div 2)*(range_size): tot_diff
            = sum(diffs) * s;
        var 0..ub(tot_diff) div n: result_ = tot_diff div n;
        var 0..ub(result_): result = result_ div M }
  in v = result /\ M = sum(X);
```

Note that we try to bound the initial domains reasonably tightly, to reduce overhead in the solver. The function `lb_array` (and `ub_array`) returns the least (and greatest) possible value known *at compile time* of an array of decision variables. The upper bound on the sum of differences of numbers in the range $\min X .. \max X$ is to have half at each end, thus $\lceil \frac{n}{2} \rceil \times \lfloor \frac{n}{2} \rfloor \times (range\_size)$ where $range\_size$ is the difference between the minimum and maximum possible values of X.

Before going into the details of the propagator algorithm, we will prove a lemma that will help us. The formulation from Equation (2) will be used, since it is faster to calculate for ν-centred assignments and will simplify the proofs in this section. We show that the minimum Gini coefficient of the problem arises for a ν-centred assignment.

▶ **Lemma 7.** *For a given fixed mean value μ of variables X, a ν-centred assignment leads to the minimal value of the sum of absolute values of pairwise differences.*

Proof. We prove the lemma by contradiction. Assume w.l.o.g., that a non-ν-centred assignment ϕ for X is presented in non-decreasing order, i.e., $\phi(x_i) \leq \phi(x_{i+1})$ for $1 \leq i < n$, with the mean $\mu = \frac{1}{n}\sum_{i=1}^{n} \phi(x_i)$, and let $A = \sum_{i=1}^{n}(2i - n - 1)\phi(x_i)$ be the sum of the absolute values (by Equation (2)). Assume to the contrary that ϕ leads to a minimum sum A. Since the mean μ is fixed, we need not analyse the denominator of the Gini coefficient.

At least one variable must be unfixed, otherwise ϕ must be ν-centred. There must be at least one variable x_j with $\phi(x_j) < \overline{x_j}$, otherwise ϕ is ν-centred with $\nu = \max_{i=1}^{n} \phi(x_i)$. Furthermore, there must be at least one variable x_k with $\underline{x}_k < \phi(x_k)$ (analogously) and $1 \leq j < k \leq n$ (otherwise ϕ must be ν-centred). W.l.o.g., we can assume that $\phi(x_j) < \phi(x_{j+1})$ and $\phi(x_{k-1}) < \phi(x_k)$, because one can reorder the variables having the same value in ϕ, so that indices j and k are the greatest and least one with the values $\phi(x_j)$ and $\phi(x_k)$, respectively.

We construct a new assignment ϕ' for X, having the same mean as ϕ, as follows: $\phi'(x_i) = \phi(x_i)$ for $1 \leq i \leq n$ with $i \notin \{j,k\}$, $\phi'(x_j) = \phi(x_j) + \delta$, and $\phi'(x_k) = \phi(x_k) - \delta$ where $\delta \in \mathbb{R}$ and $\delta > 0$ is chosen, so that $\phi(x_j) + \delta \leq \min\{\phi(x_{j+1}), \overline{x}_j\}$ and $\max\{\phi(x_{k-1}), \underline{x}_k\} \leq \phi(x_k) - \delta$. Note that ϕ' is also in non-decreasing order, thus, its sum of absolute values is $A' = \sum_{i=1}^{n}(2i - n - 1)$ by Equation (2). If A is the minimal sum of absolute values then $0 \geq A - A'$. Since ϕ and ϕ' only differ in the variable values for x_j and x_k, it holds $0 \geq (2j - n - 1)(\phi(x_j) - \phi'(x_j)) + (2k - n - 1)(\phi(x_k) - \phi'(x_k))$, which is $0 \geq (2j - n - 1)(-\delta) + (2k - n - 1)\delta = 2(k - j)\delta$. Due to $j < k$, we have $2(k - j)\delta > 0$, which contradicts the assumption that A is the minimal sum of absolute values. ◀

Since fixing the mean fixes the divisor of the Gini coefficient, we have the following obvious consequence. Given a fixed mean μ, the ν-centred assignment θ_ν where $\mu = \frac{1}{n}\sum_{i=1}^{n} \theta_\nu(x_i)$ leads to minimal Gini coefficient. And this must hold for any mean.

▶ **Corollary 8.** *A Gini-minimal assignment of X must be ν-centred.*

4.1 Lower Bound on Gini Coefficient

In this section we present a log-linear time propagator for filtering the lower bound of the Gini coefficient using binary chop.

We will now, similarly to the approach used for variance, show how to calculate a lower bound on the Gini Coefficient. The key to this is, again, finding the best ν-centred assignment, which we know must be a lower bound.

Let $L = \min_{i=1}^{n} \underline{x_i}$ be the least lower bound and $U = \max_{i=1}^{n} \overline{x_i}$ be the greatest upper bound, then we can define a real-value relaxed lower bound on the Gini coefficient of X as

$$LBG(X) = \min_{\beta=L}^{U} Gini(X,\beta), \qquad \text{where} \quad Gini(X,\nu) = \frac{\sum_{i=1}^{n}\sum_{j=i+1}^{n} |\theta_\nu(x_i) - \theta_\nu(x_j)|}{n\sum_{i=1}^{n}\theta_\nu(x_i)}.$$

By Equation (2) and given a non-decreasingly sorted list of the bounds of X, we can calculate $Gini(X,\nu)$ in linear time over $|X|$. Next, we show that we do not have to consider all values between L and U.

▶ **Lemma 9.** *The minimum Gini coefficient occurs at a ν-centred assignment where $\nu \in \cup_{i=1}^{n}\{\overline{x_i}, \underline{x_i}\}$.*

Proof. By Corollary 8, we know that the minimum Gini coefficient occurs with a ν-centred assignment. We need to show that there are no local minima in any segment between two consecutive bounds, which, if true, will allow us to disregard any point that is not a bound of one of the variables when searching for the lower bound on Gini. Consider a segment between two consecutive bounds l and u. Since no other bound occurs in between l and u, the three sets of variables in X that will be set to their upper bounds, their lower bounds, and to ν, respectively, will remain constant across all ν-centred assignments when $l \leq \nu \leq u$. Assume these variables in X, and what they are set to, occur sorted such that $B = 1..j-1$ are the indices of the variables below the segment (i.e., where $\overline{x_i} \leq l$), the indices $C = j..j+k-1$ are of the variables that cover the segment (i.e., where $\underline{x_i} \leq l \leq u \leq \overline{x_i}$), and $A = j+k..n$ are the indices of the variables above the segment (i.e., where $\underline{x_i} \geq u$). Using Equation (2), the formula for the Gini coefficient across this segment $G(\nu)$ is hence

$$G(\nu) \equiv \frac{\sum_{i\in B}(2i-n-1)\overline{x_i} + \sum_{i\in C}(2i-n-1)\nu + \sum_{i\in A}(2i-n-1)\underline{x_i}}{n\left(\sum_{i\in B}\overline{x_i} + \sum_{i\in A}\underline{x_i} + k\nu\right)}$$

Let $D = \sum_{i\in B}(2i-n-1)\overline{x_i} + \sum_{i\in A}(2i-n-1)\underline{x_i}$, $N = \sum_{i\in B}\overline{x_i} + \sum_{i\in A}\underline{x_i}$ and $c = \sum_{i\in C}(2i-n-1)$. Differentiating $G(\nu)$ w.r.t. to ν, we get the following by the quotient rule

$$\frac{\partial}{\partial\nu}\frac{D+c\nu}{n(N+k\nu)} = \frac{cn(N+k\nu)-(D+c\nu)nk}{n^2(N+k\nu)^2} = \frac{cN-kD}{n(N+k\nu)^2}$$

Note that the sign of the slope is determined by the numerator, and does not depend on ν. Hence there can be no local minima in the segment. ◀

From the above result, it follows that we need not even consider the real-relaxed case for Gini, as all values ν of importance are integers. Next, we prove that we can use binary chop to find the ν-centred assignment that minimises the Gini coefficient.

▶ **Lemma 10.** *Going though the segments of LBG in increasing order of the sorted end-points, a positive slope is not (directly or indirectly) followed by a negative slope.*

Proof. Let us revisit the derivative of each segment, i.e., $\frac{cN-kD}{n(N+k\nu)^2}$. The sign of its slope is determined by the numerator $cN-kD$. We can rewrite c to a closed form formula because it is an arithmetic sum. We get $k\frac{((2j-n-1)+(2(j+k-1)-n-1))}{2}$ which simplifies to $k(k+2j-n-2)$. Rewriting the numerator we get $k(k+2j-n-2)N - kD$, which we can simplify to $k((k+2j-n-2)N-D)$. Because k is always non-negative, the outer k cannot determine the sign of the slope, only its magnitude, thus we can safely ignore it for our purposes. Ignoring the outer k and expanding N and D we get $\left(\sum_{i\in B}(k+2j-n-2)\overline{x}_i - (2i-n-1)\overline{x}_i\right) + \left(\sum_{i\in A}(k+2j-n-2)\underline{x}_i - (2i-n-1)\underline{x}_i\right)$. This simplifies to $\sum_{i\in B}(k+2j-2i-1)\overline{x}_i +$

Algorithm 2 Only run if not all fixed. $n = |X|$.

```
Require: $B = [\beta_1, \ldots, \beta_{2n}]$ is a sorted array of bounds of $X$

procedure GiniLb($X$, $M$, $g$, $s$)
    $L,\ R \leftarrow 1, 2n$
    while $L < R$ do
        $l \leftarrow L + \lfloor (R - L)/2 \rfloor$
        $r \leftarrow l + 1$
        $G_l \leftarrow Gini(X, \beta_l)$
        $G_r \leftarrow Gini(X, \beta_r)$
        while $G_l = G_r$ do
            if l > L then
                $l, G_l \leftarrow l - 1, Gini(X, \beta_{l-1})$
            else
                $L \leftarrow r$
                break
        if $G_l < G_r$ then
            $R \leftarrow l$
        else
            $L \leftarrow r$
    $m, G \leftarrow R, Gini(X, \beta_R)$
    $Gs \leftarrow \lfloor G \times s \rfloor$
    if $Gs \le \underline{g}$ then return true
    $EX \leftarrow \{[\![x_i \le \overline{x}_i]\!] \mid \overline{x}_i \le \beta_m\} \cup \{[\![x_i \ge \underline{x}_i]\!] \mid \beta_m \le \underline{x}_i\}$
    if $Gs > \overline{g}$ then
        return $[\![g < Gs]\!] \wedge \bigwedge_{l \in EX} l \rightarrow false$
    else
        return $\bigwedge_{l \in EX} l \rightarrow [\![g \ge Gs]\!]$
```

$\sum_{i \in A}(k + 2j - 2i - 1)\underline{x}_i$. Let us denote the first term by T_B and the second term by T_A. The maximum value of $i \in B$ is $j - 1$; thus, all terms of T_B are non-negative; and the minimum value of $i \in A$ is $j + k$; thus, all terms of T_A are non-positive. At any segment where the slope is positive, we must have that $T_B > |T_A|$. Moving to the next segment variables could move from A to the overlap part, or from the overlap part to B, or both. Let us consider two cases: (1) at least one variable moves from A to the overlap and (2) at least one variable moves from the overlap to B.

Case (1): In this case, $|T_A|$ will decrease because at least one negative term will be removed, and k will increase by at least one. And T_B will increase because k will increase by at least one.

Case (2): In this case, $|T_A|$ will decrease because $k + 2j$ will increase by at least one (while k decreases, j must increase the same amount) and T_B will increase because at least one more positive term will be introduced and $k + 2j$ will increase by at least one.

As a result, moving from one segment where $T_B > |T_A|$ to the next, we must maintain that $T_B > |T_A|$. Hence, once the slope is positive it cannot become negative. ◀

▶ **Corollary 11.** *LBG has no local minima that is not a global minima.*

Given this result we can again use binary chop to find the global minimum of $Gini(X, \nu)$.

4.2 Algorithm and Clause Learning Explanations

The algorithm for computing a lower bound on the (scaled) Gini coefficient is given in Algorithm 2. It does a binary chop across the sorted list of bounds of the x variables: $\beta_1, \ldots \beta_{2n}$. L and R hold the left and right bound of indices into this array, for where the *rightmost* minimum lies. We compute the midpoint l of L and R and its neighbour r and compare the Gini values. If they are identical in value, we extend the interval $l..r$ to the left until we find a difference, or fill the entire region from L to r, in which case we chop and set $L = r$. We can incrementally compute $Gini(X, \beta_{i-1})$ from $Gini(X, \beta_i)$ in constant time with a little care. Otherwise we update the L or R to keep the rightmost minimum between them. The loop terminates when $L = R$. We calculate the Gini value G, and its scaled version Gs. Note that we use the *rightmost* ν-value when we find a range of bounds with the same Gini value. If the new lower bound is not stronger than the current bound we return a *true* clause. Otherwise we collect as explanation: all lower bounds where x is less than *or equal to* the ν value β_m, and all upper bounds where x is greater than *or equal to* the ν-value β_m. If the new bound causes a violation we return an explanation for the violation, otherwise we return (an equivalent clause) explanation for the new bound. Overall the algorithm is $O(n \log n)$ since the initial sorting is of this complexity, the while loop can only execute $O(\log n)$ times, and the *Gini* calculations are linear (using Equation (2)).

Note also that the scaling and rounding to integer takes place after the binary chop has terminated. This is important, since not having the best accuracy when calculating the slopes can lead to the algorithm terminating with a solution that is only optimal given the rounding to integer. Using this can then lead to incorrect explanations.

▶ **Lemma 12.** *Algorithm 2 returns a clause which is a consequence of* `gini`*(X,M,g,s).*

Proof. The result clearly holds for the trivial clause *true*. The correctness of the value G computed by Algorithm 1 follows from Lemma 7, Lemma 9 and Lemma 10. We now show that the explanation collected in EX is correct. Assume to the contrary that there is a solution η of `gini`(X,M,g,s) satisfying EX that has Gini coefficient $G' < G$. We can assume η is ν-centred for some ν and some bounds on X (not necessarily the current bounds) by Lemma 7. But then η must set all x_i variables not appearing in EX to ν, otherwise there is an assignment allowed by EX which would be no worse. So we can consider η as defined by ν and the bounds appearing in EX. Hence, it has the same properties as the $Gini(X, \nu)$ but with fewer bounds: importantly it has negatively sloped and flat segments, followed by positively sloped and flat segments. If we move ν smoothly up from β_m then because the slope of the curve $G(\nu)$ defined in Lemma 9 is positive by construction (this is why it is important to take the rightmost point with the least Gini value), and this slope only depends on the bounds in EX, the Gini value for η must increase. If we move ν smoothly from β_m downwards then the slope on $G(\nu)$ defined in Lemma 9 is either negative or zero by construction. Since the slope only depends on the bounds in EX, if it is negative the Gini value of η must increase. If it is zero then the Gini value must stay the same. This holds until we cross a lower bound in $\underline{x_i}$ not in EX, until then the Gini values for η are the same as $Gini(X, \nu)$. But once we cross this bound the sum defined by solution η is smaller than that for $Gini(X, \nu)$ and overall the Gini value must be larger. Contradiction. ◀

5 Experimental Evaluation

For both dispersion constraints we use four configurations: The decomposition (Dcmp); the simple minimal propagator (Simp), which only propagates on the dispersion once all variables X are fixed; the binary chop lower-bound propagator with the proposed (LB) and trivial (LB-TL) nogood learning. The trivial learning simply uses all bounds of all the variables as explanation. We implemented the above propagators in Chuffed and ran the below experiments using MiniZinc [8] on an Intel Xeon 8260 CPU (24 cores) with 268.55 GB of RAM. A single core, 8 GB of RAM, and a 20 minute timeout was allocated for the solving of each instance.

Table 1 Summary of the results of the propagators for the 195 dispersion-only instances.

prob	prop	proved			best			no solution	
		total	first	sole	total	first	sole	total	sole
Spread	Dcmp	38	17	2	43	22	7	127	127
Spread	Simp	25	2	0	85	60	58	**0**	**0**
Spread	LB-TL	38	12	0	59	30	14	**0**	**0**
Spread	LB	**40**	**25**	**3**	**115**	**96**	**71**	**0**	**0**
Gini	Dcmp	33	11	0	37	17	6	97	97
Gini	Simp	25	10	0	**171**	**154**	**144**	**0**	**0**
Gini	LB-TL	40	15	0	42	32	11	**0**	**0**
Gini	LB	**47**	**40**	**6**	26	20	2	**0**	**0**

5.1 Dispersion Only

Let's first start off with a simple problem to get a good understanding of the general performance. We have n variables $x_1, \ldots, x_n \in X$, each with domain $l_i..u_i$. For each i, two integers are uniformly randomly drawn from the range $-n^2..n^2$ (if equal, the second gets redrawn until not equal). The lower number becomes l_i and the higher becomes u_i. For each $n \in \{2, 3, \ldots, 40\}$, we create five instances, resulting in $5 \times 39 = 195$ instances. The goal is to minimise dispersion. Since Gini is ill-defined for negative values, all numbers of all Gini instances are uniformly incremented (if needed) until the least one reaches 1.

For these simple benchmarks using learning with the global propagators simply adds overhead, since the search is so simple that little is repeated. Learning does improve the decomposition though, since it can learn on intermediate variables. We show the results for the learning versions since we expect those to be used in real benchmarks.

The results are shown in Table 1. The table shows, for each method, the *total* number of instances where it *proved* optimality, the number of times the algorithm proved optimality *first* on an instance, and the number of instances where it was the *sole* algorithm to prove optimality. It then shows for all instances the *total* number of instances where the algorithm found the *best* solution of any found, the number of instances where it did this *first* of all algorithms, and the number of times it was the *sole* method to find this best solution. Finally we show the *total* number of instances where the method found *no solution*, and for how many it was the *sole* method to do so. Clearly for `spread`, LB is the best method in all categories. It finds the best solution for all instances, and is fastest on almost all. The decomposition fails on most instances because the sizes of the intermediate values it computes with get quickly too large for the solver to deal with. For `gini`, perhaps somewhat surprisingly, the simple propagator is best at finding solutions. For Gini, the best solutions are usually found early in the search, so the fast simple propagator will find good solutions very quickly compared to more expensive propagators LB and LB-TL. For `gini` the size of intermediate values is smaller, since numbers are not squared. The decomposition is competitive in terms of optimality, but still fails on many instances.

5.2 Job-Shop Scheduling

The dispersion only problems are not complex enough to illustrate why we need learning for dispersion propagators. Let us, instead, run experiments on a type of problem where using a learning solver is often desirable, namely, a scheduling problem.

Consider a job-shop scheduling problem but with multiple agents. The goal is to schedule a set of jobs J on a set of machines M. Each job $j \in J$ consists of one task per machine, where each task can only be processed by a given machine and takes a given amount of time to be processed. A machine can only process one task at a time, and the tasks within each job have to be processed in order (task t must finish before task $t+1$ starts). In addition, we have a set of agents A that submit these jobs. More formally, each job $j \in J$ belongs to exactly one agent $a \in A$.

We consider a limited horizon (after which scheduling jobs is pointless due to zero utility) with optional jobs (because not all jobs will fit within the limited horizon). The utility of a given agent is the machine-hours used by their jobs. We further assume that each agent submits more jobs than can be accommodated. This results in comparable utility functions.

The *objective* of the overall problem is either to maximise efficiency (use as many machine-hours as possible) or minimise dispersion (give similar amounts of machine-hours to each agent), respectively denoted MaxEff and MinDis. In either case, a bound on the other metric is used to ensure some solution quality in the other aspect. For example, giving no one anything is a minimal dispersion schedule, but not very efficient.

We use the large instances (30 or more jobs) of [18] (resulting in 50 instances),[2] and for each instance split the jobs uniformly across n agents, for every n such that each agent has between 10 and 20 jobs (resulting in $20 \times 2 + 20 \times 3 + 10 \times 6 = 160$ instances). We perform this uniform split three times to generate a more diverse instance space (resulting in $160 \times 3 = 480$ instances). We run these on the two configurations (MaxEff and MinDis); the resulting number of instances to test each propagator with is 960.

We first run, with a 20 minute time-out for each instance, a variation of the problem where the objective is to minimise the makespan when all jobs are scheduled. The result of this will be a good indication of how long to set our limited horizon for each instance such that no agent can fit all their jobs. We set the limited horizon for the instance in question to be half of this minimum makespan.

Next, we run, again with a 20 minute time-out for each instance, a variation of the problem where the objective is to maximise machine-hour utilisation given the limited horizon. This will give a good indication of the optimal utilisation possible for each instance. We will then use this number H to calculate our machine-hour utilisation bounds and dispersion bounds. For spread we use scale 1, because the numbers are already large, and for Gini we use scale 10000. For both spread and Gini with MinDis we use $\lceil H \times 0.8 \rceil$ as a lower bound on efficiency, allowing 20% slack on efficiency; for spread with MaxEff we use $\left\lfloor (0.2 \times H/|A|)^2 \right\rfloor$ as an upper bound on spread, allowing a 20% standard deviation from the per-agent average machine-hour utilisation. For Gini and MaxEff we use 2000 as an upper bound on Gini, allowing a 20% inequal Gini distribution.

The cactus plots in Figure 1 show the number of problems solved to optimality at different times for the different variations: using the spread problems on top, the Gini problems on bottom, the MaxEff problems on the left, and the MinDis problems on the right. In general, the MaxEff problems are much harder than the minimisation problems.

Clearly for `spread` the decomposition is very weak, this is because the size of intermediates involved is very large, and often goes outside the range that solvers can handle. The simple propagator is much better, and the full spread propagators much better again. Interestingly our small explanations are only slightly more effective than the trivial explanations.

The results for `gini` are quite different. The decomposition is actually capable of proving optimality more than other methods on the minimisation problems. This is because it can use learning on the intermediate variables in the decomposition. Having a richer language

[2] There are 20 instances with 30 jobs (the first half of which have 15 machines and the rest have 20 machines), 20 instances with 50 jobs (again, the first half of which have 15 machines and the rest have 20 machines), and 10 instances with 100 jobs, all with 20 machines.

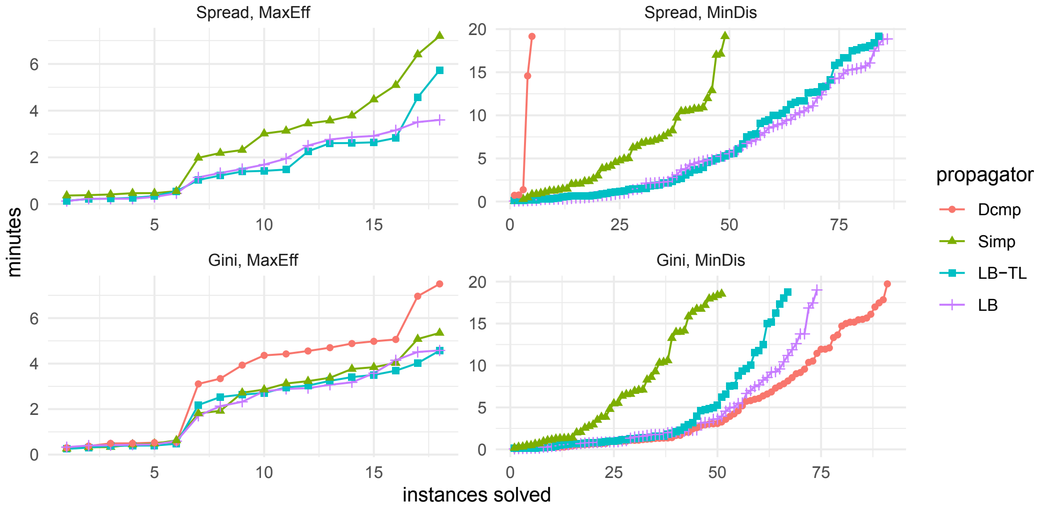


Figure 1 Cactus plot of instances solved to proof of optimality of the propagators for jobshop.

Table 2 Summary of the results of the propagators for the 960 jobshop instances.

prob	prop	proved			best			no solution	
		total	first	sole	total	first	sole	total	sole
Spread	Dcmp	5	3	0	10	8	5	927	904
Spread	Simp	67	14	0	270	137	100	**15**	**0**
Spread	LB-TL	102	48	12	433	250	148	18	**0**
Spread	LB	**104**	**53**	**16**	**695**	**558**	**412**	18	**0**
Gini	Dcmp	**109**	**46**	**15**	264	123	82	104	86
Gini	Simp	69	11	0	249	130	108	**16**	1
Gini	LB-TL	85	25	1	345	197	131	19	**0**
Gini	LB	92	30	2	**620**	**500**	**402**	19	**0**

of learning can often improve proofs of optimality/unsatisfiability. The propagator with our small explanations is the second best. For the MaxEff problems, the decomposition is performing the worst and the other methods are roughly equivalent.

For a more detailed comparison we examine Table 2 which is organised like Table 1. Table 2 clearly illustrates the power of the global propagators. For `spread`, the propagator with our small explanations dominates all methods. For `gini`, while the decomposition is best for proving optimality, overall, it is much weaker than the global propagator, which finds the best solution in over twice as many instances, and almost always finds a solution. It also more clearly illustrates the importance of small explanations, LB is far better than LB-TL in terms of best solutions. We also ran with no learning on all four methods, but it simply timed out on all instances.

6 Related Work

The spread constraint was introduced by [13]. Many previous methods assume that the mean is fixed, e.g. [17]. This is no impediment when the problem is about allocating a given set of resources without mutual exclusions and without uncertainties, but not for the general case where the amount of resource available, or the amount that can be used is not fixed.

[12] introduce a domain consistent propagator for dispersion constraints (including spread) but again assume a fixed/given mean. While an extension for a bounded (variable) mean is proposed, the main criticism is that that algorithm basically runs the domain propagator for each value in the mean, essentially adding another factor to the time complexity. The resulting time complexity is essentially $\mathcal{O}(|X|^3 \cdot |D(X)| \cdot |D(\mu)|)$, which is impractical in many cases. An MDD-formulation of the spread constraint is presented in [11]. They, again, assume a fixed mean. They extend their algorithm to work on variable means, but do not fully answer the question for unknown means. Their use of a probability density function cannot guarantee correct propagation when the mean is fully unknown.

An earlier paper by [13] introduces a bounds consistent propagator for the spread constraint, which covers mean, median, and standard deviation (and thus indirectly variance). They assume finite-domain variables X and bounded continuous domains for mean and standard deviation. They introduce two propagators, one which runs in $\mathcal{O}(|X| + |D(X)|^2)$ time and one (bounds consistent) that runs in $\mathcal{O}(|X|^2)$. One criticism (posed in [17] which have one author in common) is that the latter has never been implemented, is hard to understand, and contains mathematical errors. Thus, we do not compare against it. The former assumes that $|X|$ dominates $|D|^2$, but it does not hold in our use cases. In our jobshop instances, $|X|$ (between 2 and 10) is much smaller than $|D(X)|$ (between 0 and 47186).

Propagators for several other statistical constraints have been introduced in the literature. These include, the deviation constraint [16], mean, median, and weighted mean constraints [2], an occurence balancing constraint [1], the two-sum constraint [7] (which subsumes spread), and many other constraints [15]. [10] provide a review of when to use what balancing/dispersion constraint, depending on the nature of the situation.

Note that in all of the spread propagators above, none generate explanations of their propagation, as required if we want to use them in a learning CP solver [9].

To the best of our knowledge, we are not aware of any propagators for the Gini coefficient. Bounds on the Gini coefficient have been established in the statistics literature [5, 6]; however, these approaches concern obtaining bounds on the actual Gini coefficient of an existing, large population from relatively few samples, and not about filtering possibilities of an unknown population.

7 Conclusion

In this paper we define efficient propagators for propagating the lower bound on variance and the Gini coefficient that also generate explanations of their propagations, allowing them to be used in learning CP solvers. Propagating the lower bound is the critical case to consider for measures of dispersion, since the typical requirement is to bound or minimize dispersion to generate fair solutions. We do not consider propagating the bounds of the X variables being measured, since preliminary experimentation indicated this happens very late in the search tree, and hence cannot lead to significant speedups. Note however that if we stored the explanations generated by the propagator as clauses, some propagation on the X variables could occur.

References

1 Christian Bessiere, Emmanuel Hebrard, George Katsirelos, Zeynep Kiziltan, Émilie Picard-Cantin, Claude-Guy Quimper, and Toby Walsh. The balance constraint family. In Barry O'Sullivan, editor, *CP'14*, pages 174–189. Springer, 2014.

2 Alessio Bonfietti and Michele Lombardi. The weighted average constraint. In Michela Milano, editor, *CP'12*, pages 191–206. Springer, 2012.

3 Geoffrey Chu. *Improving Combinatorial Optimization*. PhD thesis, Department of Computing and Information Systems, University of Melbourne, Australia, 2011.

4 Philip M. Dixon, Jacob Weiner, Thomas Mitchell-Olds, and Robert A. Woodley. Bootstrapping the Gini coefficient of inequality. *Ecology*, 68:1548–1551, 1987. (Erratum in *Ecology*, 69:1307, 1987).

5 Joseph L Gastwirth. The estimation of the Lorenz curve and Gini index. *The review of economics and statistics*, pages 306–316, 1972.

6 Farhad Mehran. Bounds on the Gini index based on observed points of the Lorenz curve. *Journal of the American Statistical Association*, 70(349):64–66, 1975.

7 Jean-Noël Monette, Nicolas Beldiceanu, Pierre Flener, and Justin Pearson. A parametric propagator for pairs of sum constraints with a discrete convexity property. *AIJ*, 241:170–190, December 2016. doi:10.1016/j.artint.2016.08.006.

8 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *CP'07*, LNCS 4741, pages 529–543. Springer, 2007.

9 Olga Ohrimenko, Peter Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14:357–391, 2009.

10 Philippe Olivier, Andrea Lodi, and Gilles Pesant. Measures of balance in combinatorial optimization. *4OR*, June 2021. doi:10.1007/s10288-021-00486-x.

11 Guillaume Perez and Jean-Charles Régin. MDDs are efficient modeling tools: An application to some statistical constraints. In Domenico Salvagnin and Michele Lombardi, editors, *CPAIOR'17*, pages 30–40. Springer, 2017.

12 Gilles Pesant. Achieving domain consistency and counting solutions for dispersion constraints. *INFORMS J. Comput.*, 27(4):690–703, 2015. doi:10.1287/ijoc.2015.0654.

13 Gilles Pesant and Jean-Charles Régin. SPREAD: A balancing constraint based on statistics. In Peter van Beek, editor, *CP'05*, LNCS 3709, pages 460–474. Springer, 2005. doi:10.1007/11564751_35.

14 Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

15 Roberto Rossi, Özgür Akgün, Steven Prestwich, and S. Armagan Tarim. Declarative statistics, 2017. arXiv:1708.01829.

16 Pierre Schaus, Yves Deville, and Pierre Dupont. Bound-consistent deviation constraint. In Christian Bessière, editor, *CP'07*, pages 620–634. Springer, 2007.

17 Pierre Schaus and Jean-Charles Régin. Bound-consistent spread constraint. *EURO J. Comput. Optim.*, 2(3):123–146, 2014. doi:10.1007/s13675-013-0018-8.

18 Eric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.

Plotting: A Planning Problem with Complex Transitions

Joan Espasa
School of Computer Science, University of St Andrews, UK

Ian Miguel
School of Computer Science, University of St Andrews, UK

Mateu Villaret
Department of Computer Science, Applied Mathematics and Statistics, University of Girona, Spain

Abstract

We focus on a planning problem based on Plotting, a tile-matching puzzle video game published by Taito. The objective of the game is to remove at least a certain number of coloured blocks from a grid by sequentially shooting blocks into the same grid. The interest and difficulty of Plotting is due to the complex transitions after every shot: various blocks are affected directly, while others can be indirectly affected by gravity. We highlight the difficulties and inefficiencies of modelling and solving Plotting using PDDL, the de-facto standard language for AI planners. We also provide two constraint models that are able to capture the inherent complexities of the problem. In addition, we provide a set of benchmark instances, an instance generator and an extensive experimental comparison demonstrating solving performance with SAT, CP, MIP and a state-of-the-art AI planner.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Computing methodologies → Planning and scheduling

Keywords and phrases AI Planning, Modelling, Constraint Programming

Digital Object Identifier 10.4230/LIPIcs.CP.2022.22

Supplementary Material *Software (Source Code and Data)*: `https://github.com/stacs-cp/CP2022-Plotting`; archived at `swh:1:dir:8bb3fab72f1db2cac9715651ed05528cd2cf24c8`

Funding This work uses the Cirrus 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: funded by EP/V027182/1.
Mateu Villaret: funded by grant RTI2018-095609-B-I00 (MCIU/AEI/FEDER, UE).

1 Introduction

Automated planning is a fundamental discipline in Artificial Intelligence [14]. Given a model of the environment, a planning problem is to find a sequence of actions to progress from an initial state of the environment to a goal state while respecting some constraints. Examples of planning problems in industry and academia are numerous, such as drilling operations [22], logistics [25] or chemistry [23]. Among other techniques, Constraint Programming has been successfully used to solve planning problems [5, 6]. It is especially suited when the problem requires a certain level of expressivity, such as temporal reasoning or optimality [31, 3].

Herein, we focus on finding optimal solutions for a discrete time and space puzzle, *Plotting*, a puzzle video game published by Taito in 1989 and ported to many platforms. The objective is to reduce a given grid of coloured blocks to a goal number or fewer (Figure 1). This is achieved by the avatar character repeatedly shooting the block it holds into the grid. It is also known as *Flipull* in Japan as well as in versions for the Famicom and Game Boy.

Plotting is naturally characterised as a planning problem, to find a sequence of positions from which to fire such that enough blocks are removed to beat the current scenario. It is of interest because of the complexity of the state transitions after every shot: some blocks are

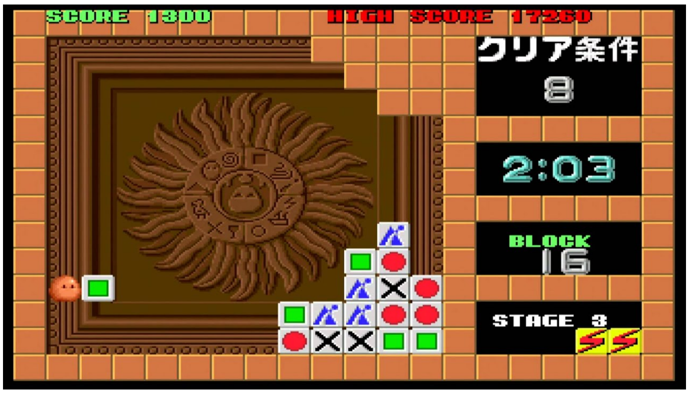


Figure 1 Plotting (Taito, 1989). The avatar is seen on the left, holding a green block. The objective is to reduce the number of blocks in the middle pile up to the goal. In this particular case there are 16 left (see center-right of the image), and the goal is 8 or less (see top-right of image).

affected directly, while others can be indirectly affected by gravity, as explained in Section 3. Modelling the complex dynamics of the game in the de-facto standard modelling language for planning problems, PDDL [17], is difficult, as we will demonstrate. The resulting complexity of the model severely hinders the ability of planning systems to produce a valid plan.

Constraint modelling languages can be used to express planning problems [3, 6, 9, 30]. They are richer than PDDL and, while still a challenge to formulate, permit a more concise representation of Plotting. We present two models of the game in Essence Prime [27] and employ Savile Row [26] to transform them into SAT, MIP, and CP instances for solution.

Plotting is also of interest as an example application in the video games industry, which last year was last year valued at over USD 300 billion [1]. Puzzle games are perennially popular, with other examples similar to Plotting including Puzznic (Taito, 1989) and Lumines (Q Entertainment, 2004). Constraint Programming can provide a tool to assist game designers [16]. Randomly generated levels are commonly used either to save developer time or to generate more content for players. The ability to model game mechanics and solve generated levels provides the opportunity to check if they have a solution, or to get an impression as to how difficult they are [20]. This paper contributes to this growing effort; in addition to the constraint and PDDL models we provide a parameterised instance generator, and an empirical evaluation of the proposed models with a variety of solving back-ends.

2 Background

A *classical planning problem* is a tuple $\prod = \langle F, A, I, G \rangle$, where: F is a set of propositional state variables, A is a set of actions, I is the initial state and G is the goal. A *state* is a variable-assignment (or valuation) function over state variables F, which maps each variable of F into a truth value. An action $a \in A$ is defined as a tuple $a = \langle \mathit{Pre}_a, \mathit{Eff}_a \rangle$, where Pre_a refers to the preconditions and Eff_a to the effects of the action. Preconditions (Pre) and the goal G are first-order formulas over propositional state variables. Action effects (Eff) are sets of assignments to propositional state variables.

An action a is *applicable* in a state s only if its precondition is satisfied in s ($s \models \mathit{Pre}_a$). The outcome after the application of an action a will be the state where variables that are assigned in Eff_a take their new value, and variables not referenced in Eff_a keep their current values. A sequence of actions $\langle a_0, \ldots, a_{n-1} \rangle$ is called a *plan*. We say that the application of a plan starting from the initial state I brings the system to a state s_n. If each action is applicable in the state resulting from the application of the previous action and the finalstate satisfies the goal (i.e., $s_n \models G$), the sequence of actions is a *valid plan.* A planning problem has a solution if a valid plan can be found for the problem.

The Planning Domain Definition Language (PDDL) [17] is the de-facto standard modelling language for planning problems, supported by most planning systems. Its widespread use started thanks to the collaborative efforts and desire of the community to facilitate benchmarking and applications of planning systems. When using PDDL, the user describes the problem in terms of predicates, actions and functions with parameters. In turn, these parameters are instantiated with a set of defined objects.

2.1 Planning as Satisfiability

When a planning problem has a fixed length, such as peg solitaire [19], modelling in a constraint language is simplified to deciding a fixed-length sequence of actions. Otherwise, the modeller must consider how to find a plan of unknown length. There have been various successful approaches to encoding a planning problem into SAT [21, 29] and to CP [6, 30, 3, 24], amongst others. When encoding these problems, it is common in this situation to solve the planning problem by considering a sequence of satisfaction problems ϕ_0, ϕ_1, ϕ_2, $\ldots$, where ϕ_i encodes the existence of a plan that reaches a goal state from the initial state in i steps.

As described in Section 5, in constructing each ϕ herein we take the common approach [6, 19] of formulating a “state and action” constraint model of the planning problem, where we employ decision variables to capture both the state of the puzzle at each time step and the action taken to transform the preceding into the succeeding state. Constraints ensure that when an action is executed, its preconditions hold with respect to the problem variables and its effects are applied to modify the state. Constraints on the variables representing the state of the final step require that the goal conditions are met. Finally, *frame axioms* are made explicit, i.e. constraints specify that if no action has modified a variable, it keeps its value between steps. There are semantics such as the $\forall$ and $\exists$-step [29], or transition-systems [15] that allow more than one action per step. Since we are interested in optimal plans in the total number of actions, we consider sequential plans, i.e., one action per step.

3 Plotting

Plotting is played by one agent with full information of the state, and the effects of each action are deterministic. This situation is common in puzzle-style video games, and similar to pen and paper puzzles [10], some variants of patience like Black Hole [12], and board games such as peg solitaire [19] or the knight's tour [2]. The objective in Plotting is to reduce a given grid of coloured blocks down to a goal number or fewer. This is achieved by the avatar character shooting the block it holds into the grid, either horizontally directly into the grid, or by shooting at the wall blocks above the grid, and bouncing down vertically onto the grid. When shooting a block, if it hits a wall as it is travelling horizontally, it falls vertically downwards. In a typical level, additional walls are arranged to facilitate hitting the blocks from above. Alternatively, if it falls onto the floor, it rebounds into the avatar's hand. The rules for a shot block S colliding with a block B in the grid are a bit more complex:

- If the first block S hits is of a different type from itself, S rebounds into the avatar's hand and the grid is unchanged – a null move.
- If S and B are of the same type, B is consumed and S continues to travel in the same direction. All blocks above B fall one grid cell each.

- If S, having already consumed a block of the same type, hits a block B of a different type, S replaces B, and B rebounds into the avatar's hand.

A simple horizontal shot is depicted in Figure 2. A red block is shot, consuming the two red blocks of the second row and traversing the empty space between them. It replaces the green block, which rebounds to the avatar's hand, ready for the next action. Blocks above the two removed red blocks fall. A more complex shot is depicted in Figure 3, where a green block consumes an entire row of the grid, hits the wall, and continues to consume blocks as it falls until it finds a differently colored block (red). Finally, the block shot replaces the final red block, which rebounds to the avatar's hand. As before, blocks above the consumed green blocks fall. If, after making a shot, the block that rebounds into the avatar's hand is such that there is now no possible shot that can further reduce the grid, we reach a dead end and the block in the avatar's hand is transformed into a wildcard block, which transforms into the same type as the first block it hits. However, in our models we consider the task of finding a solution without reaching any dead end. Each level also begins with the avatar holding a wildcard block.

Considered as a planning problem, Plotting's initial state is the given grid, and there are usually multiple goal states where the grid is sufficiently reduced to meet the target. We abstract the avatar's movement to consider the key decisions: the rows or columns chosen at which to shoot the held blocks. Therefore, the sequence of actions to get us from the initial to the goal state is comprised of individual shots at the grid, either horizontally or vertically.

4 Modelling Plotting in PDDL

As Plotting is naturally characterised as a planning problem, we start by modelling it in PDDL [17], the de-facto standard language for AI planners. Due to its length, the full PDDL model can be found in the supplementary material. PDDL is an expressive and modular modelling language, able to encode many real-life problems with complex dynamics. However, the complexity of its many features resulted in most AI planners lagging behind, supporting only a small core set of features.

To compactly model the sets of state variables F and actions A as described in Section 2, PDDL models use parameterised representations with types. PDDL is *action-oriented*: a PDDL model mainly defines the possible actions at each step. Also for each action, we must define the `precondition` over the state of the previous time step required to perform the action, and the `effect` over the state when that action is performed.

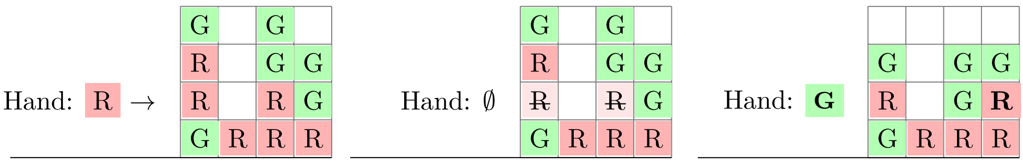


Figure 2 Diagram of a horizontal shot. R and G denote red and green blocks respectively. The initial state is shown on the left figure. The middle figure shows the blocks directly affected: the two light-red crossed out blocks will be removed, and all of the blocks on top will fall downwards. Finally, the right figure shows the resulting state after the shot, having swapped the hand's initial colour for the first one found in the trajectory that is not equal. A vertical shot works similarly.

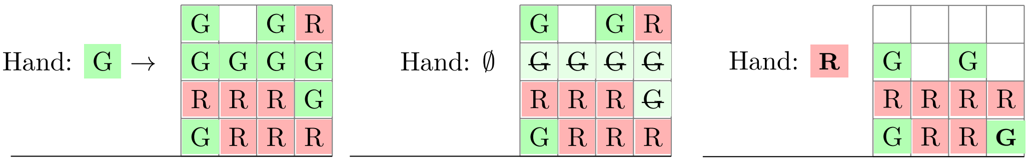


Figure 3 A more complex shot where the firing block reaches the end and goes downwards. Note the top right red block has to fall a variable number of positions (two in this case), depending on the state of the board and the colour of the shot.

4.1 On Numeric Planning

Naturally, one would gravitate towards the PDDL versions for numeric planning to be able to use numeric indexing. In [11], where PDDL is extended with numeric features, it is said:

> Numeric expressions are not allowed to appear as terms in the language (that is, as arguments to predicates or values of action parameters) ...Functions in PDDL2.1 are restricted to be of type $Object_n \rightarrow \mathbb{R}$, for the (finite) collection of objects in a planning instance, *Object* and finite function arity n.

Namely, no action, predicate or function can have a number as a parameter. Sadly, these severe limitations render numeric planning useless for our needs.

In addition, an essential construct in the preconditions and effects of the actions would be the usage of arithmetic to deal with indices of rows and columns. For example, when we remove a block in a given `row` and `col`, if there was a block above it, this block would fall and we would need to refer to its color. As we will see, this can be easily expressed in Essence Prime by arithmetically operating on the indices of the matrix: `grid[row+1, col]`. Unfortunately, since `row` cannot be a number in PDDL, here we are forced to use quantifiers to be able to refer to the "block that is above it" (i.e., its row is equals to `row+1`). Therefore, we must define predicates to simulate some basic arithmetic on indices.

4.2 The PDDL Model

In this section we provide fragments of the model to illustrate the main drawbacks of PDDL for modelling Plotting. The game board is abstracted as a grid of coloured cells. The colour of the cell is the colour of the block it contains, or `null` if empty. Therefore, the full viewpoint (or state F) is the colour of each cell and the colour of the block in the avatar's hand.

To parameterise the actions and the predicates defining the state, we use two types of objects: `colour` and `number`, where `number` is the name of a type used to manually encode the basic required numerical properties. The predicate `hand` has one colour parameter, and encodes if the avatar has a block of the given colour. Given parameters `row`, `col` and `c`, the `coloured` predicate expresses if the block in that row and column has the given colour.

```
(hand ?c - colour)
(coloured ?row ?col - number ?c - colour)
```

Auxiliary predicates such as `islastcolumn` or `isbottomrow` are added for perspicuity and to reduce the use of quantifiers and so the burden on the planner's preprocessor.

```
(isfirstcolumn ?n - number)
(islastcolumn ?n - number)
(istoprow ?n - number)
(isbottomrow ?n - number)
```

Moreover, we need to encode some integer relations as Boolean predicates:

```
(succ ?p1 ?p2 - number)         ; p1 is successor of p2
(lt ?p1 ?p2 - number)           ; p1 is less than p2
(distance ?p1 ?p2 ?p3 - number) ; p3 is p2 - p1
```

These predicates must be defined in each instance file, along with the specific scenario information. For instance, when dealing with a 5 × 5 board we need to state `succ` for every pair of successive numbers between 1 and 5, and `lt` and `distance` for every pair of two numbers $(p1, p2)$ between 1 and 5 such that $p1 < p2$.

Listing 1 Fragment of the action *shoot-partial-row* of the the PDDL model. Note that the `when` operator has two parameters: the condition and the effect.

```
 1 (:action shoot-partial-row
 2     ;; ?r - what row we are shooting at, ?t - the end cell, ?c - the colour we are removing
 3     :parameters (?r - number ?t - number ?c - colour)
 4     :precondition (and
 5         ;; ?col is the successor of ?t with a different colour than ?c
 6         (exists (?col - number)
 7             (and  (succ ?col ?t)
 8                   (not (coloured ?r ?col ?c))
 9                   (not (coloured ?r ?col null))))
10         ...
11         ;; all the blocks up to ?t have either the colour ?c or are null
12         (forall (?col - number)
13             (or  (lt ?t ?col)
14                  (and (= ?col ?t) (coloured ?r ?t ?c))
15                  (or (coloured ?r ?col ?c)
16                      (coloured ?r ?col null)))))
17     :effect (and
18         ;; Change hands colour and the next cell that we cannot remove gets the colour from hand
19         (forall (?nextcolumn - number ?nextcolour - colour)
20             (when
21                 (and (succ ?nextcolumn ?t)
22                      (coloured ?r ?nextcolumn ?nextcolour))
23                 (and (not (coloured ?r ?nextcolumn ?nextcolour))
24                      (coloured ?r ?nextcolumn ?c)
25                      (hand ?nextcolour)
26                      (not (hand ?c)))))
27         ;; Move everything downwards. Consider 2 cases: base case (top row), and general case (rest).
28         (forall (?currentrow ?nextrow ?currentcol - number)
29             (and ;; First, the general case. Any row except the top one
30                 (forall (?currentcolor ?nextcolor - colour)
31                     (when
32                         (and (lt ?currentrow ?r)
33                              (succ ?nextrow ?currentrow)
34                              (or (lt ?currentcol ?t) (= ?currentcol ?t))
35                              ;; We ensure that the cells have the pertaining colours
36                              (coloured ?currentrow ?currentcol ?currentcolor)
37                              (coloured ?nextrow ?currentcol ?nextcolor)
38                              (not (= ?currentcolor ?nextcolor))) ; avoids a contradiction
39                         (and (not (coloured ?nextrow ?currentcol ?nextcolor))
40                              (coloured ?nextrow ?currentcol ?currentcolor)))))))
41                 ; Then, the base case of firing on the top row.
42                 ...))
```

Listing 1 is an excerpt of the action consisting of partially removing blocks of colour `?c` in row `?r` until column `?t`, i.e. not reaching the last column. One of the principal difficulties is in identifying successors and predecessors of particular rows or columns (e.g. Lines 6,12,19,28), which could have been alleviated through support for arithmetic expressions on parameters.

The lack of support for multi-valued variables makes the encoding of some transitions difficult. For example, when changing the colour held by the avatar we must state: *remove previous colour in the hand and set the new colour* (lines 25-26). Multi-valued variables would make this change straightforward. Due to the lack of support for function symbols in the considered PDDL fragment, we must also employ quantification to name specific objects. For instance, the column of the cell next to `?t` (`?nextcolumn`) and its colour (`?nextcolour`) have to be discovered. This quantification is introduced in line 19, and the values of `?nextcolumn` and `?nextcolour` are discovered in lines 20-22 as a condition for the effect to take place.

If we could use function symbols and arithmetic, we could remove variables `?nextcolumn` and `?nextcolour`, changing the `coloured` symbol to a function that, given a row and column, maps to the colour in that cell. Overall, lines 19-26 could theoretically be simplified to:

```
    (assign (hand (coloured ?r (?t + 1))))
    (assign (coloured ?r (?t + 1)) ?c)
```

Unfortunately, as per the previous subsection, functions can not have numeric expressions as parameters. Essence Prime naturally deals with these kinds of statements (see Section 5).

Finally, we must define the initial and goal states for every instance. The initial state is simply stated with a `coloured` statement for each cell. However, the goal state is more complex to express if we do not have arithmetic or aggregate functions to count the number of cells coloured with `null`. In our instances we define the goal as follows. Let g be the maximum allowed number of non-`null` cells in order to satisfy the goal state. We require that there exist g different cells such that any other cell is `null`. For instance, requiring at most 2 non-`null` cells creates the following statement:

```
    (:goal  ;; at most 2 cells are not null, i.e., g=2
        (exists (?x1 ?x2 ?y1 ?y2 - number)
            (and (or (not (= ?x1 ?x2))
                     (not (= ?y1 ?y2)))
                 (forall (?x3 ?y3 - number)
                    (or ; Or is one of cell 1 or cell 2, or is null
                        (and (= ?x1 ?x3) (= ?y1 ?y3))
                        (and (= ?x2 ?x3) (= ?y2 ?y3))
                        (coloured ?x3 ?y3 null))))))
```

The length of this goal is $\Theta(g^2)$, since the g cells must be pair-wise different. Again, this is much simpler to state in a constraint language with, for example, an `atleast` constraint.

5 Constraint Models in Essence Prime

Rendl et al. [28] provide a brief description of an incomplete constraint model of Plotting, as it does not support the difficult case of a shot travelling horizontally all the way through the grid and then continuing to consume blocks in the final column. We present two complete models of the problem, formulated in a state and action style, as noted in Section 2.1. Here, the state is the current grid configuration and the contents of the hand of the avatar, and the single action is a shot along a particular row or column.

5.1 A Common Viewpoint

Our models share a common *viewpoint*, i.e. the choice of variables and domains, which we summarise before describing each individual model.

Each block type is identified with a colour, and the colours are represented by a contiguous range of natural numbers in Essence Prime. Empty grid cells are represented by 0. Step 0 is the initial state, with the action chosen at step 1 transforming the initial state into the state at step 1, and so on. Hence, the parameters and constants for the models are:

```
given initGrid : matrix indexed by[int(1..gridHeight), int(1..gridWidth)] of int(1..)
letting GRIDCOLS be domain int(1..gridWidth)
letting GRIDROWS be domain int(1..gridHeight)
letting NOBLOCKS be gridWidth * gridHeight
letting COLOURS be domain int(1..max(flatten(initGrid)))
letting EMPTY be 0
letting EMPTYANDCOLOURS be domain int(EMPTY) union COLOURS
given goalBlocksRemaining : int(1..NOBLOCKS)
given noSteps : int(1..)
letting STEPSFROM1 be domain int(1..noSteps)
letting STEPSFROM0 be domain int(0..noSteps)
```

We capture the current state of the grid and the contents of the avatar's hand at each time step with a time-indexed 2d array of decision variables and an individual variable per time step respectively. Only one action is possible per time step, which is the decision as to where to fire the block held. Here we introduce a pair of variables per time step, one representing the column fired down (if any) and one representing the row fired along (if any):

```
find fpRow : matrix indexed by[STEPSFROM1] of int(0..gridHeight)
find fpCol : matrix indexed by[STEPSFROM1] of int(0..gridWidth)
find grid  : matrix indexed by[STEPSFROM0, GRIDROWS, GRIDCOLS] of EMPTYANDCOLOURS
find hand  : matrix indexed by[STEPSFROM0] of COLOURS
```

5.2 Common Constraints

The two models also share some constraints on the viewpoint described above, which we describe in what follows. The initial state constrains the 0th 2d array of `grid` to be equal to the parameter `initGrid`. The goal state counts the number of empty grid cells:

```
    $ Initial state:
    forAll gCol : GRIDCOLS .
      forAll gRow : GRIDROWS .
        grid[0, gRow, gCol] = initGrid[gRow, gCol],
    $ Goal state:
    atleast(flatten(grid[noSteps,..,..]), [NOBLOCKS - goalBlocksRemaining], [EMPTY]),
```

Having transformed Plotting into a decision problem that asks if there is a plan with a fixed number of steps, we might take the view that moves that do not alter the state of the puzzle (e.g. firing the held block into one of a different colour) might be used to "pad" a short plan to the given length. This is of little benefit and could lead to redundant search, so we disallow moves that do not progress the solution of the puzzle:

```
    $ Each move must do something useful:
    forAll step : STEPSFROM1 .
      sum(flatten(grid[step-1,..,..])) > sum(flatten(grid[step,..,..])),
```

Care will be necessary with our frame constraints, which we will describe in the context of the two individual models. Any cell unconstrained will be vulnerable to the solver assigning an arbitrary (low-numbered) colour so as to satisfy the sum constraint above.

The other constraint we consider here states that we must fire horizontally or vertically (a shot at the wall blocks above the grid that then bounces down) but not both:

```
    forAll step : STEPSFROM1 . $ Exactly one fp axis must be 0. (XOR, only ONE fired angle)
      (fpRow[step] * fpCol[step]) = 0 /\ (fpRow[step] + fpCol[step]) > 0,
```

5.3 An Action-focused Constraint Model of Plotting

Our two models differ in the way they describe the transition from one state to another via the action selected. We start describing a model that focuses on the action selected and what must therefore be true of the grid at the preceding step (the action's preconditions) and of the grid subsequently (the action's effects). Due to the complexity of the state changes, this model is quite substantial in size and is provided in full in the supplementary material. Herein, we give an overview along with some illustrative fragments of the model. The constraints in this model are divided into two, depending on whether the shot is down a column or along a row. The column shot is simpler, as it only affects the selected column:

```
forAll step : STEPSFROM1 .
  (fpCol[step] > 0) ->
  $ All other columns are untouched.
  (forAll col : GRIDCOLS .
   (col != fpCol[step]) ->
   (forAll row : GRIDROWS . grid[step,row,col] = grid[step-1,row,col])
  ) /\
  $ Must exist a row where grid[step-1,row,fpCol[step]] = hand.
  (exists row : GRIDROWS .
   (grid[step-1,row,fpCol[step]] = hand[step-1]) /\
   $ Everything above is empty or same colour as the hand.
   (forAll above : int(1..row-1) .
     grid[step-1,above,fpCol[step]] = EMPTY \/
     grid[step-1,above,fpCol[step]] = hand[step-1]) /\
   $ Effect is to make everything down to this row empty
   (forAll clear : int(1..row) . grid[step,clear,fpCol[step]] = EMPTY) /\
   ($ Either this is bottom in which case hand remains same.
    (row = gridHeight) /\ (hand[step] = hand[step-1])
    \/
    $ Or the next row down is of a different colour, swaps with hand.
    (grid[step-1,row+1,fpCol[step]] != hand[step-1] /\
     grid[step,row+1,fpCol[step]] = hand[step-1] /\
     hand[step] = grid[step-1,row+1,fpCol[step]] /\
     forAll below : int(row+2..gridHeight) .
       grid[step,below,fpCol[step]] = grid[step-1,below,fpCol[step]]))
  ),
```

The row shot is considerably more complex, since its effects typically include blocks falling as a result of gravity. We must also support a horizontal shot reaching the wall on the right and falling. We sub-divide into three cases: the shot block is exchanged with another in the same row; the block is exchanged with another in the final column, having hit the wall and fallen; and the block travels all the way to the rightmost column and falls to the floor, consuming only blocks of the same colour, resulting in the same colour block returning to the hand. For brevity we show the first of these below. The two remaining can be found in the full model contained in the supplementary material.

```
forAll step : STEPSFROM1 .
  (fpRow[step] > 0) ->
  (exists col : GRIDCOLS .
   $ Preconds: col with a block different from hand.
   ( (grid[step-1,fpRow[step],col] != hand[step-1]) /\
     (forAll left : int(1..col-1) . $Left, empty/hand colour, must exist a block of hand colour.
        grid[step-1,fpRow[step],left] = EMPTY \/
        grid[step-1,fpRow[step],left] = hand[step-1]) /\
     (exists left : int(1..col-1) .
        grid[step-1,fpRow[step],left] = hand[step-1]))
   /\
   $ Effects:
   ($ left: Blocks falling, staying fixed.
    (forAll left : int(1..col-1) .
       $ Everything below is fixed
       (forall below : GRIDROWS .
          (below > fpRow[step]) ->
          (grid[step,below,left] = grid[step-1,below,left])) /\
       (grid[step,1,left] = EMPTY) /\ $ Top row guaranteed to be empty.
       $ Otherwise fall from above.
       ((fpRow[step] > 1) ->
        (forAll above : int(2..gridHeight) .
           above <= fpRow[step] -> grid[step,above,left] = grid[step-1,above-1,left]))
    ) /\
    $ this col: all fixed apart from fprow, which exchanges with the hand
    (hand[step] = grid[step-1, fpRow[step], col]) /\
    (grid[step, fpRow[step], col] = hand[step-1]) /\
    (forAll colBlock : GRIDROWS .
       (colBlock != fpRow[step]) ->
       (grid[step,colBlock,col] = grid[step-1,colBlock,col])) /\
    $ right: all fixed
    (forAll right : int(col+1..gridWidth) .
       forAll colBlock : GRIDROWS .
         grid[step,colBlock,right] = grid[step-1,colBlock,right])))
```

5.4 A State-focused Constraint Model of Plotting

We now describe an alternative model that focuses on the state of the hand and each cell of the grid, how each might change or remain the same, and the valid reasons for doing so. Again, due to its substantial size we give an overview along with some illustrative model fragments. The full model is provided in the supplementary material.

We found it expedient to introduce a time-indexed set of auxiliary variables to this model to capture the distance travelled in the final column when a block is shot horizontally, reaches the wall, then consumes blocks as it falls down the last column. We use these auxiliary variables throughout the model to simplify the statement of the constraints.

```
find wallFall : matrix indexed by[STEPSFROM1] of int(0..gridHeight)
```

The constraints to make the calculation enumerate each possible value for the `wallFall` variable and stipulate what must be true for that value to be valid:

```
forAll step : STEPSFROM1 .
 forAll i : int (1..gridHeight) .
  (wallFall[step] = i)
  =
  (exists row : int(1..gridHeight) .
    (fpRow[step] = row) /\
    $ Travelled to the rightmost column
    (forAll col : int(1..gridWidth) .
      grid[step-1,row,col] = EMPTY \/
      grid[step-1,row,col] = hand[step-1]) /\
    $ Travelled i in the last column
    (forAll underRow : int (row..row+i-1) .
      grid[step-1,underRow,gridWidth] = hand[step-1] \/
      grid[step-1,underRow,gridWidth] = EMPTY) /\
    $ And no more
    ((grid[step-1,row+i,gridWidth] != hand[step-1]) \/
     (row+i > gridHeight)) /\
    $ And consumed a block somewhere, otherwise not a progressing move.
    ((exists col : GRIDCOLS .
        grid[step-1,row,col] = hand[step-1]) \/
     (exists underRow : int(row..row+i-1) .
        grid[step-1,underRow,gridWidth] = hand[step-1]))
  ),
```

The constraints in the state-focused model are subdivided into four cases: The hand is unchanged, a grid cell becomes empty, a grid cell stays the same and grid cell changes colour to something other than empty, which can affect the hand. These are all stated in an if-and-only-if form to ensure that no part of the state (hand or grid) is left unconstrained and therefore vulnerable to the solver assigning arbitrary values.

There are two scenarios leaving the hand unchanged when we require a progressing move. First, firing down a column of the same colour blocks as the block fired. Second, along a row of the same colour, hitting the wall, then consuming everything beneath on the rightmost column before hitting the floor. The `wallFall` variables simplify this second scenario:

```
forAll step : STEPSFROM1 .
  (hand[step-1] = hand[step])
  =
  ( $ Fired down col, hitting wall
    ( (forAll colBlock : GRIDROWS .
        ((grid[step-1,colBlock,fpCol[step]] = hand[step-1]) \/
         (grid[step-1,colBlock,fpCol[step]] = EMPTY)))
    ) \/
    $ Fired row, hitting wall, dropping through hand-colour only. Test by comparing wallFall with fpRow:
    (wallFall[step] = gridHeight-fpRow[step]+1)
  ),
```

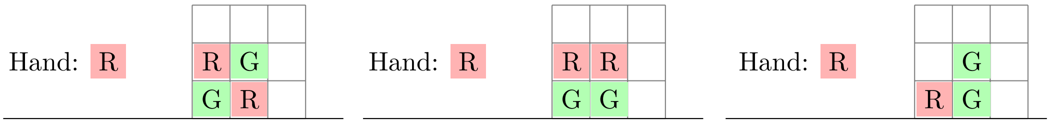
A grid cell remains empty if it was empty at the previous time step. Otherwise it becomes empty if the block that was occupying it is deleted by the chosen shot, or the block that was occupying it falls through the action of gravity. In both of these scenarios we must check that another block has not fallen into this cell and of course we must cater for the fact that in the rightmost column several blocks can be consumed or fall. We present an illustrative fragment below, again exploiting `wallFall`, and refer the reader to the full model for the complete constraint covering this case:

```
forAll step : STEPSFROM1 .
  forAll gRow : GRIDROWS .
    forAll gCol : GRIDCOLS .
      (grid[step,gRow,gCol] = EMPTY)
      =
      ( $ When a cell is EMPTY, it stays EMPTY
        (grid[step-1,gRow,gCol] = EMPTY) \/
        ...
        $ Final Column shot along a row consuming several blocks underneath
        ( $ Only the final column
          (gCol = gridWidth) /\
          $ There was a wallfall - this implies a successful row shot.
          (wallFall[step] > 0) /\
          $ The shot was beneath here
          (fpRow[step] > gRow) /\
          $ Nothing there to fall into here
          (grid[step-1,gRow-wallFall[step],gridWidth] = EMPTY \/
           gRow-wallFall[step] < 1)
        ) \/ ...
      )
```

A grid cell remains unchanged from one time step to the next primarily if it is unaffected by the action chosen. This may be, for example, because a shot was fired down a different column or along a row above. A more subtle scenario is when a block falls down from the current cell, but another of the same colour falls from above to take its place. In all, we have subdivided this case into nine such scenarios, which can be seen in the full model. An illustrative fragment is shown below:

```
forAll step : STEPSFROM1 .
  forAll gRow : GRIDROWS .
    forAll gCol : GRIDCOLS .
      (grid[step,gRow,gCol] = grid[step-1,gRow,gCol])
      =
      ( $ Fired along row above, last col. Something in way on row or last col.
        ( (gCol = gridWidth) /\
          (fpRow[step] != 0) /\
          (fpRow[step] < gRow) /\
          ( (exists rowBlock : int(1..gridWidth) .
             ((grid[step-1, fpRow[step], rowBlock] != EMPTY) /\
              (grid[step-1, fpRow[step], rowBlock] != hand[step-1]))
            ) \/
            (exists colBlock : int(1..gRow-1) .
             ((colBlock >= fpRow[step]) /\
              (grid[step-1, colBlock, gridWidth] != EMPTY) /\
              (grid[step-1, colBlock, gridWidth] != hand[step-1]))
            )
          )
        ) \/
        $ This row or below. Same colour block falls here. Last col.
        ( (gCol = gridWidth) /\
          (fpRow[step] >= gRow) /\
          (wallFall[step] > 0) /\
          (grid[step-1,gRow-wallFall[step],gCol] = grid[step-1,gRow,gCol])
        ) \/ ...
      )
```

Finally, the contents of a grid cell change to something other than empty either as a result of an exchange with the hand or if a different coloured block. Here, we have subdivided into five scenarios, depending on whether a row or column shot was selected, and whether the final column is involved. A fragment is shown below:



(a) A game state with non-interchangeable column shots. **(b)** A game state with non-interchangeable column shots. **(c)** A state that can only lead to dead ends.

Figure 4 Illustrative Plotting game situations.

```
    forAll step : STEPSFROM1 .
      forAll gRow : GRIDROWS .
        forAll gCol : GRIDCOLS .
          ((grid[step,gRow,gCol] != grid[step-1,gRow,gCol]) /\
           (grid[step,gRow,gCol] != EMPTY))
          =
          ( ...
            $ Cell swaps with hand: row then down last col.
            ( $ rightmost col
              (gCol = gridWidth) /\
              $ WallFall implies travel row then col.
              (wallFall[step] > 0) /\
              $ and this cell must be at fpRow+wallFall
              (gRow = wallFall[step] + fpRow[step]) /\
              $ Exchanges with hand
              (hand[step] = grid[step-1,gRow,gridWidth]) /\
              (hand[step-1] = grid[step,gRow,gridWidth]) /\
              $ Which was a different colour
             (hand[step-1] != grid[step-1,gRow,gridWidth])
            ) \/ ...
          )
```

5.5 Symmetry Breaking

Shooting along an empty row has the same effect as shooting down the last column. These two actions are interchangeable, so we can disallow the former:

```
    forAll step : STEPSFROM1 .
      $ Assume bottom row not going to be empty.
      forAll gRow : int(1..gridHeight-1) .
        ((sum gCol : int(1..gridWidth) . grid[step-1,gRow,gCol]) = 0) -> (fpRow[step] != gRow),
```

This remains true if the row is empty except for the last column, and the block in the last column on that row has nothing above it:

```
    forAll step : STEPSFROM1 .
      $ Assume bottom row not going to be empty.
      forAll gRow : int(1..gridHeight-1) .
        ((sum gCol : int(1..gridWidth-1) . grid[step-1,gRow,gCol]) = 0) /\
        ((gRow = 1) \/ (grid[step-1,gRow-1,gridWidth] = EMPTY))
        ->
        (fpRow[step] != gRow),
```

Since they do not interfere with each other in terms of the grid state, it is tempting to think that we can freely permute a sequence of consecutive column shots. This is to ignore the state of the hand, however. Consider Figure 4a we can shoot down the left column, resulting in a green block in the hand, followed by the right column - but not vice versa. If the column “prefix” is the same, as per Figure 4b, we can now shoot down either column. However, after one such shot we could not immediately fire down the other column because the hand would now contain a green block. Therefore, there can be no consecutive column shots (with this pair of columns) to permute. If, however, the columns are monochrome, consecutive column shots are possible, and so we can insist that they are ordered:

```
forAll step : int(1..noSteps-1) .
  forAll gCol : int(1..gridWidth-1) .
    forAll gCol2 : int(gCol+1..gridWidth) .
     $ Monochrome
     (forAll gRow : int(1..gridHeight) .
       ((grid[step-1,gRow,gCol] = EMPTY) \/
        (grid[step-1,gRow,gCol] = hand[step-1])) /\
       ((grid[step-1,gRow,gCol2] = EMPTY) \/
        (grid[step-1,gRow,gCol2] = hand[step-1])))
     -> ( $ If consecutive must be left to right
       fpCol[step] = gCol2 -> fpCol[step+1] != gCol),
```

5.6 An Implied Constraint

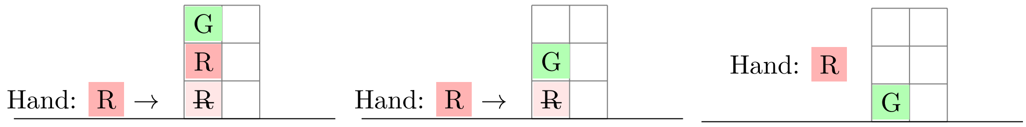
Consider an arbitrary grid with one red block. If that red block is transferred to the avatar's hand then there is no possible move. Hence, this state is only permissible following the final shot in the sequence. If red is already in the hand then the next move must shoot at the red block in the grid, again resulting in another colour in the hand and one red block in the grid, except in a situation like Figure 4c, where we could shoot down the first column, consume the red block and keep red in the hand. Again, however, there will be no possible move. So, the implied constraint is: given a single block of colour c in the grid at time step t, then colour c cannot be in the hand until the goal state (when no further shots are necessary):

```
forAll step : int(0..noSteps-2) .
  forAll colour : COLOURS .
    atmost(flatten(grid[step,..,..]), [1], [colour]) ->
        forAll step2 : int(step+1..noSteps-1) . hand[step2] != colour,
```

It might be conjectured that a similar condition holds for two blocks of a particular colour remaining. Consider an arbitrary grid with two red blocks. When one is hit, having consumed a block of another colour, it appears in the hand. The next shot must be at the other red block. That seems to suggest that red can appear at most once in the hand in the remainder of the sequence. Consider, however, Figure 5a. If we shoot on the bottom row the red block is consumed and the shot block hits the wall, rebounding into the hand, resulting in Figure 5b. Similarly, if we again shoot on the bottom row, the result is Figure 5c. Hence, a counterexample: red appears twice in the hand when there are only two blocks in the grid. Note that the constraints in Section 5.5 and this implied constraint are applicable to models in Sections 5.3 and 5.4 as they both share the same viewpoint.

6 Empirical Evaluation

We have created a dataset of 200 instances using our parameterised instance generator. These have similar properties to the original game levels in terms of size, number of colours and goals: their sizes range from 2 × 4 to 7 × 7, the number of colours range from 2 to 4 and the maximum allowed remaining blocks (goal) range from 5 to 2. In the original game, the scenario sizes range from 4 × 4 to 6 × 6 with 4 colors. The goal objectives also depend on the



(a) State 1.

(b) State 2.

(c) State 3.

Figure 5 With two red blocks remaining, red can appear in the hand twice.

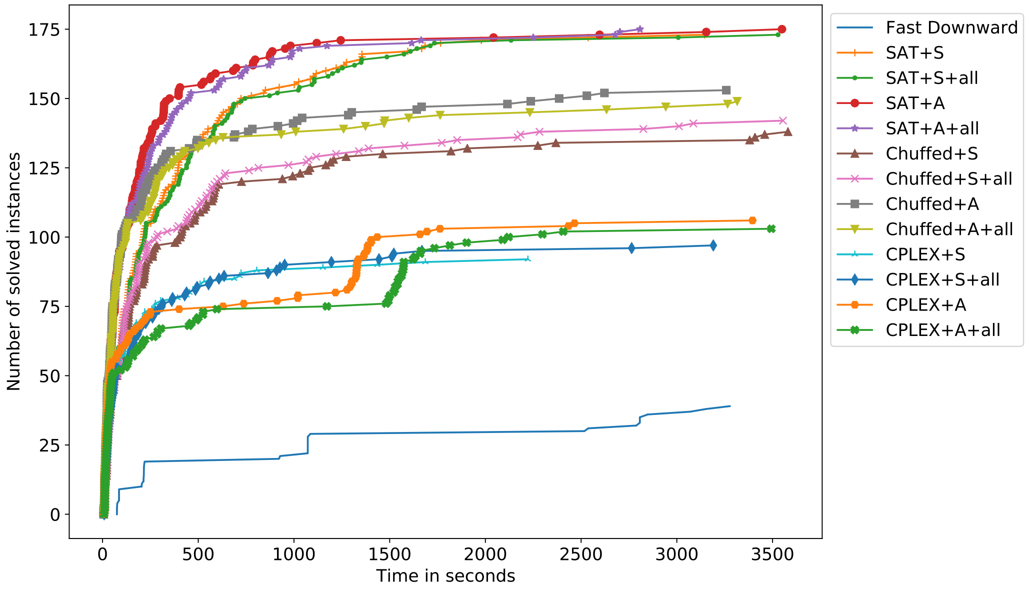


Figure 6 Cumulative instances solved for each model and solver. The *all* variant of the state- (S) and action-focused (A) constraint models includes implied and symmetry-breaking constraints.

difficulty level but usually range from 7 to 3. The only difference in our synthetic instances is that we always allow firing on all rows and columns. Five of our synthetic instances are unsolvable, i.e., you always reach a state where you cannot make a progressing move.

Our experiments were executed on a cluster of compute nodes with two 2.1 GHz 18-core Intel Xeon (Broadwell) processors each. Each process was given a limit of 8GB of memory and 1-hour timeout. We used Savile Row [26] 1.9.1 with three different backend solvers: CaDiCaL [7], Chuffed [8] and CPLEX Optimisation Studio 20.10. We also used the Fast Downward [18] 20.06+ planner. We did consider all planners present in the last IPC and only 9 claimed to support the features required. Of those, 7 were based on the Fast Downward preprocessor and the others crashed when given the instances. We opted to include only results on Fast Downward because pre-processing for all planners based on Fast Downward is the same, and for the successfully pre-processed instances the search time is very small.

Fast Downward is the best-known, supported and reused state-of-the-art planning system, winning the last International Planning Competition (IPC) using some of its portfolio configurations. Its preprocessing module performs sophisticated transformations from PDDL to the more solver-amenable SAS+ format [4], and is reused by many state-of-the-art planners. Still, planning benchmarks do not usually require the expressivity in the language that Plotting does. The extensive use of quantifiers and complex conditional effects in the PDDL model are a heavy burden on the preprocessor, preventing the planner from pre-processing grids greater than 3×3 within the given time-out and memory constraints.

The longest satisfiable instance solved within the time and memory limits has 26 steps. As per Section 2.1, when not using Fast Downward, for each instance we consider a sequence of decision problems from 1 to $(width \times height) - goal$ steps. We generally observe a phase transition around the first satisfiable step. In most cases pre-processing by Savile Row is significant. For the solved instances, an average of 54% of the total time is spent on preprocessing for CPLEX, 51% for SAT and 53% for Chuffed. For some intermediate steps, Savile Row can prove an instance unsatisfiable before encoding it for the backend solver.

Table 1 Number of instances solved and PAR2 score per solver and model. Column *none* is performance without the extra constraints. Columns *de*, *em* and *mo* show the differences in performance with the dead end implied constraint, the empty column and monochrome symmetry breaking constraints respectively. Column *all* shows their combined effect. A decreasing value for the PAR2 score signals that problems are solved faster, and so a negative value is better. For example, CPLEX+A solves more instances when separately adding the *de* and *em* constraints to the base model, but solves less instances when adding *mo* or *all* of them in combination. The PAR2 score summarizes how this affects solving times in all instances.

	#instances					PAR2 Score				
	none	de	em	mo	all	none	de	em	mo	all
SAT+S	174	0	0	0	0	248764	-428	+1129	+1093	+3714
Chuffed+S	139	+5	+5	+1	+4	493458	-34174	-42729	-15553	-28534
CPLEX+S	93	+7	+6	+5	+5	788953	-36517	-32037	-25446	-26264
SAT+A	176	0	-1	-1	0	213866	+1674	+7078	+6875	+3994
Chuffed+A	154	-15	-12	-10	-4	371833	+96809	+76535	+63611	+28808
CPLEX+A	107	+1	+4	-1	-3	719877	-8118	-15288	+10127	+29473

We refer to the action-focused (Section 5.3) state-focused (Section 5.4) as models A and S. Figure 6 shows a cactus plot, considering both with and without additional constraints. The plot clearly splits the solvers in four performance profiles. SAT solves most instances, followed by Chuffed, CPLEX and finally Fast Downward. Comparing models S and A, we see three different behaviours. With SAT, the number of solved instances converges regardless of the model, with model A slightly faster. For Chuffed, there is a clear performance gap between them throughout. CPLEX seems to work better with model S until around the 1500 second mark, where model A overtakes it. Overall, model A performs consistently better.

Table 1 summarises performance with and without the extra constraints. The PAR2 score is equal to the CPU time of the solver when the instance is solved, and 2 times the timeout when the instance is unsolved for any reason. Considering the PAR2 scores, the extra constraints are generally slightly harmful for SAT, with only one exception: the dead end implied constraint when using SAT+S. Chuffed and CPLEX show a notable difference between models: Adding additional constraints to the S model consistently help, while if we do the same for model A it generally hinders solving efficiency.

Breaking symmetries in the PDDL model would require even more involved preconditions. For instance, we must state that when shooting a monochrome column there is no (same-coloured) monochrome column in a precedent position. Unfortunately, preprocessing time in the planner is critical in comparison to solving time. Therefore we have not implemented symmetry breaking in PDDL. The native way of handling these is using the `constraints` PDDL3.0 extension [13], sadly with no support among state-of-the-art planners.

7 Conclusions and Further Work

Although Plotting is a planning problem, we have shown that automated planners cannot deal efficiently with a natural PDDL model. The lack of support for some crucial PDDL features such as multi-valued variables, functional symbols and numeric reasoning makes the modelling of problems with complex transitions a cumbersome and error-prone process.

We have presented alternative models in Essence Prime and, in an extensive empirical analysis supported by a new instance generator, experimentally validated that this approach is efficient using a variety of solving technologies. Although both planning and constraint models are quite involved, since Essence Prime is a more expressive language most key points in the model are easier to encode. Native constructs for Essence Prime to express planning-specific primitives would further aid the encoding of planning problems.

References

1 Accenture. The Global Gaming Industry Value Now Exceeds $300 Billion, New Accenture Report Finds. https://newsroom.accenture.com/news/global-gaming-industry-value-now-exceeds-300-billion-new-accenture-report-finds.htm, 2021. [Online; accessed 2-Feb-2022].

2 Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, Peter Nightingale, and András Z. Salamon. Automatic discovery and exploitation of promising subproblems for tabulation. In *Principles and Practice of Constraint Programming - 24th International Conference, CP*, volume 11008, pages 3–12, 2018. doi:10.1007/978-3-319-98334-9_1.

3 Behrouz Babaki, Gilles Pesant, and Claude-Guy Quimper. Solving classical AI planning problems using planning-independent CP modeling and search. In Daniel Harabor and Mauro Vallati, editors, *Proceedings of the Thirteenth International Symposium on Combinatorial Search, SOCS*, pages 2–10. AAAI Press, 2020.

4 Christer Bäckström and Bernhard Nebel. Complexity Results for SAS+ Planning. *Comput. Intell.*, 11:625–656, 1995. doi:10.1111/j.1467-8640.1995.tb00052.x.

5 Roman Barták, Miguel A Salido, and Francesca Rossi. Constraint satisfaction techniques in planning and scheduling. *Journal of Intelligent Manufacturing*, 21(1):5–15, 2010.

6 Roman Barták and Daniel Toropila. Reformulating constraint models for classical planning. In David Wilson and H. Chad Lane, editors, *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference, May 15-17, 2008*, pages 525–530. AAAI Press, 2008.

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

8 Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed 0.10.4. https://github.com/chuffed/chuffed, 2019 (accessed 03-05-2022).

9 Joan Espasa, Ian Miguel, Jordi Coll, and Mateu Villaret. Towards lifted encodings for numeric planning in essence prime. *Proceedings of the 18th International Workshop on Constraint Modelling and Reformulation (ModRef)*, 2019.

10 Joan Espasa Arxer, Ian P Gent, Ruth Hoffmann, Christopher Jefferson, Matthew J McIlree, and Alice M Lynch. Towards generic explanations for pen and paper puzzles with MUSes. In *Proceedings of the SICSA eXplainable Artifical Intelligence Workshop*, 2021.

11 Maria Fox and Derek Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003. doi:10.1613/jair.1129.

12 Ian P Gent, Chris Jefferson, Tom Kelsey, Inês Lynce, Ian Miguel, Peter Nightingale, Barbara M Smith, and S Armagan Tarim. Search in the patience game 'black hole'. *AI Communications*, 20(3):211–226, 2007.

13 Alfonso Gerevini and Derek Long. Plan constraints and Preferences in PDDL3. Technical report, Technical Report 2005-08-07, Department of Electronics for Automation, University of Brescia, Brescia, Italy, 2005, 2005.

14 Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice.* Elsevier, 2004.

15 Nina Ghanbari Ghooshchi, Majid Namazi, M. A. Hakim Newton, and Abdul Sattar. Encoding domain transitions for constraint-based planning. *Journal of Artificial Intelligence Research*, 58:905–966, 2017. doi:10.1613/jair.5378.

16 Gaël Glorian, Adrien Debesson, Sylvain Yvon-Paliot, and Laurent Simon. The dungeon variations problem using constraint programming. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France*, volume 210 of *LIPIcs*, pages 27:1–27:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CP.2021.27.

17 Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019. doi:10.2200/S00900ED2V01Y201902AIM042.

18 Malte Helmert. The Fast Downward Planning System. *J. Artif. Intell. Res.*, 26:191–246, 2006. doi:10.1613/jair.1705.

19 Christopher Jefferson, Angela Miguel, Ian Miguel, and Armagan Tarim. Modelling and solving english peg solitaire. *Comput. Oper. Res.*, 33(10):2935–2959, 2006. doi:10.1016/j.cor.2005.01.018.

20 Christopher Jefferson, Wendy Moncur, and Karen E Petrie. Combination: Automated generation of puzzles with constraints. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 907–912, 2011.

21 Henry A. Kautz and Bart Selman. Planning as Satisfiability. In *ECAI*, pages 359–363, 1992.

22 Derek Long. Drilling down: Planning in the field. Invited Talk, Twenty-Ninth International Conference on Automated Planning and Scheduling, (ICAPS), Berkeley, CA, USA, 2019.

23 Arman Masoumi, Megan Antoniazzi, and Mikhail Soutchanski. Modeling Organic Chemistry and Planning Organic Synthesis. In *Global Conference on Artificial Intelligence (GCAI)*, pages 176–195, 2015.

24 Ian Miguel, Peter Jarvis, and Qiang Shen. Flexible graphplan. In *ECAI*, pages 506–510, 2000.

25 Tim Niemueller, Erez Karpas, Tiago Vaquero, and Eric Timmons. Planning competition for logistics robots in simulation. In *Workshop on Planning and Robotics (PlanRob) at International Conference on Automated Planning and Scheduling (ICAPS)*, 2016.

26 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.

27 Peter Nightingale and Andrea Rendl. Essence' description. *CoRR*, abs/1601.02865, 2016. arXiv:1601.02865.

28 Andrea Rendl, Ian Miguel, Ian P. Gent, and Peter Gregory. Common subexpressions in constraint models of planning problems. In *Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA*. AAAI, 2009.

29 Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as Satisfiability: Parallel Plans and Algorithms for Plan Search. *Artificial Intelligence*, 170(12-13):1031–1080, 2006.

30 Peter van Beek and Xinguang Chen. CPlan: A Constraint Programming Approach to Planning. In *Sixteenth National Conference on AI and Eleventh Conference on Innovative Applications of AI*, pages 585–590, 1999.

31 Vincent Vidal and Héctor Geffner. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence*, 170(3):298–335, 2006.

Nucleus-Satellites Systems of OMDDs for Reducing the Size of Compiled Forms

Hélène Fargier ✉
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3, France

Jérôme Mengin ✉
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3, France

Nicolas Schmidt ✉
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3, France

Abstract

In order to reduce the size of compiled forms in knowledge compilation, we propose a new approach based on a splitting of the main representation into a nucleus representation and satellite representations. Nucleus representation is the projection of the original representation onto the "main" variables and satellite representations define the other variables according to the nucleus. We propose a language and a method, aimed at OBDD/OMDD representations, to compile into this split form. Our experimental study shows major size reductions on configuration- and diagnosis-oriented benchmarks.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Constraint and logic programming

Keywords and phrases Knowledge representation, knowledge compilation, ordered multivalued decision diagram

Digital Object Identifier 10.4230/LIPIcs.CP.2022.23

Funding The authors gratefully acknowledge the support of the Artificial and Natural Intelligence Toulouse Institute – ANITI. ANITI is funded by the French "Investing for the Future – PIA3" program under grant agreement ANR-19-PI3A-0004.

Acknowledgements We thank anonymous reviewers for the CPAIOR and CP conferences for their numerous comments and suggestions that helped improve the paper.

1 Introduction

Knowledge compilation aims at translating (off line) a problem expressed in some language into a target language in which operations which are important for the application targeted can be performed efficiently [4, 8]. Decision diagrams for instance (OBDDs [3], OMDDs [17, 12], ordered MDDGs [23, 18]) have shown to be a good target language for many problems expressed as constraint satisfaction problem or as CNF, and in particular for product configuration problems [26, 1, 13].

The size of the compiled form being a main criterion in knowledge compilation, its reduction is a major issue in the field. This is magnified by the fact that many information redundancies can be observed. Indeed, caching and the detection of isomorphic nodes allow the detection of equivalent sub-graphs, but not of all redundant information. This last aspect is why, in this paper, we propose a method for reducing the size of the compiled form, up to an additional (but polynomial) computational cost for the handling of queries and transformations. Because targetting interactive configuration problems, we focus our work on OBDD/OMDD [3, 28, 17, 12] representation languages, and show that the method proposed allows a quickest compilation of the original problem, leads to a much smaller compiled form and above all to an important saving in time when the compiled form is exploited.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 23; pp. 23:1–23:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This article is structured as follows. The next section introduces necessary background and notations. The "satellite system" approach we propose is developed in Section 3. We then evaluate the succinctness of this language from an experimental point of view in Section 4. We relate in Section 5 the theoretical concept on which our approach is based to the notion of "definability", as it has been studied in propositional logic.

2 Background and notations

2.1 Representation languages

Consider a finite set $\mathcal{X}$ of variables, each variable x ranging over a finite domain D_x. For any set $X \subseteq \mathcal{X}$, $\vec{x}$ denotes an assignment to the variables from X. D_X is the set of all assignments of X (the Cartesian product of the domains of the variables in X). The concatenation of two assignments $\vec{x}$ and $\vec{y}$ of disjoint subsets X and Y is an assignment to $X \cup Y$ denoted $\vec{x} \cdot \vec{y}$.

We consider functions f of variables from a subset $\mathrm{Scope}(f) \subseteq \mathcal{X}$ to a set $\mathcal{V}$. We write D_f to denote the domain of f, i.e. $D_f = D_{\mathrm{Scope}(f)}$. For any $Z \subseteq \mathrm{Scope}(f)$, $f_{\vec{z}}$ denotes the *restriction* (or *semantic conditioning*) of f by $\vec{z}$, that is, the function on $\mathrm{Scope}(f) \setminus Z$ such that for any $\vec{x} \in D_{\mathrm{Scope}(f)\setminus Z}$, $f_{\vec{z}}(\vec{x}) = f(\vec{z} \cdot \vec{x})$. Slightly abusing notations, if X and Y are two disjoint sets of variables, f is a function such that $\mathrm{Scope}(f) = X$, and $\vec{x} \cdot \vec{y}$ is an assignment of a super set $X \cup Y$ of X, then we write $f(\vec{x} \cdot \vec{y})$ for $f(\vec{x})$.

A representation language over $\mathcal{X}$ w.r.t a valuation set $\mathcal{V}$ is a set of data structures equipped with an interpretation function that associates with each data structure a mapping from $D_{\mathcal{X}}$ to $\mathcal{V}$. This mapping is called the *semantics* of the data structure, and the data structure is a *representation* of the mapping.

▶ **Definition 1** (representation language; inspired by [11]). *A representation language* $\mathtt{L}$ *over* $\mathcal{X}$ *w.r.t* $\mathcal{V}$*, is a 4-tuple* $\langle C_{\mathtt{L}}, \mathrm{Scope}_{\mathtt{L}}, f^{\mathtt{L}}, ||_{\mathtt{L}}\rangle$*, where:*

- $C_{\mathtt{L}}$ *is a set of data structures* ϕ *(also referred to as* $\mathtt{L}$ *representations or "formulæ"),*
- $\mathrm{Scope}_{\mathtt{L}} : C_{\mathtt{L}} \to 2^{\mathcal{X}}$ *is a scope function associating with each* $\mathtt{L}$ *representation the subset of* $\mathcal{X}$ *it depends on,*
- $f^{\mathtt{L}}$ *is an interpretation function associating with each* $\mathtt{L}$ *representation* ϕ *a mapping* $f^{\mathtt{L}}_{\phi}$ *from the set of all assignments over* $\mathrm{Scope}_{\mathtt{L}}(\phi)$ *to* $\mathcal{V}$*,*
- $||_{\mathtt{L}}$ *is a size function from* $C_{\mathtt{L}}$ *to* $\mathbb{N}$ *that provides the size* $|\phi|_{\mathtt{L}}$ *of any* $\mathtt{L}$ *representation* ϕ*.*

Two formulæ ϕ *and* ψ *(possibly from different languages) are equivalent iff they have the same scope and semantics; this is denoted* $\phi \equiv \psi$*.*

In the following, $\mathcal{X}$ is a set of discrete variables and $\mathcal{V} = \{\top, \bot\}$. Given two functions f and g, the assignments $\vec{x}$ of $\mathcal{X}$ such that $f(\vec{x}) = \top$ are said to be "models" (or "solutions") of f and g is said to be a consequence of f (denoted $f \models g$) if any model $\vec{x}$ of ϕ can be extended to a model of ψ: $f(\vec{x}) = \top$ implies $\exists \vec{y} \in D_{\mathrm{Scope}(g)\setminus\mathrm{Scope}(f)}$ such that $g(\vec{x}.\vec{y}) = \top$. Given two $\mathtt{L}$ representations ϕ and ψ, $\vec{x}$ is a model (or "solution") of ϕ iff it is a model of $f^{\mathtt{L}}_{\phi}$, $\vec{x}$ is then said to be consistent with f. ψ is said to be a consequence of ϕ (denoted $\phi \models \psi$) iff any model of ϕ can be extended to a model of ψ (i.e. $f^{\mathtt{L}}_{\phi} \models f^{\mathtt{L}}_{\psi}$).

For any function f from (a subset of) $\mathcal{X}$ to $\mathcal{V}$ and any partition (Y, Z) of $\mathrm{Scope}(f)$, $\exists Z.f$ is the function on Y which maps $\vec{y}$ to $\top$ iff there exist a $\vec{z}$ such that $f(\vec{y}.\vec{z}) = \top$. $\exists Z.f$ is the projection of f on $\mathcal{X} \setminus Z$. Finally, $f \wedge g$ denotes the conjunction of f and g: $(f \wedge g)(\vec{x}) = \top$ iff $f(\vec{x}) = \top$ and $g(\vec{x}) = \top$; and $f \vee g$ denotes their disjunction: $(f \vee g)(\vec{x}) = \top$ iff $f(\vec{x}) = \top$ or $g(\vec{x}) = \top$.

2.2 CSPs

A CSP formula is a set $\phi = \{f_1, \ldots, f_{|\phi|}\}$ of functions f_i (also called "constraints") mapping assignments of subsets of $\mathcal{X}$ to $\{\top, \bot\}$. For any $\vec{x}$ in $D_{Scope(f_i)}$, $f_i(\vec{x}) = \top$ means that $\vec{x}$ satisfies the constraint. A model (a solution) of the CSP is an assignment of $\mathcal{X}$ satisfying all the constraints – ϕ is a representation of the function $f^{CSP}(\phi) = \bigwedge_{f_i \in \phi} f_i$. No assumption is made on the way constraints are represented – it is simply assumed that $f_i(\vec{x})$ can be computed quickly (generally, in linear or constant time).

2.3 Propositional Logic, CNFs

Given a set of *Boolean* variables $\mathcal{X}$, a *literal* over $\mathcal{X}$ is either a variable of $\mathcal{X}$ or the negation of a variable of $\mathcal{X}$. Well formed logical formulae are defined as usual using the logical connectors $\neg$, $\vee$, $\wedge$. For any well formed formula ϕ, f_ϕ^{PROP} obeys the classical semantic. For instance a *clause* is a disjunction $cl_i = l_1 \vee \cdots \vee l_{|cl_i|}$ of literals; Scope(cl_i) is the set of variables on which the literals of cl_i bear; the semantics of cl_i is the Boolean function $f_{cl_i}^{Clause}$ from $D_{Scope(cl_i)}$ to $\{\top, \bot\}$ defined by $f_{cl_i}^{PROP}(\vec{x}) = \top$ iff $\vec{x}$ maps value $\top$ to at least one positive literal of ϕ or value $\bot$ to at least one negative literal of ϕ. Likewise, a CNF over $\mathcal{X}$ is a conjunction $\phi = cl_1 \wedge \cdots \wedge cl_{|\phi|}$ of clauses; then Scope(ϕ) $= \bigcup_{cl_i \in \phi}$ Scope(cl_i) and ϕ is a representation of the function $f^{CNF}(\phi) = \bigwedge_{cl_i \in \phi} f_{cl_i}^{Clause}$ – it is satisfied iff all the clauses of ϕ are satisfied.

2.4 Decision diagrams

A decision diagram (DD) is a directed acyclic graph with a single root node denoted $root(\phi)$ and two leaf nodes labelled with $\top$ and $\bot$ respectively. Non-leaf nodes can be of two kinds, "AND" nodes and decision nodes.

- *Decision* nodes are labelled with variables of $\mathcal{X}$; if v is a decision node labelled with $x \in \mathcal{X}$, then v has as many children w as there are values in D_x, and the edges (v, w) are univocally labelled with the values in D_x. We write $a = label(v, w)$ to indicate that edge (v, w) is labelled with value a. For every $a \in D_x$, $next(v, a)$ will denote the child of v selected by value a, i.e. $next(v, a) = w$ iff edge (v, w) is labelled with value a. $next(v)$ denotes the set of children of v.

  The paths of the DD are often assumed to satisfy the read-once property: no path from the root to the $\top$ leaf node contains a given variable label more than once.
- "AND" nodes are labelled with $\wedge$. An AND node v can have any number of children, and if w is one of them then the edge (v, w) is not labelled;

The scope of a decision diagram is naturally defined as the set of variables that label its decision nodes.

The interpretation function of decision diagrams is defined as follows. Let v be the root node of ϕ. If ϕ contains only one node (v is aleaf), it is necessarily labelled with a constant $c \in \{\top, \bot\}$; then $f_\phi^{DD}(\vec{x}) = c$. If v is an AND node, then $f_\phi^{DD}(\vec{x}) = \bigwedge_{v' \in next(v)} f_{\phi(v')}^{DD}(\vec{x})$. If v is a decision node with label x_i, then $f_\phi^{DD}(\vec{x}) = f_{next(v,a)}^{DD}(\vec{x})$ where a is the value assigned to x_i by $\vec{x}$.

A decision diagram is in reduced form iff all isomorphic subgraphs are merged. The reduction of a decision diagram can always be performed in linear time. We assume in the following that the decision diagrams are in reduced form. Several valuable categories of decision diagrams have been identified:

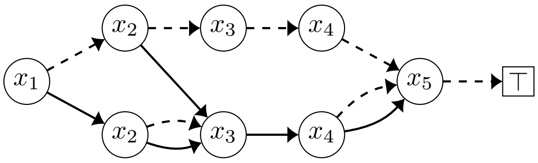


Figure 1 An OBDD equivalent to the logical formula $\phi = \big[(x_1 \wedge x_3) \vee (\neg x_1 \wedge ((x_2 \wedge x_3) \vee (\neg x_2 \wedge \neg x_3 \wedge \neg x_4)))\big] \wedge \neg x_5$; dashed edges are implicitly labelled with value 0, plain edges with value 1.

- A *MDDG* is a decision diagram the AND nodes of which are *decomposable*, meaning that if w and w' are two distinct children of AND node v, then the two sets of variables that appear in the two decision diagrams rooted at w and w' respectively must be disjoint.
- Let $<$ be total order on $\mathcal{X}$. A DD is ordered by $<$ iff for any pair $(v,\ v')$ of decision nodes in ϕ, if v' can be reached from v, then $label(v) < label(v')$ (on a path, the nodes are encountered according to $<$) – as a consequence the DD does not contain any AND node. Such graphs are called *Ordered Multivalued Decision Diagrams* (OMDD). They constitute a generalization of well-known Ordered Binary Decision Diagrams (OBDD), and allow Boolean functions of discrete variables, instead of Boolean variables only.

In the "logical" definition of OMDDs given above, there are two sink nodes labelled with $\top$ and $\bot$, and every node has an outgoing edge for every value of the domain of the variable that labels the node; but when implementing OMDDs, and when drawing them, it is sufficient to implement / draw only the paths that lead to $\top$ – every "dead-end" then corresponds to a path to $\bot$; the sink node labelled $\bot$ is implicit. We adopt this convention for OMDDs throughout. This means that for every node v, if $\text{next}(v) \neq \emptyset$ then v has $\top$ as one of its descendants; if there is no node $next(v, a)$ for a value a in the domain of the variable labeling v, this is equivalent to having $next(v, a) = \bot$.

▶ **Example 2.** Figure 1 depicts an OBDD over $\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5\}$ that is equivalent to the following formula of propositional logic:

$$\phi = \big[(x_1 \wedge x_3) \vee (\neg x_1 \wedge ((x_2 \wedge x_3) \vee (\neg x_2 \wedge \neg x_3 \wedge \neg x_4)))\big] \wedge \neg x_5$$

2.5 Operations

In the following, we will use several basic operations on formulae. Let ϕ and ψ be two representations in a language L.

CD: The *conditioning* of ϕ by the assignment $z = a$ of variable z computes a L representation of $f(\vec{x}) = f^{\text{L}}_{\phi}(\vec{x}.a)$

FO: the *forgetting* of a set of variables $X \subseteq \mathcal{X}$ in ϕ computes a L representation of $\exists \vec{x}.f^L_\phi$

$\bigwedge$BC: the *conjunction* of two L representations ϕ and ψ computes a L representation of the function $f^L_\phi \wedge f^L_\psi$

$\bigvee$BC: the *disjunction* of two L representations ϕ and ψ computes a L representation of the function $f^L_\phi \vee f^L_\psi$

GIC: A value $a \in D_x$ is Globally Inversely Consistent for some L representation ϕ if there exist at least an assignment $\vec{x}$ for which the value of x is a and $f_\phi(\vec{x}) = \top$. Ensuring *Global Inverse Consistency* of a formula consists in computing, for each variable, the set of all its globally inversely consistent values.

Operations CD and GIC are particularly useful in the domain of interactive product configuration, where ϕ represent a configurable product (each model is a feasible product): at each step, the user chooses a variable and the system then computes the set of its globally consistent values. The user then chooses one of these values and the corresponding conditioning is performed.

These two operations can be performed in linear time on decision diagrams – hence their attractiveness as a target language for compilation. Moreover, the forgetting, bounded conjunction and bounded disjunction are tractable when considering OMDD ($O(|\phi|)$ for the former, $O(|\phi| \times |\psi|)$ for the latter two).

2.6 Compilation of CSP/CNF into OMDD/OBDDs

Knowledge compilation aims at translating a formula ϕ expressed in some language into a language in which operations which are important for the application targeted can be performed efficiently – of course, there is no free lunch: there may exist some instances for which this process in not tractable.

In this paper, we consider the compilation of Constraint Satisfaction Problems (resp. CNF) into Ordered Multivalued (resp. Boolean) Decision Diagrams. Several compilers exist in this context, that preserve the set of models of the original representation; that is, if ϕ is the CSP representation of a problem, and ψ the Decision Diagram representation computed, any model of ϕ is a model of ψ and reciprocally.

Existing compilers are either *top-down* compilers or *bottom-up* compilers. Roughly, *top-down* compilers [18, 7, 27, 20, 24] perform a backtrack search of the models, adding the (set of) models reached to the current compiled form. These compilers make use of a SAT or CSP solver.

On the other hand, *bottom-up* compilers [28, 10, 5] start by separately compiling the constraints of the CSP representation, using a common order of the variables; the conjunction of these compiled constraints is then computed, using the $Apply_\wedge$ algorithm [3]. Each bounded conjunction is realized in polytime – the resulting form can be smaller than the former one, but it may also grow (polynomially) at each step. There is thus a risk of explosion when the number of conjunctions is not limited.

3 Nucleus-Satellites System of OMDD

The overall approach that we propose in this paper consists in reducing the size of one OMDD ϕ by extracting the information about some variable y. This information is represented in another formula ϕ_y, and y is deleted from ϕ (ϕ becomes $\exists y \phi$). This can induce a direct and moderate gain on the depth of the OMDD, and an indirect and bigger gain in the width of the structure. In many cases, repeated applications of this process can significantly reduce the size of the overall structure, if the information about the variables deleted from the original OMDD depends on a limited number of variables (in which case, ϕ_y will be small). ϕ_y is called a satellite and the reduced OMDD is a nucleus of the formula. This is illustrated on Figure 2.

Formally, we propose a new language, which we call Nucleus-Satellites System of OMDDs:

3.1 Definition

▶ **Definition 3.** *A Nucleus-Satellites System (NSS) of OMDDs is a triple* $\Phi = (\phi_n, Y_\phi, \{(y, \phi_y) \mid y \in Y_\phi\})$ *where* Y_ϕ *is a set of variables,* ϕ_n *and the* ϕ_y*'s are OMDDs on subsets of* $\mathcal{X}$*, such that:*

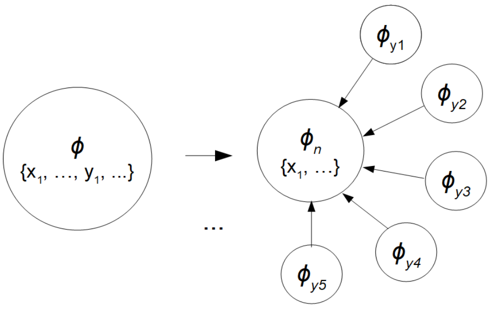


Figure 2 From an OMDD ϕ to a Nucleus-Satellites System.

1. *ϕ_n does not involve any of the y's in Y_ϕ and each $y \in Y_\phi$ appears only in ϕ_y ($\forall y \in Y_\phi$, $y \notin \text{Scope}(\phi_n)$ and $\forall y' \neq y \in Y_\phi, y' \notin \text{Scope}(\phi_y)$);*
2. *ϕ_n and the ϕ_y's obey the same order on $\mathcal{X} \setminus \{y\}$ and y labels the root of ϕ_y;*
3. *for any model $\vec{x}$ of ϕ_n, there exists an assignment $\vec{y}$ of the y's such that $\vec{x}.\vec{y}$ is a model of each ϕ_y.*

ϕ_n is called the nucleus of the system, and the ϕ_y's are its satellites.

A satellite system represents the conjunction of all its element, i.e. for any $\vec{x} \in D_{X \setminus Y_\phi}$ and any $\vec{y} = \vec{y_1}.\dots.\vec{y_m} \in D_{Y_\phi}$:

- $f^{NSS}_\phi(\vec{x}.\vec{y}) = f^{OMDD}_{\phi_n}(\vec{x}) \wedge f^{OMDD}_{\phi_1}(\vec{x}.\vec{y_1}) \wedge \dots \wedge f^{OMDD}_{\phi_m}(\vec{x}.\vec{y_m})$
- $\text{Scope}(\Phi) = \mathit{Scope}(\phi_n) \cup (\bigcup_{y \in Y_\phi} \text{Scope}(\phi_y))$
- $\mathit{size}(\Phi) = \mathit{size}(\phi) + \Sigma_{y_i \in Y_\phi} \mathit{size}(\phi_{y_i})$.

A satellite system can be viewed as a tree of OMDDs [9] of depth 1 where each edge ϕ_{y_i} uniquely defines the value of a variable y_i (not present in the nucleus nor in the other satellites).

Given some input L representation ϕ, we want to compute a Nucleus-Satellites System $\Phi = (\phi_n, Y_\phi, \{(y, \phi_y) \mid y \in Y_\phi\})$ that is much smaller than ϕ but equivalent to it: that is, we want that $f^L_\phi \equiv f^{NSS}_\Phi$. As we shall see shortly:

- Y_ϕ will be a set of variables such that $\exists Y_\phi.\phi$ is easy to compute;
- $\phi_n \equiv \exists Y_\phi.\phi$;
- for each $y \in Y_\phi$, ϕ_y retains enough information about y in order to be able to answer some queries of interest.

Because ϕ and the ϕ_y's are OMDDs, the fact that the nucleus ϕ_n and each satellite ϕ_y obey the same variable ordering (condition 2 in Definition 3) guarantees that the conjunction of these two OMDDs can be performed in quadratic time – this will be important for the online exploitation of the data structure.

In the next section, we describe a method for computing an NSS that is equivalent to, and hopefully much smaller than, an initial OMDD.

3.2 Computing a satellite system

We first show how it is possible to easily recognise, in an OMDD ϕ, some variables that will turn out to be easily forget from ϕ, and are therefore good candidates to be in the satellites.

▶ **Definition 4.** *A node v in an OMDD ϕ is passive iff it has at most one child different from $\bot$. Variable y is passive in ϕ iff each node labelled with y is passive.*

Algorithm 1 Satellisation.

```
Input: OMDD $\phi$;
Output: satellite system equivalent to $\phi$
1. $Y_\phi = \emptyset$; $\mathcal{X}_N = \emptyset$;
2. for all $y \in \mathcal{X}$:
   a. if $y$ is passive in $\phi$: add $y$ to $Y_\phi$;
   b. else: add $y$ to $\mathcal{X}_N$;
3. for all $y \in Y_\phi$:
   a. compute weak definition $\phi_y$ of $y$ in $\phi$
   b. compute a representation of $\exists y.\phi$
   c. $\phi \leftarrow \exists y.\phi$
4. return($(\phi, Y_\phi, \{(y, \phi_y) \mid y \in Y_\phi\})$)
```

In other words, node v labelled by y is passive iff for every pair $a, b \in D_y$, if $next(v, a) \neq \bot$ and $next(v, b) \neq \bot$, then $next(v, a) = next(v, b)$. For instance, x_3 and x_4 are passive in the OMDD of Figure 1.

The set of passive variables in a given OMDD ϕ can be computed with a single traversal of ϕ (simply checking, for each variable y whether all the nodes labelled by y have at most one child different from $\bot$).

▶ **Proposition 5.** *If y is a passive variable of an OMDD ϕ, an OMDD ψ that represents $\exists y.f_\phi$ can be computed in linear time and $size(\psi) \leq size(\phi)$.*

Proof. Let us apply the classical algorithm processing the forgetting of one variable. When forgetting a node labelled by some y, this algorithm performs the disjunction (by a pass of the $Apply_\vee$ algorithm on all the children of this node, except on the $\bot$ node).

By definition if y is passive each node v labelled by y has at most two children: $\bot$ and another node u. So there is no need to perform the $apply_\vee$ stage and node v is directly replaced by node u (all the edges pointing at v now point at u).

So, ϕ can be transformed into a representation of $\exists y.f_\phi$ by replacing every node m labelled with y, by its unique child different from $\bot$. ◀

So we have a way to identify variables that can be easily forgotten in an OMDD of interest. Our next step is to provide a way to retain enough information about such a variable y , in another, hopefully small, OMDD, in order to be able to reason about y.

▶ **Definition 6.** *Given a formula ϕ of some representation language $\mathtt{L}$ over $\mathcal{X}$, and some variable $y \in \text{Scope}(\phi)$, a $\mathtt{L}$ formula ϕ_y over some subset $Z \subseteq \mathcal{X} \setminus \{y\}$* weakly defines [1] *$y$ in ϕ iff $f^{\mathtt{L}}_\phi = (\exists y.f^{\mathtt{L}}_\phi) \wedge f^{\mathtt{L}}_{\phi_y}$.*

We are now ready to describe the computation of the Nucleus-Satellites System that correspond to some input formula ϕ. It is formalized in Algorithm 1. The set of passive variables of the input OMDD ϕ is computed at step 2. Then, at step 3, for every passive variable y, a weak definition ϕ_y for y in ϕ is computed with Algorithm 2, described below; y is then forgotten in ϕ.

[1] As we shall see in section 5, this notion is close to the notion of *definability* as it has been studied in propositional logic.

Note that at the end of Algorithm 1 the main OMDD ϕ only bears on the variables in $\mathcal{X} \setminus Y_\phi$, since all variables in Y_ϕ have been forgotten in ϕ. The succession of forgetting at step 3b of this algorithm never increases the size of the nucleus (Proposition 5), and can significantly reduce the size of ϕ: the height of ϕ is lowered by 1 every time a variable is forgotten; and the recovery of a reduced form (the fusion of isomorphic nodes) that follows can lower its breadth.

After Algorithm 1 has been executed on some OMDD , the "satellite" variables – i.e. those in Y_ϕ – do not appear in the nucleus anymore. Moreover, it can easily be checked that if y, y' are two satellite variables, then y' does not appear in ϕ_y: either y' is below y in the variable ordering (and will thus not appear in the satellite) or the value of y is independent of that of y' (because y' is passive): it will never appear in the y satellite.

We now turn to the computation of the satellites. The main idea is that when a small set Z of variables defines some y in ϕ, ϕ_y should be small. Importantly the size of ϕ_y is bounded by $|D_y| \cdot \mathit{size}(\phi)$, because, as we shall see below, ϕ_y is a disjunction of $|D_y|$ formulas, each of which being no larger than ϕ.

The main loop of Algorithm 2, at step 2, iterates over all values $a \in D_y$: it computes the part of a weak definition of y in ϕ that pertains to value a. For every value $a \in D_y$, ϕ is simplified so as to retain just enough information to decide when an instantiation of the variables in ϕ_n is consistent with $y = a$. A fresh copy of ϕ is made at step 2a. Non-sink nodes below y-nodes are bypassed at step 2b, since they do not influence the consistency of value a for y. Then the ancestors of y nodes, and that are not relevant w.r.t. y and a, are bypassed at step 2c; they are called *undecisive*, see definition 7 below, their parents are redirected to one of their children (function redirect); finally y nodes are bypassed at steps 2d and 2e.

▶ **Definition 7.** *Given an OMDD ϕ over $\mathcal{X}$, given $y \in \mathcal{X}$ and $a \in D_y$, we say that a node v of ϕ is undecisive w.r.t. variable $y \in \mathcal{X}$ and $a \in D_y$ in ϕ if v is not labelled with y and:*

1. $|\operatorname{next}(v)| = 1$*; or*
2. *for all $w \in \operatorname{next}(v)$, w is labelled with y and* $\operatorname{next}(w, a) \neq \bot$*; or*
3. *for all $w \in \operatorname{next}(v)$, w is labelled with y and* $\operatorname{next}(w, a) = \bot$*.*

▶ **Example 8.** Figure 3 describes the computation of a satellite system for the OBDD of Figure 1, with $Y_\phi = \{x_3, x_4\}$. Figure 3a describes in details step 2 for $y = x_4$, $a = 1$: at step 2b, the plain edge $\textcircled{$x_4$}\!\rightarrow\!\textcircled{x_5}$ is redirected to $\boxed{\top}$; at step 2c, there are 3 passive nodes : the two nodes labelled x_3, and the one labelled x_2 on the bottom path, they are bypassed ; finally, at steps 2e and 2d, we bypass variable x_4, keeping only the paths that go through an edge where $x_4 = 1$. Note that x_5 is passive too: the "bottom" variable of an OMDD is always passive, according to our definition. However, in practice, all variables do not have to be "sent into orbit", passive variables disappear from the satellites anyway.

▶ **Proposition 9.** *Given some OMDD ϕ over $\mathcal{X}$, $y \in \mathcal{X}$ passive in ϕ, let ψ_y be the OMDD returned by Algorithm 2 when called with ϕ, y.Then $f_\phi = (\exists y.f_\phi) \wedge f_{\psi_y}$.*

The proof of the proposition is based on the following lemma. Its proof is in the appendix, and shows that equation (I) below is an invariant of the main loop in Algorithm 2.

▶ **Lemma 10.** *Given OMDD ϕ, y passive in ϕ, $a \in D_y$, if ψ_a is the OMDD computed at steps 2a to 2e in Algorithm 2 then*

$$\phi \wedge (y = a) \models \psi_a \quad \textit{and} \quad \psi_a \wedge (y = a) \wedge \exists y.\phi \models \phi \tag{I}$$

Algorithm 2 Computation of a weak definition.

```
Input : OMDD $\phi$; $y$ passive in $\phi$ and s.t. every path from root to $\top$ has a $y$-node;
Output : OMDD $\psi$ s.t. $\phi \wedge (y = a) \models \psi$ and $\psi \wedge (y = a) \wedge \exists y.\phi \models \phi$.
1. $\phi_y \leftarrow \bot$;
2. for every $a \in D_y$ do:
   a. $\psi_a \leftarrow$ a fresh copy of $\phi$;            // nodes below $y = a$ edges are bypassed
   b. for every node $v$ labelled with $y$, if next$(v, a) \neq \bot$: next$(v, a) \leftarrow \top$;
   c. while there is some undecisive node w.r.t. $y, a$ in $\psi_a$ do:
   //nodes not “relevan” w.r.t. which models are possible when $y = a$ are bypassed;
      i. $v \leftarrow$ a node of $\psi_a$ undecisive w.r.t. $y, a$;
      ii. $w \leftarrow$ some node $\in$ next$(v)$;
      iii. redirect$(v, w)$;
   //$y$ nodes are bypassed, keeping only information pertaining to value $a$
   d. for every node $v$ labelled with $y$ s.t. next$(v, a) = \top$: redirect$(v, \top)$;
   e. for every node $v$ labelled with $y$ s.t. next$(v, a) = \bot$: redirect$(v, \bot)$;
   f. delete from $\psi_a$ every node that is not accessible from the root anymore;
   g. $\phi_y \leftarrow \phi_y \vee (y = a \wedge \psi_a)$; //compute disjunction of diagrams computed for all $y$ values
3. return $\phi_y$.
Uses function redirect$(v, w)$: for every $u, b$ such that next$(u, b) = v$: next$(u, b) \leftarrow w$.
```

Proof of the proposition. Let $\psi_y = \bigvee_{a \in D_y}(y = a \wedge \psi_a)$. We must prove that $\phi \models \psi_y$ and that $\psi_y \wedge (\exists y.\phi) \models \phi$. Consider some assignment $\vec{x}$ of $\mathcal{X}$, and let a be the value assigned to y in $\vec{x}$. We know, from lemma 10, that $\phi \wedge (y = a) \models \psi_a$ and $\psi_a \wedge (y = a) \wedge \exists y.\phi \models \phi$.

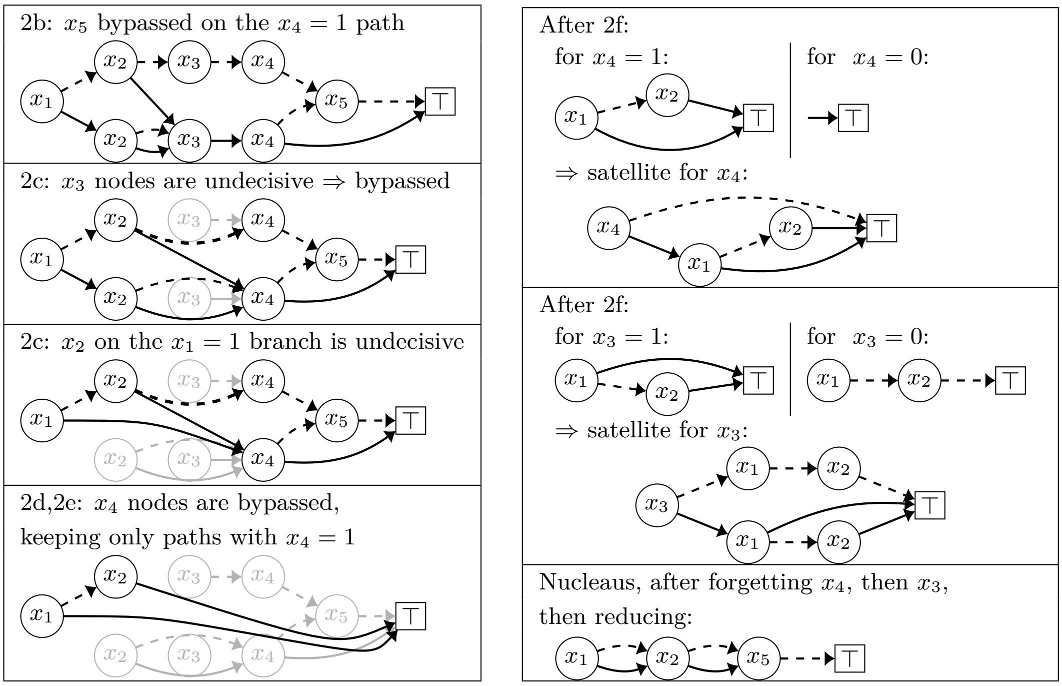
Suppose first that $f_\phi(\vec{x}) = \top$: $\phi \wedge (y = a) \models \psi_a$, $f_{\psi_a}(\vec{x}) = \top$, thus $f_{\psi_y}(\vec{x}) = \top$. For the converse, suppose that $f_{\psi_y}(\vec{x}) = f_{\exists y.\phi}(\vec{x}) = \top$. Then $f_{y=a'}(\vec{x}) = \bot$ for every $a' \in D_y$ with $a' \neq a$, thus it must be the case that $f_{\psi_a}(\vec{x}) = \top$, hence, because of equation (I), $f_\phi(\vec{x}) = \top$. ◀

In practice, making a fresh copy of the main OMDD ϕ at step 2a of Algorithm 2 is not efficient (nodes are created in the unique tables that will be destroyed immediately). Our implementation, used for the experiments described in Section 4, starts from an empty OMDD for ϕ_a, performs a bottom-up traversal of ϕ, starting at the y nodes, and adds decisive nodes (the ones that are not undecisive) to ϕ_a as they are encountered. Decisive nodes are recognised with some colouring scheme applied during this bottom-up traversal of ϕ.

3.3 Conditioning and maintaining GIC of an NSS of OMDDs

The application we target, interactive product configuration, mainly relies on two operations, the conditioning of one variable by the user and the maintaining of the global inverse consistency (GIC): if ϕ represents the current set of possible products, and if value a is chosen by the user for some currently unassigned variable y, then some representation of $\phi \wedge (y = a)$ must be computed that satisfies global inverse consistency (so that the user cannot choose, for the next assignment, a value that cannot lead to a feasible product). In the case of *partial conditioning*, the set of admissible values for y is restricted to, say, $\{a_1, \ldots, a_k\}$, and some representation of $\phi \wedge (y = a_1 \vee \ldots \vee y = a_k)$ must be computed and GIC restored.

By definition the GIC property holds for OMDDs. When constructing an OMDD, or when applying some transformation on it ($\bigwedge$BC, CD, etc), GIC is ensured during the reduction phase, by suppressing inconsistent values from the domains of their respective variables.


2b: x_5 bypassed on the $x_4 = 1$ path
2c: x_3 nodes are undecisive ⇒ bypassed
2c: x_2 on the $x_1 = 1$ branch is undecisive
2d,2e: x_4 nodes are bypassed,
keeping only paths with $x_4 = 1$
After 2f:
for $x_4 = 1$:
for $x_4 = 0$:
⇒ satellite for x_4:
After 2f:
for $x_3 = 1$:
for $x_3 = 0$:
⇒ satellite for x_3:
Nucleaus, after forgetting x_4, then x_3,
then reducing:
x_1
x_2
x_3
x_4
x_5
⊤

(a) Step by step exec. of step 2 for $x_4 = 1$.

(b) Satellites for x_3, x_4 and nucleus.

Figure 3 Execution of Algorithm 2 on the OBDD of figure 1, with $Y_\phi = \{x_3, x_4\}$.

A satellite system $(\phi, Y_\phi, \{(y, \phi_y)|y \in Y_\phi\})$ returned by Algorithm 1 has the GIC property, because it is logically equivalent to the input OMDD, which has the GIC property, and no new value has been introduced for any variable during satellisation.

Now, consider a satellite system that satisfies GIC, and a variable y on which a conditioning (possibly partial) must be performed :

- if $y \in Y_\phi$: conditioning is first performed on the satellite ϕ_y; then a new nucleus must be computed which is a representation of $\phi_n \wedge \phi_y$.
- if $y \notin Y_\phi$: conditioning is only performed on the nucleus ϕ_n.

In both cases, the nucleus has been modified, so GIC may then be lost for the satellite variables; in order to restore it, one can perform a "blank" computation of $\phi_y \wedge \phi_n$ for each ϕ_y (where ϕ_n is now the conditioned nucleus), without returning a new OMDD but just to check which values of y are not "GIC" anymore, in order to remove them from ϕ_y.

Now, recall that the computation of the conjunction of two OMDDs takes time at most quadratic in their size [3, 8]. As a consequence, the worst-time complexity of the conditioning of a satellite system is not linear in its size but quadratic. However, if the satellisation significantly reduces the size of the nucleus and leads to small satellites, then the effective time taken to perform conditioning is reduced too. Furthermore, recall that the size of a nucleus is necessarily smaller than the one of the OMDD equivalent to the full NSS. So the conjunction of a nucleus and a satellite leads to a new nucleus, and the maximal space taken by the conjunction of the original nucleus and the satellite cannot be higher than the one of the OMDD obtained if no satellisation were to be performed. This is because the nucleus is identical to the original OMDD except for the nodes corresponding to the satellites variables, that are passive in the original OMDD and can be forgotten by just by-passing them; the satellites are also obtained by bypassing irrelevant nodes.

Table 1 Configuration benchmarks – size (number of nodes) of the OMDD, nucleus-satellites system of OMDD and MDDG representation of the configuration instances. Column “Biggest” provide the maximal size reached during compilation process.

CSP	OMDD			nucleus-satellites system of OMDD					MDDG
	final	biggest	time	Nucleus	Satellites	sum	biggest	time	size
Small	**321**	328	0.05s	13	58	**71**	73	0.04s	**22**
Medium	**829**	941	0.6s	81	118	**199**	299	0.5s	**64**
Big	**13,916**	14,312	10s	1,475	220	**1,695**	1,723	8s	**2,552**
Master	**41,190**	42,343	13s	2,495	397	**2,892**	4,601	10s	**4,129**
Megane	**146,295**	150,506	18s	1,576	753	**2,329**	5,002	4s	**10,922**

4 Experimental results

In order to evaluate the efficiency of the approach, we have implemented a bottom-up OMDD compiler and an NSS bottom-up compiler which computes directly a nucleus-satellites system from a CSP given as input. The passive variables are detected on the fly, as early as possible during the compilation process, that is to say, as soon as they do not appear in any remaining uncompiled constraint. This method reduces the size of the maximal memory needed (recall that in a bottom-up approach, the size of the current data structure may increase and decrease with the addition of new constraints) – and as a side effect, the compilation time. This method leads to the same final NSS as the naive one (building the full OMDD first and satellizing the passive variables in a second step). Only the compilation time and maximal memory occupation may differ.

The following experiments are based on two families of benchmarks, configuration benchmarks[2], one the one hand, and diagnosis benchmarks[3], on the other hand. More precisely, we have compiled each instance (i) as an NSS and (ii) as an OMDD, and we have measured the sizes in terms of number of nodes, as this number is representative of the size of the diagram. We also measured the CPU time used by each compilation. Each instance has also be given to the CN2MDDG top-down compiler [18, 21] as a base line for the evaluation in terms of size spatial evaluation of the compiled form. Compilation times with CN2MDDG seem irrelevant here (different programming language C++ vs Java, valued vs non valued compilation, different programmers...) The experiments were performed on an Intel(R) Core(TM) i5-8265U CPU 1.60GHz 1.80 GHz, with 32Go of RAM.

4.1 Product configuration benchmarks

Product configuration benchmarks are CSPs representing real products (car) provided by the french car manufacturer Renault.

Table 1 gives the results on configuration instances: it shows a good spatial efficiency of nucleus-satellites systems. The bigger the problem gets, the more efficient they seem to be compared to OMDD. For example the size is divided by 5 on smaller instances and by 50 on bigger ones. This reduction in size makes it competitive with the MDDG language and even smaller on some benchmarks.

[2] https://www.irit.fr/~Helene.Fargier/BR4CP/benches.html

[3] http://www.cril.univ-artois.fr/KC/benchmarks/cnf/circuit.tgz

Table 2 Diagnosis benchmarks – size (number of nodes) of the OMDD, nucleus-satellites system of OMDD and MDDG representation of the configuration instances. Column “biggest” provides the maximal size reached during compilation process.

CNF	OMDD			nucleus-satellites system of OMDD					MDDG
	final	biggest	time	nucleus	satellites	sum	biggest	time	size
s344	**197,284**	293,972	59s	3,838	1,620	**5,458**	168,611	33s	**215**
s400	**23,014**	70,830	13s	1,340	2,341	**3,681**	40,235	9s	**429**
s444	**20,485**	27,081	5s	1,969	1,885	**3,854**	17,666	3s	**325**
s420	**35,900**	1,040,643	155s	5,697	1,653	**7,350**	348,317	78s	**316**
c499	**2,117,382**	2,117,382	101s	62,746	56,749	**119,495**	167,674	52s	23,424,571
s938	**Memory out**	>3,000,000		3,668	6,617	**10,285**	237,570	182s	**669**

4.2 Diagnosis benchmarks

As to diagnosis benchmarks (see Table 2), the NSS approach leads to a huge gain in time and space (one order of magnitude) with respect to the pure OMDD approach. Moreover, the NSS compiler makes it possible to compile a benchmark that cannot be compiled under the OMDD form (out of memory) – it should be noticed that the NSS representation obtained on this instance is much smaller (several orders of magnitude) than the OMDD built when the OMDD compiler ran out of memory.

4.3 Exploitation of the compiled form

Finally, since the goal of compilation is to be able to perform some operations on the compiled form, the present section compares the performances of OMDD and nucleus-satellites systems on a protocol of product configuration [2]. Namely, the compiled form is submitted to a sequence of succession of variable conditioning, each followed by a GIC closure. Each conditioning represents a choice made by a user on the product : at each step, the user chooses whichever variable and assigns a value to this variable. For each variable, the GIC closure then suppresses every non-consistent value of its domain. Since this kind of operation is done online by a user, a quick answer is necessary.

For the experiment[4], at every step, variables were chosen randomly among variables that still have at least 2 consistent values. Time differences between various sequences of user choices where very small. We experimented with the configuration protocol on all car instances, and obtained similar results in terms of comparison between OMDDs and NSSs. The results reported in Figure 4 for the “*big*” instance show that despite the necessity of additional computations induced by nucleus-satellites representation when processing a CD operation, there is a global gain in CPU time. The reduction in size obtained by the use of an NSS largely compensates the additional processing.

Note that our compiler can handle partial conditioning. We ran experiments on this point, and did not notice sizeable difference in the response time.

5 Related work

The idea of extracting, from some initial formula ϕ, information about a particular y has been studied in propositional logic with the notions of *functional dependency* [14, 15, 16] and *definability* [22]. Definability has recently been used by [19] to facilitate model counting in propositional logic.

[4] We did not experiment the configuration protocol with MDDGs since CN2MDDG is not a solver.

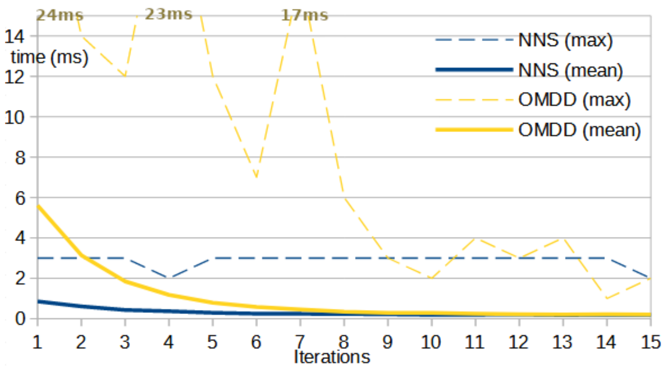


Figure 4 Processing time (in ms) at each iteration (conditioning + GIC closure) during a product configuration protocol (on 1000 tries) with the instance "*big*", compiled as an OMDD (in yellow) or as an nucleus-satellites system (in blue).

▶ **Definition 11** ([22]). *Let ϕ be a formula of propositional logic, $Z \subseteq \mathcal{X}$, $y \in \mathcal{X} \setminus Z$, then ϕ (explicitly) defines*[5] *y in terms of Z if and only if there is a formula ψ of propositional logic with* Scope$(\psi) \subseteq Z$ *such that $\phi \models \psi \leftrightarrow y$. ψ is then called a definition of y on Z in ϕ.*

The following result shows that definability in the sense of [22] is a sufficient condition to build a Nucleus-Satellites System.

▶ **Proposition 12.** *If ϕ is a propositional formula on $\mathcal{X}$, and if ψ is a definition of y on Z in ϕ, then $\phi \equiv \exists y.\phi \wedge (y \leftrightarrow \psi)$.*

Proof. It is well-known that $\phi \models \exists V.\phi$ for any set of variables V, thus $\phi \models \exists y.\phi$. And $\phi \models y \leftrightarrow \psi$ by definition of a "definition". Suppose now that $m \models \exists y.\phi \wedge y \leftrightarrow \psi$. Let m' be the interpretation identical to m except that $m' \models \neg y$ if and only if $m \models y$. Since $m \models y \leftrightarrow \psi$, $m \models y$ if and only if $m \models \psi$, if and only if $m' \models \psi$ since m and m' give the same interpretation to all variables that appear in ψ. Thus $m' \models \neg y$ iff $m' \models \psi$, or, equivalently, $m' \models \neg y \leftrightarrow \psi$, or, equivalently, $m' \models \neg(y \leftrightarrow \psi)$. But by assumption $\phi \models y \leftrightarrow \psi$, thus $m' \models \neg\phi$. But, since $m \models \exists y.\phi$, it must be the case that $m \models \phi$ or $m' \models \phi$. Thus $m \models \phi$. ◀

However, definability, as studied in propositional logic, is not guaranteed; whereas it is always possible to extract enough information about a variable from a given formula in order to "satellize" it. The next example illustrates this.

▶ **Example 13.** Consider the propositional logic formula ϕ of Example 2:

$$\phi = \big[(x_1 \wedge x_3) \vee (\neg x_1 \wedge ((x_2 \wedge x_3) \vee (\neg x_2 \wedge \neg x_3 \wedge \neg x_4)))\big] \wedge \neg x_5$$

Considering the OBDD of Figure 1 equivalent to ϕ, it is easy to check that x_4 is not definable in the sense of [22] in ϕ: when $x_1 =$ false and $x_2 =$ true, x_4 can be true but can also be false.

On the other hand, consider the formulas $\phi_y = (x_4 \rightarrow (x_1 \vee x_2)) \wedge (\neg x_4 \rightarrow \top)$ and $\psi = \big[(x_1 \wedge x_3) \vee (\neg x_1 \wedge ((x_2 \wedge x_3) \vee (\neg x_2 \wedge \neg x_3)))\big] \wedge \neg x_5$. It is easy to check that $\phi \equiv \phi_y \wedge \psi$.

[29] propose a similar approach, called *macro extraction and expansion*, for optimising the size of BDDs used for symbolic model checking. Their experimental results also indicate important gains when using this optimisation.

[5] [22] also introduce a notion of *implicit definability*, but they prove that, because of the projective Beth's theorem, both notions are equivalent in proposition logic.

6 Conclusion

This paper has proposed a method that allows to detect, on an OMDD, a set of variables that can be defined apart in several satellites, according to another common set of variables called the nucleus. We experimentally observe that it can lead to huge reductions of size and allows the compilation of benchmarks that could not be compiled as classical OMDDs.

Nucleus-satellites systems have to be studied further. First, satellites could also define a variable only partially (for example define and forget only passive nodes of a variable, and keep active nodes in the nucleus; the satellite would only be used when the variable is missing on path). A satellite could also represent a group of variables. With additional online computing, satellites could use variables defined in an other satellite to define a new variable, and create a satellite of "higher degree".

Finally, the approach can directly apply to any sub-languages of the *d*-DNNF family for which the operation of bounded conjunction is tractable under some conditions, e.g. structured d-DNNFs [25] or Sentential Decision Diagrams [6] – this is possible when (i) the order is computed on the basis on the original CSP and (ii) the operations targeted (here, the conditioning) do not modify the constraint graph. The question of the efficiency of satellite system based on other languages – and in particular of satellite systems of MDDG – is less easy to address; the question of the on line conditioning indeed becomes more tricky, although such structures can be easily defined and satellisation algorithms could be developed (the definition remain quasi unchanged and the notion of passive variable still applies).

References

1 Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic CSPs - Application to configuration. *Artificial Intellligence*, 135(1-2):199–234, 2002.

2 Jean-Marc Astesana, Laurent Cosserat, and Hélène Fargier. Constraint-based vehicle configuration: A case study. In *ICTAI 2010*, pages 68–75. IEEE Computer Society, 2010.

3 Randall E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers (TC)*, 38.8:677–691, 1986.

4 Marco Cadoli and Francesco M. Donini. A survey on knowledge compilation. *AI Communications*, 10(3-4):137–150, 1997.

5 Arthur Choi and Adnan Darwiche. Dynamic minimization of sentential decision diagrams. In *AAAI'2013*, pages 187–194, 2013.

6 A. Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *IJCAI'2011*, pages 819–826, 2011.

7 Adnan Darwiche. New Advances in Compiling CNF into Decomposable Negation Normal Form. In *ECAI 2004*, pages 328–332, 2004.

8 Adnan Darwiche and Pierre Marquis. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research (JAIR)*, 17:229–264, 2002.

9 Hélène Fargier and Pierre Marquis. Knowledge Compilation Properties of Trees-of-BDDs, Revisited. In *IJCAI 2009*, pages 772–777, 2009.

10 Hélène Fargier, Pierre Marquis, and Nicolas Schmidt. Semiring Labelled Decision Diagrams, Revisited: Canonicity and Spatial Efficiency Issues. In *IJCAI 2013*, pages 884–890, 2013. URL: `http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6623`.

11 Goran Gogic, Henry Kautz, Christos Papadimitriou, and Bart Selman. The Comparative Linguistics of Knowledge Representation. In *IJCAI'1995*, pages 862–869, 1995.

12 Tarik Hadzic, Henrik Reif Andersen, John N. Hooker, and Peter Tiedemann. A constraint store based on multivalued decision diagrams. In *CP 2007*, pages 118–132, 2007.

13 Tarik Hadzic, Rune Jensen, and Henrik Reif Andersen. Calculating Valid Domains for BDD-Based Interactive Configuration. *Computing Research Repository (CoRR)*, 0704.1394, 2007. `arXiv:0704.1394`.

14 Toshihide Ibaraki, Alexander Kogan, and Kazuhisa Makino. Functional dependencies in horn theories. *Artificial Intelligence*, 108(1-2):1–30, 1999.

15 Toshihide Ibaraki, Alexander Kogan, and Kazuhisa Makino. On functional dependencies in q-horn theories. *Artificial Intelligence*, 131(1-2):171–187, 2001.

16 Toshihide Ibaraki, Alexander Kogan, and Kazuhisa Makino. Inferring minimal functional dependencies in horn and q-horn theories. *Annals of Mathematics and Artificial Intelligence*, 38(4):233–255, 2003.

17 T. Kam, T.Villa, R.K. Brayton, and A.L Sangiovanni-Vincentelli. Multi-valued decision diagrams: Theory and applications. *International Journal of Multiple-Valued Logic*, 4:9–12, 1998.

18 Frédéric Koriche, Jean-Marie Lagniez, Pierre Marquis, and Samuel Thomas. Compiling constraint networks into multivalued decomposable decision graphs. In *IJCAI 2015*, 2015.

19 Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Definability for model counting. *Artificial Intelligence*, 281:103229, 2020.

20 Jean-Marie Lagniez and Pierre Marquis. An improved decision-DNNF compiler. In *IJCAI'2017*, volume 17, pages 667–673, 2017.

21 Jean-Marie Lagniez, Pierre Marquis, and Anastasia Paparrizou. Defining and evaluating heuristics for the compilation of constraint networks. In *CP 2017*, pages 172–188. Springer, 2017.

22 Jérôme Lang and Pierre Marquis. On propositional definability. *Artificial Intelligence*, 172(8):991–1017, 2008.

23 Robert Mateescu and Rina Dechter. Compiling Constraint Networks into AND/OR Multi-valued Decision Diagrams (AOMDDs). In *CP 2006*, pages 329–343, 2006.

24 Christian Muise, Sheila A McIlraith, J Christopher Beck, and Eric I Hsu. D sharp: fast d-DNNF compilation with sharpsat. In *CCAI'12*, pages 356–361, 2012.

25 Knot Pipatsrisawat and Adnan Darwiche. New compilation languages based on structured decomposability. In *AAAI 2008*, pages 517–522, 2008.

26 Carsten Sinz. Knowledge compilation for product configuration. In *Proceedings of the Workshop on Configuration at the 15th European Conference on Artificial Intelligence (ECAI)*, pages 23–26, 2002.

27 Takahisa Toda and Takehide Soh. Implementing efficient all solutions SAT solvers. *ACM Journal of Experimental Algorithmics*, 21(1):1.12:1–1.12:44, 2016.

28 Nageshwara Rao Vempaty. Solving Constraint Satisfaction Problems Using Finite State Automata. In *AAAI'92*, pages 453–458, 1992.

29 Bwolen Yang, Reid G. Simmons, Randal E. Bryant, and David R. O'Hallaron. Optimizing symbolic model checking for constraint-rich models. In Nicolas Halbwachs and Doron A. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 328–340. Springer, 1999. `doi:10.1007/3-540-48683-6_29`.

A Appendix: Proof of the main lemma

▶ **Lemma 10.** *Given OMDD ϕ, y passive in ϕ, $a \in D_y$, if ψ_a is the OMDD computed at steps 2a to 2e in Algorithm 2 then*

$$\phi \wedge (y = a) \models \psi_a \quad \textit{and} \quad \psi_a \wedge (y = a) \wedge \exists y.\phi \models \phi \tag{I}$$

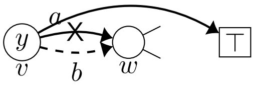
We will show that equation (I) is an invariant of the loop main loop of the algorithm, at step 2. For $\vec{x} \in D_{\mathcal{X}}$, variable z and $b \in D_z$, we write $\vec{x}[z] = b$ when $\vec{x}$ assigns value b to z.

Step 2a: That (I) holds just after ψ has been created is trivial, since at that point $\psi = \phi$.

For the remainder of the proof, we define three predicates, for OMDD ψ and $\vec{x} \in D_{\mathcal{X}}$ (w.r.t. some fixed OMDD ϕ):
$\boldsymbol{I_1(\psi,\vec{x})}$ is true iff $\vec{x}[y] \neq a$ or $f_\phi(\vec{x}) = \bot$ or $f_\psi(\vec{x}) = \top$;
$\boldsymbol{I_2(\psi,\vec{x})}$ is true iff $\vec{x}[y] \neq a$ or $f_\psi(\vec{x}) = \bot$ or $\exists y.f_\phi(\vec{x}) = \bot$ or $f_\phi(\vec{x}) = \top$.
$\boldsymbol{I_3(\psi,\vec{x})}$ is true iff $\vec{x}[y] \neq a$ or $\exists y.f_\phi(\vec{x}) = \bot$ or $\exists y.f_\psi(\vec{x}) = \top$.
We will prove that after every transformation that happen at steps 2b, 2c, 2d or 2e, ψ is such that for every $\vec{x} \in D_{\mathcal{X}}$, $I_1(\psi,\vec{x})$, $I_2(\psi,\vec{x})$ and $I_3(\psi,\vec{x})$ hold; and that $I_1(\psi,\vec{x})$ and $I_2(\psi,\vec{x})$ still hold after steps 2d and 2e.

In the remainder of the proof, $\vec{x} \in D_{\mathcal{X}}$ denotes a model such that $\vec{x}[y] = a$. We write that $\vec{x}$ "*passes through*" some node v in some OMDD ψ if the path in that diagram from the root to either $\top$ or $\bot$ that corresponds to the assignments in $\vec{x}$ contains v.

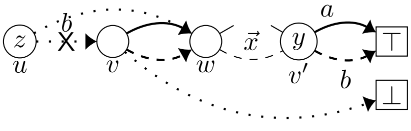
Step 2b: Let ψ denote the OMDD that is obtained after step 2a is executed: for every node v such that $label(v) = y$ and $\text{next}_\phi(v,a) \neq \bot$, $\text{next}_\psi(v,a) = \top$.



Suppose that $\vec{x}$ passes through such a node v in ϕ. (Otherwise, trivially $f_\psi(\vec{x}) = f_\phi(\vec{x})$). Note that $\vec{x}$ passes through v in ψ too, since the transformation applied here does not change the OMDD above y nodes).
$\boldsymbol{I_1(\psi,\vec{x})}$: by construction, $f_\psi(\vec{x}) = \top$. (Recall that we assume that $\vec{x}[y] = a$.)
$\boldsymbol{I_2(\psi,\vec{x})}$: if $\exists y.f_\phi(\vec{x}) = \top$, there must be some $b \in D_y$ such that $f_\phi(\vec{x}') = \top$, where $\vec{x}'$ is obtained by replacing with b the value, a, assigned to y in $\vec{x}$. But y is passive in ϕ, and $\text{next}_\phi(v,a) \neq \bot$, $\text{next}_\phi(v,b) \neq \bot$, so $\text{next}_\phi(v,a) = \text{next}(v,b)$ so necessarily $f_\phi(\vec{x}) = \top$.
$\boldsymbol{I_3(\psi,\vec{x})}$: that $\exists y.f_\psi(\vec{x}) = \top$ follows from the fact that $f_\psi(\vec{x}) = \top$.

We now turn to the transformations that take place at steps 2c. ψ will denote the current OMDD that is being built before such a transformation takes place, and ψ' will denote the OMDD just after the transformation has been applied. At every iteration of step 2c, an undecisive node v of ψ is picked, a child $w \in \text{next}(v)$ is picked, and for every parent u of v in ψ labelled with variable z, for every $b \in D_z$ such that $\text{next}_\psi(u,b) = v$, $\text{next}_{\psi'}(u,b) = w$. We must prove that for every $\vec{x} \in D_{\mathcal{X}}$ that passes through u and v (and thus such that $\vec{x}[z] = b$), if $I_1(\psi,\vec{x})$, $I_2(\psi,\vec{x})$ and $I_3(\psi,\vec{x})$ hold, then $I_1(\psi',\vec{x})$, $I_2(\psi',\vec{x})$ and $I_3(\psi',\vec{x})$ hold too.

Step 2c, undecisive node of type 1: $label(v) \neq y$, and $|\text{next}(v)| = 1$.

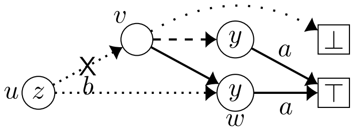


$\boldsymbol{I_1(\psi',\vec{x})}$: if $f_\phi(\vec{x}) = \top$, since $I_1(\psi,\vec{x})$ holds it must be the case that $f_\psi(\vec{x}) = \top$, so the path in ψ from w to a leaf that corresponds to $\vec{x}$ ends at $\top$, and this path is unchanged in ψ'; so $f_{\psi'}(\vec{x}) = \top$.
$\boldsymbol{I_2(\psi',\vec{x})}$: suppose that $f_{\psi'}(\vec{x}) = \top$; then the path in ψ' from w to a leaf that corresponds to $\vec{x}$ ends at $\top$, passing through à node v' labelled by y. Suppose too that $\exists y.f_\phi(\vec{x}) = \top$; then, since $I_3(\psi,\vec{x})$ is true, $\exists y.f_\psi(\vec{x}) = \top$: there is some $\vec{x}'$ identical to $\vec{x}$ except possibly that

$\vec{x}'[y] = b$ for some other $b \in D_y$, such that $f_\psi(\vec{x}') = \top$. Since y is passive, and Step 2 has been executed, it implies that $\text{next}_\psi(v',a) = \top$, and that $f_\psi(\vec{x}) = \top$. But then, since $I_2(\psi,\vec{x})$ holds, it must be the case that $f_\phi(\vec{x}) = \top$.

$\boldsymbol{I_3(\psi',\vec{x})}$**:** suppose that $\exists y.f_\phi(\vec{x}) = \top$; it implies that $\exists y.f_\psi(\vec{x}) = \top$, because $I_3(\psi,\vec{x})$ holds; but then $\exists y.f_{\psi'}(\vec{x}) = \top$ must also be true because ψ' creates a simple "shortcut" for $\vec{x}$.

Step 2c, undecisive node of type 2: v is a node of ψ, whose children, except $\bot$ if it is a child of v, are all y-nodes that are consistent with $y = a$.



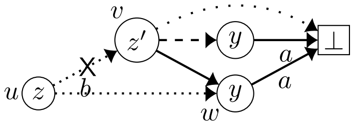
Note that $f_{\psi'}(\vec{x}) = \top$.

$\boldsymbol{I_1(\psi',\vec{x})}$**:** $f_{\psi'}(\vec{x}) = \top$ is true by construction.

$\boldsymbol{I_2(\psi',\vec{x})}$ Suppose that $\exists y.f_\phi(\vec{x}) = \top$. Since $I_3(\psi,\vec{x})$ holds, t cannot be the case that $\text{next}_\psi(v,a) = \bot$ because $\exists y.f_\psi(\vec{x}) = \top$, so $\text{next}\psi(v,a)$ is one of the y node consistent with $y = a$, hence $f_\psi(\vec{x}) = \top$. As a consequence, since $I_2(\psi,\vec{x})$ holds, $f_\phi(\vec{x}) = \top$.

$\boldsymbol{I_3(\psi',\vec{x})}$ That $\exists y.f_{\psi'}(\vec{x}) = \top$ is a simple consequence of the fact that $f_{\psi'}(\vec{x}) = \top$.

Step 2c, undecisive node of type 3: Let v be a node of ψ such that for all $w \in \text{next}(v)$, w is labelled with y and $\text{next}(w,a) = \bot$.



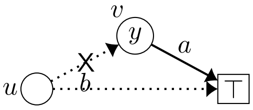
Note that $f_\psi(\vec{x}) = f_{\psi'}(\vec{x}) = \bot$.

$\boldsymbol{I_1(\psi',\vec{x})}$**:** since $I_1(\psi,\vec{x})$ holds and $f_\psi(\vec{x}) = \bot$, it must be the case that $f_\phi(\vec{x}) = \bot$.

$\boldsymbol{I_2(\psi',\vec{x})}$**:** holds because $f_{\psi'}(\vec{x}) = \bot$.

$\boldsymbol{I_3(\psi',\vec{x})}$**:** Let z' be the variable at v. If $\text{next}_\psi(v,\vec{x}[z']) = \bot$, then $\exists y.f_\psi(\vec{x}) = \bot$, thus, since $I_3(\psi,\vec{x})$, $\exists y.f_\phi(\vec{x}) = \bot$. Otherwise, $\text{next}_\psi(v,\vec{x}[z'])$ is a y node, and we assume that the input OMDD is "normalized", so every node has at least one successor different from $\bot$, so w has a successor $\neq \bot$; thus $\exists y.f_{\psi'}(\vec{x}) = \top$.

Step 2d: Let v be a node of ψ labelled with y such that $\text{next}(v,a) = \top$. Let u be a parent of v in ψ labelled with variable z, let $b \in D_z$ such that $\text{next}_\psi(u,b) = v$, then $\text{next}_{\psi'}(u,b) = \top$.

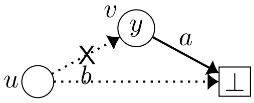


Suppose that $\vec{x}$ passes through u and v in ψ (otherwise, trivially $f_\psi(\vec{x}) = f_{\psi'}(\vec{x})$), so that $\vec{x}$ directly goes from u to $\top$ in ψ'. Note that $f_\psi(\vec{x}) = f_{\psi'}(\vec{x}) = \top$.

$\boldsymbol{I_1(\psi',\vec{x})}$**:** holds because $f_{\psi'}(\vec{x}) = \top$.

$\boldsymbol{I_2(\psi',\vec{x})}$**:** since $I_2(\psi,\vec{x})$ holds and $f_\psi(\vec{x}) = \top$, $\exists y.f_\phi(\vec{x}) = \bot$ or $f_\phi(\vec{x}) = \top$.

Step 2e: Let v be a node of ψ labelled with y such that $\text{next}(v,a) = \bot$. Let u be a parent of v in ψ labelled with variable z, let $b \in D_z$ such that $\text{next}_\psi(u,b) = v$, then $\text{next}_{\psi'}(u,b) = \bot$.



Suppose that $\vec{x}$ passes through u and v in ψ (otherwise, trivially $f_\psi(\vec{x}) = f_{\psi'}(\vec{x})$). Note that $f_\psi(\vec{x}) = f_{\psi'}(\vec{x}) = \bot$.

***I*₁(ψ′,x⃗):** since $I_1(\psi,\vec{x})$ holds and $f_\psi(\vec{x}) = \bot$, it must be the case that $f_\phi(\vec{x}) = \bot$.
***I*₂(ψ′,x⃗):** true because $f_{\psi'}(\vec{x}) = \bot$.

Heuristics for MDD Propagation in Haddock

Rebecca Gentzel
University of Connecticut, Storrs, CT, USA

Laurent Michel
Synchrony Chair in Cybersecurity, University of Connecticut, Storrs, CT, USA

Willem-Jan van Hoeve
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Haddock, introduced in [11], is a declarative language and architecture for the specification and the implementation of multi-valued decision diagrams. It relies on a labeled transition system to specify and compose individual constraints into a propagator with filtering capabilities that automatically deliver the expected level of filtering. Yet, the operational potency of the filtering algorithms strongly correlate with heuristics for carrying out refinements of the diagrams. This paper considers how to empower Haddock users with the ability to unobtrusively specify various such heuristics and derive the computational benefits of exerting fine-grained control over the refinement process.

2012 ACM Subject Classification Mathematics of computing → Decision diagrams; Theory of computation → Constraint and logic programming

Keywords and phrases Decision Diagrams

Digital Object Identifier 10.4230/LIPIcs.CP.2022.24

Supplementary Material *Software (Source Code)*: `https://bitbucket.org/ldmbouge/minicpp/src/v1.1/`

Funding Laurent Michel and Rebecca Gentzel were partially supported by Synchrony. Willem-Jan van Hoeve was partially supported by Office of Naval Research Grant No. N00014-21-1-2240 and National Science Foundation Award #1918102.

1 Introduction

Heuristics are a key ingredient in Constraint Programming. They have been at the core of search procedures for decades. The first-fail heuristic [15] is probably the most well-known representative of how one can affect the performance of a constraint solver with a mere influence on the search strategy that guides the branching process towards the most promising variables. Modern constraint programming solvers typically offer a full complement of such heuristics including weighted degree [8], impact-based search [23], activity-based search [21], conflict-driven search [25], or counting-based search [13] to name just a few. This practice is equally common in mathematical programming with strong branching [3, 1] or pseudo-cost branching [10] or even machine learning based heuristics [5]. This is also true in Boolean satisfiability, with LRB (Learning Rate Branching) [20] and VSIDS (Variable State Independent Decaying Sum) [22] being two of the most regarded such heuristics.

Yet, all these heuristics operate on the level of the entire model and exploit "global behaviors" of the solvers. In constraint programming, for instance, the propagators of most constraints use a prescribed level of consistency when they execute, which dictates the fixpoint they reach. This often leaves little to no room for heuristics to play a role *within* the propagators themselves; however, this is not always true. Cost-based filtering propagators [9, 24] can make use of relaxations to derive bounds on the objective function of a model and use that signal to filter variable domains. Recently, [7] showed how to seek specific Lagrangian multipliers that improve filtering. It is notable that the adoption of relaxations within propagators creates opportunities for heuristics.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 24; pp. 24:1–24:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Decision diagrams present similar opportunities. When applied to optimization problems, multi-valued decision diagrams (MDDs) typically adopt a bounded width (the maximum number of nodes in a layer) and therefore employ some form of relaxation to merge nodes of the diagram [2, 14, 6]. Such merging decisions induce the presence of paths in the MDD that no longer correspond to solutions, necessitating a search process to seek solutions. During the search, internal nodes belonging to layers of the MDD propagator get filtered out (possibly leading to the filtering of variable domains) which reduces the layer size and prompts refinement phases. Indeed, a depleted layer has room to accommodate more nodes that only currently exist in a latent form as part of another, merged node within the layer. Merging and refining nodes are core operations that raise key questions about the impact of choices made on the quality of the obtained relaxation. *The purpose of this paper is to explore the impact of such choices and provide the solver user with a way to dictate the policies that govern relaxation-inducing choices.* Our findings can potentially be applied to any solver that uses relaxed decision diagrams [6, 11, 12].

Haddock [11] provides a specification language and implementation architecture for automatic decision diagram compilation. Haddock provides the rules for refining (splitting) and filtering (propagating) MDD abstractions. The filtering rules are determined by the properties and functions detailed in the specification language, but the refinement process is more abstract. While the filtering rules give valuable tools to remove arcs and states from the MDD, how the MDD is split determines whether filtering rules are able to find infeasible arcs and states and to ultimately filter domains [14].

Contributions. This paper presents an approach to MDD refinement containing configurable heuristics that integrate into Haddock such that all existing Haddock solutions still fit the framework. These heuristics allow the tailoring of refinement rules to specific constraints or models. The rules for refinement play a large role in MDD propagation, and we present insights into why certain refinement rules outperform others.

Paper Structure. The remainder of the paper is organized as follows. Section 2 introduces a motivating example using among constraints. Section 3 reviews the relevant preliminaries, including the formalization used in Haddock. Section 4 discusses the heuristics that parameterize the refinement strategy. Section 5 treats the aggressiveness of the refinement process across layers through the reboot hyper-parameter, while Section 6 reports on the empirical results, and Section 7 concludes the paper.

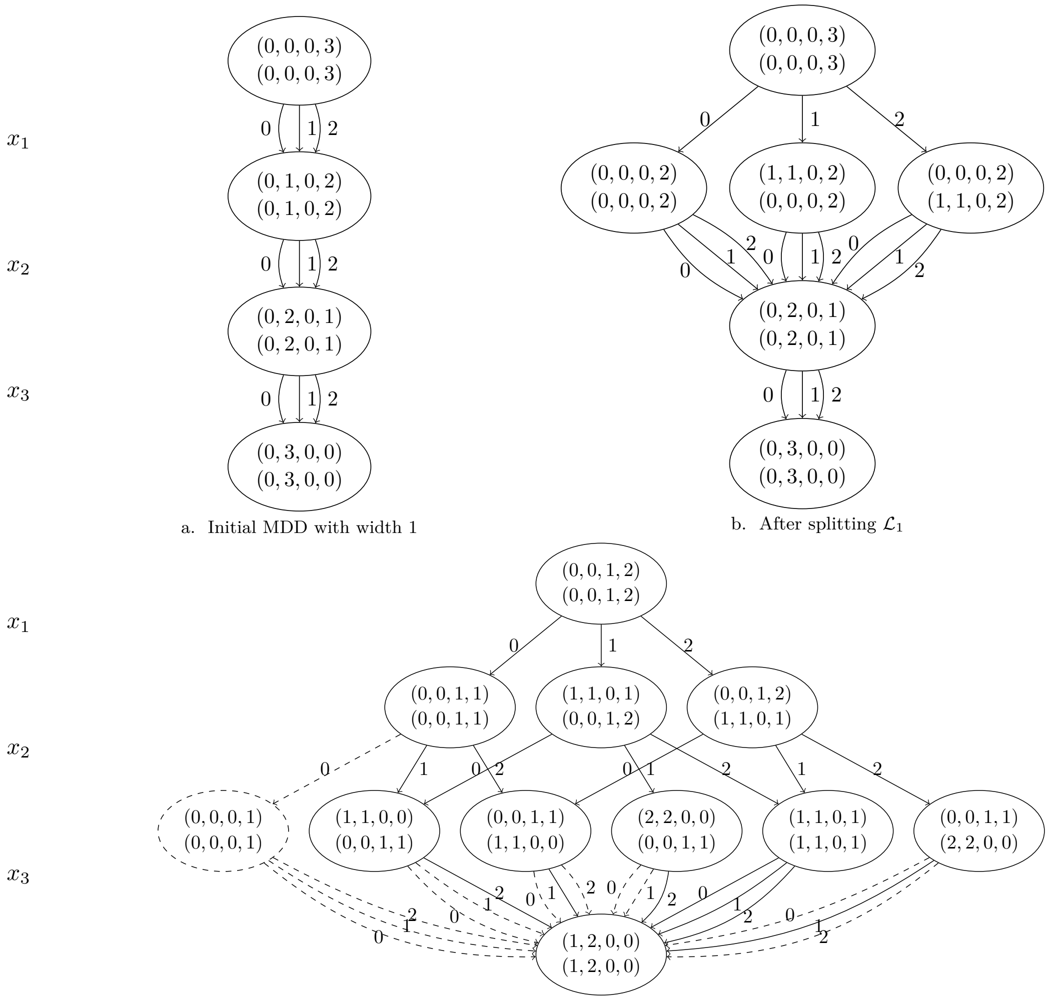
2 Motivating Example

The following example explores the impact that state selection can have on the accuracy of the relaxation produced by an MDD propagator.

▶ **Example 1.** Recall the definition of the among global constraint on an ordered set X of n variables [4]. It counts the number of occurrences of values taken from a given set Σ and ensures that the total number is between l and u, i.e.,

$$\texttt{among}(X, l, u, \Sigma) := l \leq \sum_{i=1}^{n} (x_i \in \Sigma) \leq u.$$

Consider two constraints $c_1 = \textsc{Among}(\{x_1, x_2, x_3\}, l_1 = 1, u_1 = 2, \Sigma_1 = \{1\})$ and $c_2 = \textsc{Among}(\{x_1, x_2, x_3\}, l_2 = 1, u_2 = 2, \Sigma_2 = \{2\})$ where each variable has domain $\{0, 1, 2\}$. An MDD for these constraints is a layered directed acyclic graph with four layers $(\mathcal{L}_0, \ldots, \mathcal{L}_3)$, a source $s_\perp$, and a sink $s_\top$. Arcs flow from a node in layer $\mathcal{L}_{i-1}$ to a node in layer $\mathcal{L}_i$ and



a. Initial MDD with width 1

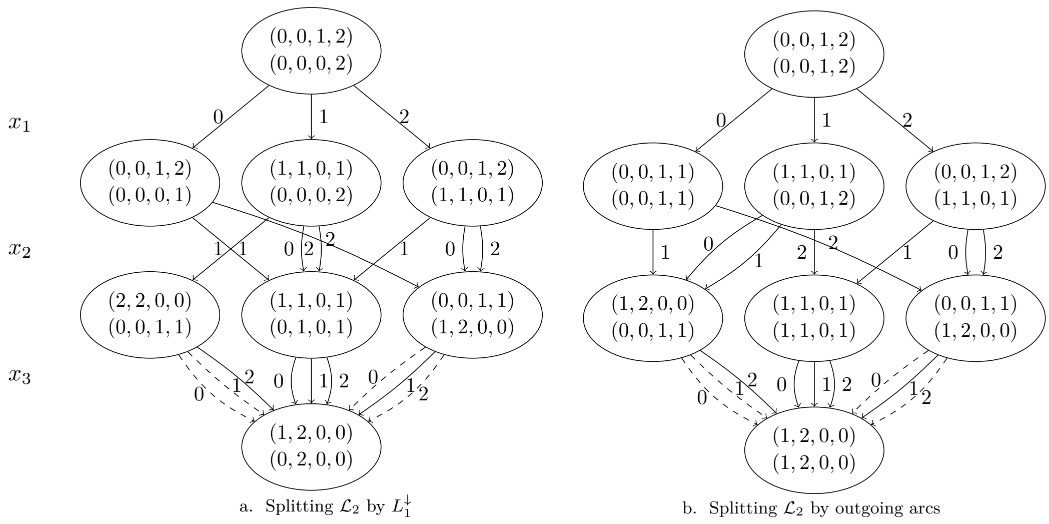
b. After splitting $\mathcal{L}_1$

c. If a full split of $\mathcal{L}_2$ were possible

Figure 1 Exact refinement process. Dashed nodes and arcs can be filtered.

are labeled with a domain value v, stating the assignment $x_i = v$. Every $s_\perp$-$s_\top$ path denotes a candidate solution. Each node carries a state $s = \langle s_1, s_2\rangle$ with $s_1 = \langle L_1^\downarrow, U_1^\downarrow, L_1^\uparrow, U_1^\uparrow\rangle$ and $s_2 = \langle L_2^\downarrow, U_2^\downarrow, L_2^\uparrow, U_2^\uparrow\rangle$ with the properties of c_1 and c_2. Intuitively, $L_i^\downarrow$ and $U_i^\downarrow$ denote the lower and upper bound, respectively, on the number of occurrences of values from Σ_i on any $s_\perp$-s paths in the MDD. $L_i^\uparrow$ and $U_i^\uparrow$ are similarly defined on s-$s_\top$ paths.

Figure 1(a) depicts the MDD at width 1. Assume one imposes a maximum width of 3. Refinement begins by splitting $\mathcal{L}_1$. As shown in Figure 1(b), $\mathcal{L}_1$ can be fully split into three states. Next, refinement is performed on $\mathcal{L}_2$. A full split is shown for this layer in Figure 1(c). While the state on the far left is infeasible and can be deleted, five states remain with a maximum width of 3. A splitting of this layer partitions the five states into three groups. One partitioning strategy is to solely rely on $L_1^\downarrow$. Since there are exactly three values for $L_1^\downarrow$ in these five states $(0, 1, 2)$, the five states group neatly. The result is shown in Figure 2(a). An alternative is depicted in Figure 2(b) the grouping is based on the labels of outgoing arcs to $s_\top$ ($\{1\}$, $\{2\}$, and $\{0, 1, 2\}$ after filtering infeasible arcs). While the first partition strategy still has $s_\perp$-$s_\top$ paths representing infeasible assignments, e.g. $x_1 = 0$, $x_2 = 1$, $x_3 = 1$, the second partition provides an exact MDD despite $\mathcal{L}_2$ still harboring merged states. It is clear that choices made during refinement impact the accuracy of the MDD and its ability to filter.



a. Splitting $\mathcal{L}_2$ by $L_1^{\downarrow}$

b. Splitting $\mathcal{L}_2$ by outgoing arcs

Figure 2 Options for partitioning $\mathcal{L}_2$. Dashed arcs can be filtered.

3 Background

Following [11], we formally define an MDD as a labeled transition system [17]:

▶ **Definition 2.** *A* labeled transition system *is a triplet* $\langle \mathcal{S}, \rightarrow, \Lambda \rangle$ *where* $\mathcal{S}$ *is a set of states,* $\rightarrow$ *is a relation of labeled transitions between states from* $\mathcal{S}$*, and* Λ *is a set of labels used to tag transitions.*

▶ **Definition 3.** *Given an ordered set of variables* $X = \{x_1, \ldots, x_n\}$ *with domains* $D(x_1)$ *through* $D(x_n)$*, a* multi-valued decision diagram (MDD) *on* X *is an LTS* $\langle \mathcal{S}, \rightarrow, \Lambda \rangle$ *in which:*

- *the state set* $\mathcal{S}$ *is stratified in* $n + 1$ *layers* $\mathcal{L}_0$ *through* $\mathcal{L}_n$ *with transitions from* $\rightarrow$ *connecting states between layers* i *and* $i + 1$ *exclusively;*
- *the transition label set* Λ *is defined as* $\bigcup_{i \in 1..n} D(x_i)$*;*
- *a transition between two states* $a \in \mathcal{L}_{i-1}$ *and* $b \in \mathcal{L}_i$ *carries a label* $v \in D(x_i)$ $(i \in 1..n)$*;*
- *the layer* $\mathcal{L}_0$ *consists of a single* source *state* $s_\perp$*;*
- *the layer* $\mathcal{L}_n$ *consists of a single* sink *state* $s_\top$*.*

An MDD M can represent a constraint set with specific state definitions and transition functions. If each solution in the constraint set is represented by an $s_\perp$-$s_\top$ path in M, and vice-versa, M is *exact*. If M represents a superset of the solutions of the constraint set, it is *relaxed*. In Haddock, states consist of integer-valued sets of *properties* to represent the constraints. We next describe how these are used to automatically compile the LTS, using the Among constraint as an illustration. For a complete description, we refer to [11].

State Properties. As mentioned in Example 1, a state for Among(X, l, u, Σ) carries four properties, i.e., $\langle L^{\downarrow}, U^{\downarrow}, L^{\uparrow}, U^{\uparrow} \rangle$, for each node v in the MDD:

- $L^{\downarrow} \in \mathbb{Z}$: minimum number of times a value in Σ is taken from $s_\perp$ to v.
- $U^{\downarrow} \in \mathbb{Z}$: maximum number of times a value in Σ is taken from $s_\perp$ to v.
- $L^{\uparrow} \in \mathbb{Z}$: minimum number of times a value in Σ is taken from v to $s_\top$.
- $U^{\uparrow} \in \mathbb{Z}$: maximum number of times a value in Σ is taken from v to $s_\top$.

We initialize the state for the source $s_\perp$ as $\langle 0, 0, -, - \rangle$ and the sink $s_\top$ as $\langle -, -, 0, 0 \rangle$.

Transition Functions. The transition between a node $a \in \mathcal{L}_{i-1}$ and $b \in \mathcal{L}_i$ is an arc (a, b) labeled by a value $\ell \in D(x_i)$. We use *transition functions* $T^{\downarrow}(a, b, i, \ell)$ and $T^{\uparrow}(b, a, i, \ell)$ to derive the property values (the states) for b and a, respectively. For each individual property p, we use the function $f(s, p, \ell)$ for a given state s. For Among, we apply $f(s, p, \ell) = p(s) + (\ell \in \Sigma)$ for each property p in $\langle L^{\downarrow}, U^{\downarrow}, L^{\uparrow}, U^{\uparrow} \rangle$. For example, we define $L^{\downarrow}(b) = f(a, L^{\downarrow}, \ell)$, i.e., $L^{\downarrow}(a) + (\ell \in \Sigma)$. We likewise define $L^{\uparrow}(a) = f(b, L^{\uparrow}, \ell)$, $U^{\downarrow}(b) = f(a, U^{\downarrow}, \ell)$ and $U^{\uparrow}(a) = f(b, U^{\uparrow}, \ell)$. The state-level transition functions $T^{\downarrow}$ and $T^{\uparrow}$ compute all the down or up properties of the next state as follows:

$$T^{\downarrow}(a, b, i, \ell) = \langle f(a, L^{\downarrow}, \ell), f(a, U^{\downarrow}, \ell), -, - \rangle$$
$$T^{\uparrow}(b, a, i, \ell) = \langle -, -, f(b, L^{\uparrow}, \ell), f(b, U^{\uparrow}, \ell) \rangle.$$

Note that slight variants of both functions that preserve the properties of states b and a, respectively, in the opposite directions are equally helpful. Those are:

$$T^{\downarrow}(a, b, i, \ell) = \langle f(a, L^{\downarrow}, \ell), f(a, U^{\downarrow}, \ell), L^{\uparrow}(b), U^{\uparrow}(b) \rangle$$
$$T^{\uparrow}(b, a, i, \ell) = \langle L^{\downarrow}(a), U^{\downarrow}(a), f(b, L^{\uparrow}, \ell), f(b, U^{\uparrow}, \ell) \rangle.$$

Transition Existence Function The *transition existence function* $E_t(a, b, i, \ell)$ specifies whether an arc (a, b) with label $\ell \in D(x_i)$ exists in the LTS. For Among, this function should ensure that the lower bound l is met and the upper bound u is not exceeded, i.e.:

$$U^{\downarrow}(a) + (\ell \in S) + U^{\uparrow}(b) \geq l \wedge L^{\downarrow}(a) + (\ell \in S) + L^{\uparrow}(b) \leq u.$$

Node Relaxation Functions Two states a and b in the same layer $\mathcal{L}_i$ can be relaxed (merged) to produce a new state s' according to a *relaxation function* $\texttt{relax}(a, b)$. For Among, we can use:

$$\begin{aligned} \texttt{relax}(a, b) = \langle \quad & \min\{L^{\downarrow}(a), L^{\downarrow}(b)\}, \max\{U^{\downarrow}(a), U^{\downarrow}(b)\}, \\ & \min\{L^{\uparrow}(a), L^{\uparrow}(b)\}, \max\{U^{\uparrow}(a), U^{\uparrow}(b)\} \quad \rangle. \end{aligned}$$

We also call such relaxed states *approximate* states.
State relaxation generalizes to an ordered set of states $\{s_0, s_1, \ldots, s_{k-1}\}$ as follows:

$$\texttt{relax}(s_0, \texttt{relax}(s_1, \texttt{relax}(\ldots, \texttt{relax}(s_{k-2}, s_{k-1})\ldots))).$$

For Among, we maintain MDD-bounds consistency on this expression, i.e., we only maintain a lower and upper bound on the count to ensure feasibility and rely on the above relaxation function to merge nodes and bound the width of the MDD to at most w states. The usage of a relaxation is precisely why we maintain bounds (L and U) in both up and down directions. Note that full MDD consistency for Among can be established in polynomial time by maintaining a set of exact counts [16].

Notation. For any state $s \in \mathcal{L}_i$ with $1 \leq i \leq n$, let $\delta^-(s)$ denote the set of inbound arcs from layer $\mathcal{L}_{i-1}$. Likewise let $\delta^+(s)$ denote the set of outbound arcs into $\mathcal{L}_{i+1}$. We sometimes overload notation and use $\delta^-(s)$ and $\delta^+(s)$ to also refer to the set of states in $\mathcal{L}_{i-1}$ and $\mathcal{L}_{i+1}$, respectively, one can reach from s via those arcs.

4 Decision Diagram Refinement

Haddock [11] provides an abstract definition for refining an MDD. For refining one layer, it takes a single state, orders all of that state's incoming arcs, groups these arcs based on equivalence classes, and creates new states for each of these equivalence classes [14]. This

Algorithm 1 $\texttt{refineLayer}(\mathcal{L}_i, [\mathcal{L}_0, \ldots, \mathcal{L}_{i-1}], w, \langle Y, Q, W\rangle)$.

```
Require: $|\mathcal{L}_i| \leq w$
Ensure: $|\mathcal{L}_i| = w \vee \texttt{appx}(\mathcal{L}_i) = \emptyset$
1: while $|\mathcal{L}_i| < w \wedge \texttt{appx}(\mathcal{L}_i) \neq \emptyset$ do
2:     let $s^* = \arg\max_{s \in \texttt{appx}(\mathcal{L}_i)} Y(s)$
3:     let $cs = \texttt{partition}(\texttt{refine}(s^*), Q)$
4:     if $|cs| \leq w - |\mathcal{L}_i| + 1$ then
5:         $\mathcal{L}_i = \mathcal{L}_i \setminus \{s^*\} \cup \bigcup_{j=1}^{|cs|} \texttt{relax}(cs_j)$
6:     else
7:         let $\pi = \texttt{permutation}(cs) \mid \forall j, k \in 1..|cs| : j \leq k \Rightarrow W(s_{\pi_j}) \leq W(s_{\pi_k})$
8:         $\mathcal{L}_i = \mathcal{L}_i \setminus \{s^*\} \cup \bigcup_{j=1}^{w-|\mathcal{L}_i|} \texttt{relax}(cs_{\pi_j}) \cup \texttt{relax}(\bigcup_{j=w-|\mathcal{L}_i|+1}^{|cs|} cs_{\pi_j})$
```

process introduces space for multiple heuristics. Which relaxed state is selected for splitting? How should the results of the splitting be ordered and partitioned? This section turns these choices into definable heuristic functions building off of the framework of Haddock.

Algorithm 1 gives the pseudo-code of the layer refinement. It takes as input layer $\mathcal{L}_i$, a target width w and three functions Y, W, and Q (shown in red) that are the embodiment of the user-definable heuristics. The algorithm makes use of several sub-routines (`appx`, `refine`, `partition`, and `permutation`) that will be explained below. Algorithm 1 refines a layer by repeatedly pulling out states that can be refined (if any) and replacing them in the layer by more precise versions given the availability of space in the targeted layer. The Y function drives the selection of the approximate state to replace, while Q and W govern the mechanisms to synthesize the replacement. The section closes with an in-depth discussion of `refineLayer` once all its components are laid out.

4.1 State Selection with Y

The first step is to select which state in the layer $\mathcal{L}_i$ should be refined (line 2 in Alg. 1). When the MDD is first constructed, each layer only has one state, so this is trivial. We therefore assume that $1 < |\mathcal{L}_i| < w$. $\mathcal{L}_i$ may contain both exact and approximate states as a result of prior merging. The function call $\texttt{appx}(\mathcal{L}_i)$ returns the subset of states that are the results of prior approximations (merges). Ideally, one would wish to refine the layer and replace all approximate nodes with exact ones until $|\mathcal{L}_i| = w$. The order in which we select an approximate state s^* for refinement is driven by *state priority functions*:

▶ **Definition 4.** *A* state priority function $Y : \mathcal{S} \rightarrow \mathbb{Z}$ *takes as input state* $s = \langle P_0, \ldots, P_{k-1}\rangle$ *and returns an integer value representing its priority where the larger is the more preferable.*

The refinement will retract the selected state s^* from the layer and replace it with an expansion that consists of one or more new states. The size of this expansion drives the remainder of the algorithm. Focusing on Y, several natural choices come to mind. Some are based on the local topology of the MDD around the selected state s^*, while others are *semantics* driven and leverage the properties held within s^*. Recall that the layer is an ordered set (states are ordered within the layer and have a rank between 0 and the cardinality of the set) and that states have topological properties such as the sets of incoming ($\delta^-(s)$) and outgoing ($\delta^+(s)$) arcs. While purely syntactic, these properties may be attractive. As the newest states are the ones most recently refined, the age of states may be a useful metric:

▶ **Example 5** (Rank heuristics). Let $Y(s) = -\texttt{rank}(s)$ be the heuristic to first select the oldest states inserted in the layer. Likewise, one can define $Y(s) = \texttt{rank}(s)$ to first select the nodes that were most recently inserted in the layer.

Another natural option is to consider the in-degree of the state in the MDD to get:

▶ **Example 6** (Degree heuristics). Let $Y(s) = -\delta^-(s)$ be the heuristic to first select low in-degree states, i.e., states that have few parents in the prior layer.

▶ **Example 7** (Semantics-based heuristic). Consider the constraint $\textsc{Among}(X, l, u, \Sigma)$ using state $s = (L^\downarrow, U^\downarrow, L^\uparrow, U^\uparrow)$ with $L^\downarrow$ and $U^\downarrow$ as specified earlier. Define the state selection heuristic $Y(s) = L^\downarrow(s) + L^\uparrow(s)$ to preferentially select a state with the largest lower bound on the number of occurrences of values from Σ on any path $s_\top$ to $s_\bot$. Likewise, the heuristic $Y(s) = -(U^\downarrow(s) + U^\uparrow(s))$ would select the state with the smallest upper bound on the number of occurrences of values from Σ along those paths.

4.2 Candidate Selection with Q and W

Once line 2 of Algorithm 1 has executed, state s^* needs to be refined. To evaluate its incoming arcs, we define the function $A(s)$ that collects the set of arcs leading to state s from the prior layer:

$$A(s) = \{p_j \xrightarrow{\ell_j} s \mid p_j \in \mathcal{L}_{i-1} \wedge \ell_j \in D(x_i)\}$$

Equipped with $A(s^*)$ one can compute what the true endpoint of each arc should have been without relaxation. The outgoing arcs of these endpoints are a subset of $\delta^+(s^*)$ built by removing infeasible arcs from $\delta^+(s^*)$. Namely for a true descendent s' computed from an endpoint in $A(s)$, we have

$$\delta^+(s') = \{s' \xrightarrow{\ell_j} c_j \mid s \xrightarrow{\ell_j} c_j \in \delta^+(s) \wedge E_t(s', c_j, i, \ell_j)\}$$

If $\delta^+(s') = \emptyset$, then the corresponding arc in $A(s^*)$ can be removed from the MDD. With this, we can compute $K(s^*)$, the multiset of true descendants according to the remaining arcs in $A(s^*)$ thanks to the forward state transition rule $T^\downarrow$:

$$K(s) = \{s' = T^\downarrow(p_j, s, i, \ell_j) \mid p_j \xrightarrow{\ell_j} s \in A(s) \wedge \delta^+(s') \neq \emptyset\}.$$

Note how $\texttt{relax}(K(s^*)) = s^*$ since $K(s^*)$ is none other than the multiset of states that would yield s^* if merged. The $\texttt{refine}(s^*)$ function in Algorithm 1 (line 3) is responsible for producing the multiset $K(s^*)$. With unbounded width, one could retain the *unique* states in $K(s^*)$ and add all of them into $\mathcal{L}_i \setminus \{s^*\}$ to upgrade s^*. Otherwise, we need to group together states in $K(s^*)$ to be merged. The generic `partition` function (line 3 in Alg. 1) returns a partition of $K(s^*)$ into multisets $S_1, \ldots, S_p$, each of which representing an approximately equivalent multiset of states. That is, $S_i \subseteq K(s^*)$ for $1 \leq i \leq p$, $S_i \cap S_j = \emptyset$ for $1 \leq i < j \leq p$, and $\bigcup_{i=1}^p S_i = K(s^*)$. The heuristic function Q determines which states should be grouped together. For example, if Q is a binary relation that encodes equality, $\texttt{partition}(K(s^*), Q)$ must ensure that $Q(a, b)$ holds for all $a, b \in S_i$ $(1 \leq i \leq p)$ and $Q(a, b)$ does not hold for all $a \in S_i$, $b \in S_j$ $(1 \leq i < j \leq p)$.

Whenever $|S_i| > 1$, we can apply the `relax` function to collapse S_i into a single state. The resulting states can all be added to the layer if it would not exceed maximum width (lines 4-5 in Alg. 1). Otherwise, we need to determine which states to add and which to merge. To do this, we use heuristic function W to compute a sorted permutation of the partition $S_1, \ldots, S_p$. The permutation induced by W identifies the first (and most promising) $w - |\mathcal{L}_i|$ collapsed states for inclusion and merges the remaining ones into a single state.

To formalize the description above, let us adopt the following definitions:

▶ **Definition 8** (Equivalence class). *A* state equivalence function *takes the form $Q : \mathcal{S} \times \mathcal{S} \to \mathbb{B}$. It takes as input states $a = \langle A_0, \ldots, A_{k-1}\rangle$ and $b = \langle B_0, \ldots, B_{k-1}\rangle$ and returns whether the two states are considered similar enough.*

So long as Q is an equivalence relation (reflexive, symmetric, and transitive), Q can generate a partition of $K(s^*)$. Naturally, the most direct example is pure *equality.*

▶ **Example 9** (Equality). Let $\overline{Q}(a, b)$ be a binary state equivalence function that holds over states $a = \langle A_0, \ldots, A_{k-1}\rangle$ and $b = \langle B_0, \ldots, B_{k-1}\rangle$ when all properties are point-wise equal, i.e., $\overline{Q}(a, b)$ holds if and only if $\bigwedge_{i=0}^{k-1} A_i = B_i$.

While combining equal states is helpful, one may wish to group states that are similar but not identical. We refer to all other types of state equivalence as *approximate equivalence.* Which properties are used for determining equivalence may be problem dependent. Hence the desire to make it programmable. Any states that are deemed *approximately equivalent* are relaxed together by virtue of being members of the same class. The desire to preserve a strong relaxation should bias the design of Q to induce the weakest possible losses as a result of applying the `relax` function. To appreciate this *semantic* use, consider this example:

▶ **Example 10** (Bound Slackness). Consider the constraint Among(x, l, u, Σ) using state $s = (L^{\downarrow}, U^{\downarrow}, L^{\uparrow}, U^{\uparrow})$ as before. It is easy to assess how close the current bounds on the number of occurrences of values in Σ are compared to l and u. Given two states $a, b \in K(s)$, $a = T^{\downarrow}(p_a, s, i, \ell_a)$ and $b = T^{\downarrow}(p_b, s, i, \ell_b)$. If $L^{\downarrow}(a) + L^{\uparrow}(a)$ and $L^{\downarrow}(b) + L^{\uparrow}(b)$ are equally close to l, one would incur a weak loss of precision when merging a with b since merging uses min on property $L^{\downarrow}$, and $L^{\uparrow}(a) = L^{\uparrow}(b) = L^{\uparrow}(s)$ because a and b are derived by only calling the *forward* state transition rule. The same argument applies to the $U^{\downarrow}, U^{\uparrow}$ properties and the distance to the upper bound u. Therefore, let $Q_t(a, b)$ be a parametric approximate equivalence class (with parameter t) defined as

$$\begin{aligned} Q(a, b) =& ((l - (L^{\downarrow}(a) + L^{\uparrow}(a)) > t) = (l - (L^{\downarrow}(b) + L^{\uparrow}(b)) > t)) \\ &\wedge ((u - (U^{\downarrow}(a) + U^{\uparrow}(a)) > t) = (u - (U^{\downarrow}(b) + U^{\uparrow}(b)) > t)) \end{aligned}$$

Interestingly, setting $t = 0$ means that states a and b are equivalent as soon as both bounds have any amount of slack while $t = +\infty$ means that the inequalities are never satisfied forcing each state to stand in a separate class (no relaxations as a result of similar slackness).

▶ **Definition 11** (Weight function). *A* candidate weight function *takes the form $W : \mathcal{S} \to \mathbb{Z}$. It takes as input a state and returns an integer value representing its desirability (smaller is better).*

The weight function is used to derive a permutation of $K(s^*)$. Consider the following examples that leverage simple structural properties:

▶ **Example 12** (Number of arcs heuristic). Let $W(s) = |\delta^{-}(s)|$ be the heuristic that favors nodes with fewer antecedents in the layer above.

▶ **Example 13** (Parent rank heuristic). Let $W(s) = -\max_{p \in \delta^{-}(s)} rank(p)$ be the heuristic that favors nodes with parents that were created in the parent layer the most recently.

4.3 Composing Heuristics

Haddock delivers a framework to automatically deliver MDD-driven propagators for constraints through specifications that use state definitions together with several functions to capture transition, transition existence, state existence, and relaxations. Perhaps even more interestingly, Haddock provides a composition mechanism to produce MDD specifications from the conjunction of multiple high-level constraints. Such composite specifications then drive the generation of the MDD propagator.

The addition of heuristics (Y, Q, and W) to modulate the behavior of the underlying propagator raises a natural question. When each constraint brings its own *preferred heuristics*, how does one combine them into a single composite heuristic for the propagator? We extend the definition of an MDD language from [11] to incorporate the bundle of 3 heuristics:

▶ **Definition 14** (MDD Language). *Given a constraint $c(x_1, \ldots, x_n)$ over an ordered set of variables $X = \{x_1, \ldots, x_n\}$ with domains $D(x_1), \ldots, D(x_n)$ the* MDD language *for c is a tuple $\mathcal{M}_c = \langle X, \mathcal{P}, s_\perp, s_\top, T^\downarrow, T^\uparrow, U, E_t, E_s, R, H = \langle Y, Q, W\rangle\rangle$ where $\mathcal{P}$ is the set of properties used to model states, $s_\perp$ is the source state, $s_\top$ is the sink state, $T^\downarrow$ is the forward state transition rule, $T^\uparrow$ is the reverse state transition rule, U is the state update function, E_t is the transition existence function, E_s is the state existence function [11], and $H = \langle Y, Q, W\rangle$ is the trio of heuristics controlling the refinement process.*

4.3.1 Direct Composition

Consider two MDD languages $\mathcal{M}_1$ and $\mathcal{M}_2$ for constraints c_1 and c_2 defined over overlapping ordered sets of variables X and Y ($X \cap Y \neq \emptyset$). Let the language $\mathcal{M}_1 \wedge \mathcal{M}_2$ denote the composition of $\mathcal{M}_1$ and $\mathcal{M}_2$ and associate to it a heuristic bundle $H_{\mathcal{M}_1 \wedge \mathcal{M}_2}$ defined as:

▶ **Definition 15.** *Given heuristic bundles $H_{c_1} = \langle Y_{c_1}, Q_{c_1}, W_{c_1}\rangle$ and $H_{c_2} = \langle Y_{c_2}, Q_{c_2}, W_{c_2}\rangle$, let $H_{\mathcal{M}_1 \wedge \mathcal{M}_2} = \langle Y_{c_1} + Y_{c_2}, Q_{c_1} \wedge Q_{c_2}, W_{c_1} + W_{c_2}\rangle$ denote the heuristic bundle of the composition.*

4.3.2 Portfolio Composition

While direct composition can be effective, it may be sometimes too restrictive. An MDD may encapsulate several constraints that *disagree* on the guidance that they offer individually. In such circumstances, it might be preferable instead to base the refinements on the advice of a portfolio in which the heuristic bundles coming from each constraint are prioritized. To allow for this, we define the *refinement portfolio* as:

▶ **Definition 16.** *A* refinement portfolio *is an ordered list $(h_1, \ldots, h_k)$ of heuristic bundles with $h_i = \langle Y_i, Q_i, W_i\rangle$ for each $i \in \{1, \ldots, k\}$.*

To understand how the portfolio is leveraged, consider the fixpoint algorithm used within an MDD propagator for the conjunction of m constraints $\wedge_{i=1}^{m} c_i$ shown in Algorithms 2 and 3. Blue text can be ignored at first as it relates to the reboot and maximum refinement described in Section 5. Algorithm 2 is the core of the fixpoint in the MDD propagator. It first collects into the list HP all the heuristic bundles to be used. It then proceeds in lines 3-9 to carry out passes over the layers of the MDD. Each iteration starts with a backwards pass going over layers $\mathcal{L}_{n-1}$ to $\mathcal{L}_0$ to update the “up” properties of all states. This can lead to the deletion of arcs and states. It then proceeds (line 5) with a down pass to update the forward properties of the states that changed, but also to replenish layers that are no longer full. Finally, lines 6-7 trim the variable domains to echo the changes done to the MDD representation. Any changes prompt another iteration. Algorithm 3 is the crux of the forward pass over layers $\mathcal{L}_1$ to $\mathcal{L}_n$. The loop in lines 3-8 does the layer refinement while lines 9-10 compute the update and the pruning of each layer. While Algorithm 3 implies that the process iterates over all layers, this is a simplification as the implementation only considers changed states in changed layers. That simplification does not affect the layer refinement.

Algorithm 2 mddFixpoint($\mathcal{M}_{c_1 \wedge \cdots \wedge c_m}, [x_1, \ldots, x_n], width, reboot, maxRef$).

```
let $HP = [\langle Y_1, Q_1, W_1\rangle, \ldots, \langle Y_k, Q_k, W_k\rangle]$
let $iter = 0$
repeat
    $changed$ = computeUp($\mathcal{M}_{c_1 \wedge \cdots \wedge c_m}$)
    $changed$ = computeDown($\mathcal{M}_{c_1 \wedge \cdots \wedge c_m}, width, HP, iter, reboot, maxRef$) $\vee$ $changed$
    for $i \in 1..n$ do
        trimVariable($x_i$)
    $iter = iter + 1$
until $\neg changed$
```

Algorithm 3 computeDown($\mathcal{M}_{c_1 \wedge \cdots \wedge c_m}, width, HP, iter, reboot, maxRef$).

```
let $changed = false$
if $iter < maxRef$ then
    for $hp \in HP$ do
        let $i = 1$
        repeat
            $l$ = refineLayer($\mathcal{L}_i, [\mathcal{L}_0, \ldots, \mathcal{L}_{i-1}], width, hp$)
            $i = (l < i)\ ?\ \max(l, i - reboot) : (i + 1)$
        until $i = n$
for $i \in 1..n$ do
    $changed$ = pruneLayer($\mathcal{L}_i$) $\vee$ $changed$
return $changed$
```

4.3.3 Refinement Portfolio Options

Different choices for Q are possible. One could use (for a given constraint c) either an approximation $\widetilde{Q}$ or pure state equality $\overline{Q}$. Alternatively, *both* can be used in a portfolio $[\langle Y, \widetilde{Q}, W\rangle, \langle Y, \overline{Q}, W\rangle]$ that uses them in a round-robin style. This first conservatively expands with a coarse equivalence, and, if room is still available, uses the finer grain equality.

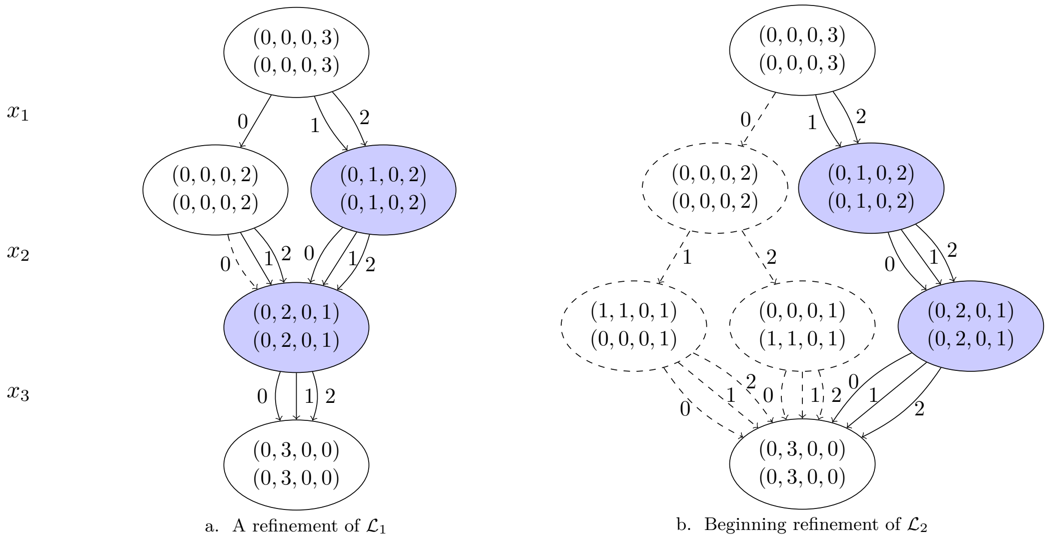
4.3.4 Refinement Portfolio with Constraint Ranking

Another option is to populate the portfolio with heuristic bundles from each constraint embedded in the MDD. Given the constraint set $\{c_1, \ldots, c_m\}$, one can produce a portfolio $HP = [\langle Y_{\pi_0}, Q_{\pi_0}, W_{\pi_0}\rangle, \ldots \langle Y_{\pi_{m-1}}, Q_{\pi_{m-1}}, W_{\pi_{m-1}}\rangle]$ that permutes the bundles according to a user defined ordering π. This can be taken a step further by grouping constraints. Groups have a single heuristic bundle obtained through composition. This grouping of constraints for MDD refinement bears similarities to propagator groups [18]. Both ideas for portfolios compose, expanding HP to include two bundles for each constraint, one that uses $\widetilde{Q}_{\pi_i}$ and one that uses $\overline{Q}_{\pi_i}$. This preserves the ranking goal by prioritizing constraints with a higher rank above constraints of lower rank while always first splitting with $\widetilde{Q}$ before $\overline{Q}$.

5 Layer Processing

5.1 Reboot Distance

The refinement of a layer in Algorithm 1 may terminate with a full layer ($|\mathcal{L}_i| = w$) that still hosts approximate states and has the potential for further refinements. As refinements proceed through layers, a call to `refineLayer`($\mathcal{L}_i, \ldots$ that causes the deletion of nodes in $\mathcal{L}_i$



a. A refinement of $\mathcal{L}_1$

b. Beginning refinement of $\mathcal{L}_2$

Figure 3 Two Among constraints with maximum width=2. Highlighted nodes are approximate.

and in some preceding layers $\mathcal{L}_l$ can trigger a return to the shallowest layer $\mathcal{L}_l$ to immediately refine it again as opposed to continuing onward from i. When this happens, the referenced loop would "reboot" to layer $\mathcal{L}_l$. It may be desirable to bound how far one might reboot with a maximum reboot distance between l and i. To reflect this, we add to `computeDown` the changes in blue on lines 6-7 of Algorithm 3. We further assume that `refineLayer` is modified to return the index of the highest layer l changed during the function call.

▶ **Example 17.** Consider an MDD similar to Example 1 with $l_2 = 2$ and maximum width 2. After splitting $\mathcal{L}_1$, we obtain the graph in Figure 3(a). Nodes are highlighted if their state is relaxed. After refining $\mathcal{L}_1$ the right state is still relaxed and cannot be refined due to lack of space in the layer. While splitting $\mathcal{L}_2$, two states in $K(s)$ have no feasible children leading to the deletion of the corresponding arcs in $A(s)$. As a result, a state in $\mathcal{L}_1$ can be removed as shown in Figure 3(b). Without reboot (or $reboot = 0$), $\mathcal{L}_2$ would continue being refined. If $reboot \geq 1$, the refinement will instead elect to further refine $\mathcal{L}_1$ first.

5.2 Maximum Refinement Iterations

Algorithm 2 iterates until a fixpoint is reached. It may be wise to bound the number of times refinement can occur within one call to the fixpoint. We denote this the maximum refinement iterations. The refinement in Algorithm 3 is conditional (line 2) and keeps track of the iteration number in Algorithm 2 (lines 2, 8).

6 Empirical Evaluation

HADDOCK is part of `MiniC++`, a C++ implementation of the MiniCP specification [19]. All benchmarks were executed on a Macbook Pro with a 3.1 GHz Intel Core i7-5557U processor and 16GB. This section explores the effects of several heuristics on the behavior of the HADDOCK propagator. Specifically, we consider the following experiments:

Experiment 1 Investigate the impact of the Y and W heuristics.

Experiment 2 Explore the merits of $\widetilde{Q}$, $\overline{Q}$, and a portfolio using first $\widetilde{Q}$, then $\overline{Q}$.

Table 1 CPU time (seconds) to obtain all solutions for Nurse Rostering using $HP = [\langle Y, \widetilde{Q}, W\rangle, \langle Y, \overline{Q}, W\rangle]$ for different Y (columns) and W (rows) heuristics.

Instance		Width 16				Width 32				Width 64			
		HR	LR	HD	LD	HR	LR	HD	LD	HR	LR	HD	LD
C-I	MA	1.9	3.4	3.6	6.2	2.2	1.5	1.0	1.9	2.1	1.1	0.6	1.1
	LA	5.4	1.5	6.0	2.5	2.3	1.0	1.1	0.9	1.7	0.8	0.6	0.9
	MinPI↓	2.1	0.6	1.4	1.0	1.0	0.7	1.1	0.9	0.4	0.9	0.7	0.8
	MinPI↑	2.2	4.3	5.0	7.9	1.0	1.1	1.2	1.2	1.0	0.9	0.8	1.0
	MaxPI↓	1.7	2.7	1.6	2.2	0.9	1.4	0.9	1.3	0.5	0.6	0.6	0.6
	MaxPI↑	3.7	3.2	5.2	6.7	1.0	1.1	0.8	1.1	1.4	1.0	1.0	0.9
C-II	MA	12.4	9.3	10.6	10.5	5.8	4.5	6.7	4.3	3.8	3.0	4.2	3.2
	LA	19.1	14.2	12.3	12.9	5.7	4.9	4.2	4.7	5.0	2.0	3.9	2.7
	MinPI↓	8.1	5.9	5.2	5.2	2.2	6.5	1.4	5.5	2.0	1.0	1.4	0.8
	MinPI↑	8.2	9.9	10.5	10.1	4.0	2.0	4.7	2.1	3.0	1.5	2.8	1.5
	MaxPI↓	6.7	5.7	4.8	4.2	4.7	4.5	1.4	3.2	1.5	1.9	1.3	1.8
	MaxPI↑	7.7	9.0	10.1	9.2	3.8	3.0	3.6	3.4	2.6	2.9	3.2	2.8
C-III	MA	21.2	28.8	27.2	18.4	19.6	20.7	13.7	19.1	15.9	18.6	14.8	19.8
	LA	17.7	27.7	30.0	24.7	18.7	14.5	15.6	14.1	19.9	14.4	15.1	16.0
	MinPI↓	16.5	18.1	20.1	15.4	16.7	11.1	10.8	11.2	16.1	11.4	13.9	11.5
	MinPI↑	19.5	29.1	23.8	23.6	16.7	16.3	12.8	16.9	17.1	15.8	12.8	15.4
	MaxPI↓	15.5	21.5	13.4	19.5	17.1	12.9	11.9	16.8	13.7	14.8	13.8	17.0
	MaxPI↑	22.4	26.0	27.0	23.5	16.5	11.8	11.9	11.4	16.4	12.7	12.7	12.4

Experiment 3 Explore portfolios where constraint groups are prioritized.

Experiment 4 Investigate the impact of *reboots*.

Experiment 5 Investigate how results carry over to MDD propagators with other constraints.

Experiment 1: Role of Y and W. First, we evaluate the performance of the Y and W heuristics on three "nurse rostering" problems from [16], which ask to schedule nurse work shifts over a horizon of 40 days, subject to a collection of AMONG constraints. There are three classes of instances: Class C-I requires at most 6 out of 8 consecutive work days and at least 22 out of 30 consecutive work days. C-II uses 6 out of 9 and 20 out of 30, while C-III uses 7 out of 9 and 22 out of 30. Each instance also requires 4 or 5 work days each week.

The portfolio was set to use $[\langle Y, \widetilde{Q}, W\rangle, \langle Y, \overline{Q}, W\rangle]$. Namely, layer refinement is driven by approximate equivalence first, followed by strict equality when space is still available. Y and W are selected among the following options:

HR Define $Y(s) = \texttt{rank}(s)$ to select the most recent state first.

LR Define $Y(s) = -\texttt{rank}(s)$ to select the oldest state first.

HD Define $Y(s) = |\delta^-(s)|$ to select the state with largest in-degree.

LD Define $Y(s) = -|\delta^-(s)|$ to select the state with lowest in-degree.

MA $W(s) = -|\delta^-(s)|$ ranks nodes according to decreasing arc set cardinality.

LA $W(s) = |\delta^-(s)|$ ranks nodes according to increasing arc set cardinality.

MinPI↓ $W(s) = -\min_{p \in \delta^-(s)} \texttt{rank}(p)$ ranks nodes with decreasing age of oldest parent.

MinPI↑ $W(s) = \min_{p \in \delta^-(s)} \texttt{rank}(p)$ ranks nodes with increasing age of oldest parent.

MaxPI↓ $W(s) = -\max_{p \in \delta^-(s)} \texttt{rank}(p)$ ranks nodes with decreasing age of youngest parent.

MaxPI↑ $W(s) = \max_{p \in \delta^-(s)} \texttt{rank}(p)$ ranks nodes with increasing age of youngest parent.

Table 1 shows the CPU time taken for each combination of Y and W above. The state equivalence function used for approximate equivalence is from example 10 using $t = 3$, maximal reboot distance is 0 and maximum refinement is 10.

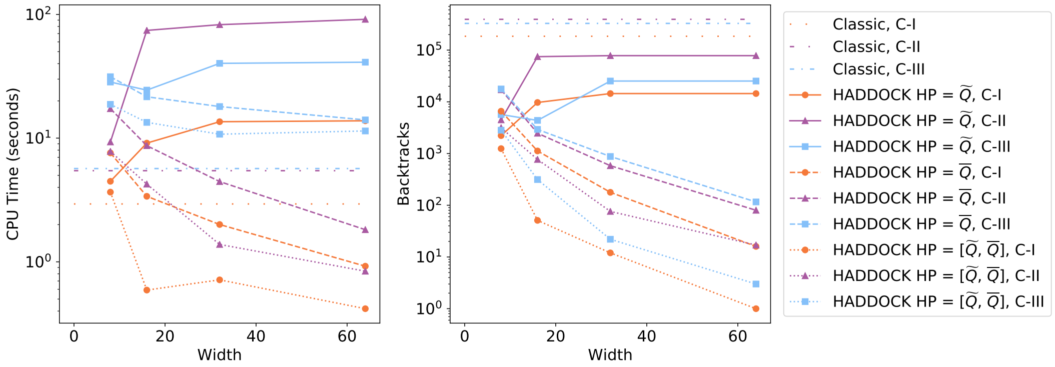


Figure 4 CPU time (left) and backtracks (right) for finding all solutions for `amongNurse` using different equivalence functions.

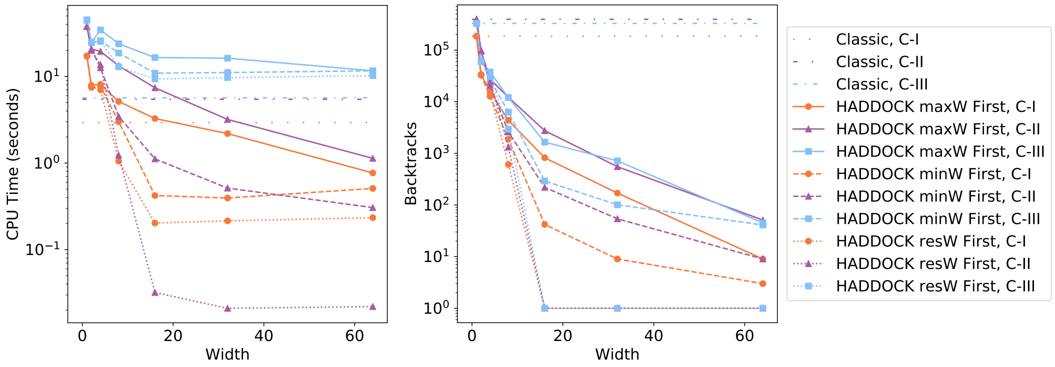


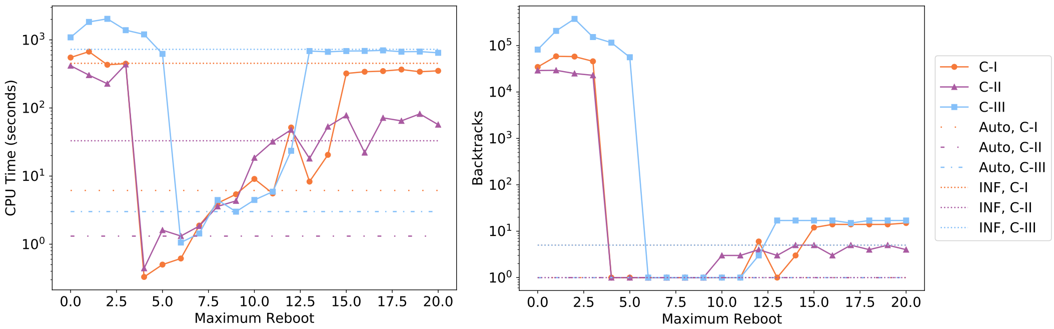
Figure 5 CPU time (left) and backtracks (right) for finding all solutions for `amongNurse` with different constraint group portfolios.

These results indicate that both Y and W have a clear impact on the method. While no single pair Y,W dominate, the LR option for Y seems to fare particularly well. Likewise, `MinPI↓` and `MaxPI↓` appear to be consistently effective. We also observe that implementing this generic heuristic framework introduces minimal, if not negligible, overhead.

Experiment 2: Role of $\widetilde{Q}$ vs. $\overline{Q}$. Consider the role of the two equivalence heuristics. Figure 4 graphs the shortest time and least number of backtracks when $\widetilde{Q}$ is used alone, $\overline{Q}$ is used alone, or as a portfolio $[\widetilde{Q}, \overline{Q}]$. At higher widths, the heuristic bundle with $\widetilde{Q}$ stagnates since the approximate equivalence prevents it from making full use of the width. The bundle using $\overline{Q}$ improves as the width increases, which is good. Yet, the best results come from the portfolio which suggest that coarser equivalence is helpful to more judiciously make use of the space in each layer and rely on the stricter $\overline{Q}$ when space is plentiful.

Experiment 3: Portfolio with constraint groups. Given the three classes of constraints that model different aspects (lower bounding the number of work days: $minW$, upper bounding the number of work days: $maxW$ and restricting the number of work days to 4 or 5 in any given week: $resW$) it is tempting to rely on 3 constraint groups and use a portfolio based on the three bundles of heuristics $\{H(minW), H(maxW), H(resW)\}$. To simplify, we test three portfolios: $minW$ First $([H(minW), H(maxW \wedge resW)])$, $maxW$

Figure 6 CPU time (left) and backtracks (right) for proving infeasibility for `Multiple AllDifferent` across different reboot values using $HP = \langle HR, \overline{Q}, MinPI\downarrow\rangle$.



First $([H(maxW), H(minW \wedge resW)])$, and $resW$ First $([H(resW), H(minW \wedge maxW)])$. Figure 5 shows the results while using $\langle LR, \overline{Q}, \texttt{MinPI}\downarrow\rangle$ for each bundle; the results are quite spread out. The best performance, on all of `C-I`, `C-II`, and `C-III`, occurs whenever $resW$ is the first entry in the portfolio, giving it the first opportunity to drive refinements.

The characteristics of constraints in $resW$ do explain such a behavior. First, these always have the *tightest* bounds ($l = 4$ and $u = 5$). Refining on the tightest constraints may give better opportunities for filtering. Second, the $resW$ constraint groups are always the smallest. Last, $resW$ constraints are stated over disjoint variable sets and since refinements occur on a layer basis (layers are associated to variables) the refinements are more *focused*.

Table 2 Multiple `AllDifferent` for different reboot values using $HP = \langle HR, \overline{Q}, MinPI\downarrow\rangle$ and a width of 16. Each row reports the fraction of full `reboots` and runtime (in seconds).

	`reboot`	1	2	3	4	5	6	7	8	Auto	INF
A-I	Full	39.2%	66.7%	54.5%	83.8%	94.3%	98.0%	98.1%	98.6%	99.5%	100%
	Time	671.6	430.7	447.0	0.3	0.5	0.6	1.9	4.0	6.2	452.5
A-II	Full	52.2%	66.1%	90.5%	82.5%	94.8%	97.3%	99.3%	99.4%	97.3%	100%
	Time	303.5	226.8	435.4	0.4	1.6	1.3	1.8	3.6	1.3	33.0
A-III	Full	48.8%	46.8%	27.3%	69.6%	66.7%	97.5%	99.1%	99.4%	99.5%	100%
	Time	1834.0	2036.0	1387.2	1202.5	622.6	1.0	1.4	4.4	3.0	725.8

Experiment 4: Reboot for Multiple AllDifferent. The assessment of the `reboot` heuristic is done with randomly generated CSPs that use `allDifferent` constraints, are infeasible and take a non-negligible amount of time to solve with a classic solver. The generator uses the parameters $\langle n, d, [(s_1, f_1, p_1), \ldots, (s_k, f_k, p_k)]\rangle$ where n is the number of variables, d is the domain size, and each (s_i, f_i, p_i) tuple describes a single group of constraints. Group i uses (s_i, f_i, p_i) to produce a set of `AllDifferent` constraints. Each constraint c_k in that set ranges over a random subset (of size ≥ 2) of variables sampled from $\{x_{k \cdot f_i + 1}, \ldots, x_{k \cdot f_i + s_i}\}$ where each variable has a probability p_i of being included. Three instances (available online at `http://hidden.url.domain`) were created from $\langle 50, 7, [(3,1,1), (6,6,1), (10,1,.3), (8,5,.6), (20,7,.2)]$. Performance is measured with time and backtracks to prove infeasibility.

Figure 6 shows the performance using a heuristic bundle of $\langle HR, \overline{Q}, MinPI\downarrow\rangle$ for different maximum reboot values with INF representing an unlimited reboot. A dramatic improvement in performance occurs around `reboots` between 4 and 6 that gets erased as the maximum reboot increases. When a reboot occurs, the refinement either moves as far back as possible or is stopped by the maximum reboot distance (Algorithm 3, line 7). To shed light on Figure 6, consider Table 2 that gives the percentage of *full* reboots across all calls to `computeDown`

Table 3 CPU time (sec.) to prove infeasibility for Multiple `AllDifferent` using $\overline{Q}$ for different Y (columns) and W (rows) heuristics with the MDD $width = 16$.

		HR	LR	HD	LD
A-I	MA	755.98	920.56	899.14	917.74
	LA	746.80	939.54	925.50	933.94
	MinPI↓	0.91	0.91	0.90	.91
	MinPI↑	795.84	949.96	923.89	935.30
	MaxPI↓	0.90	0.92	0.91	0.92
	MaxPI↑	808.84	961.56	923.37	931.59
A-II	MA	224.45	311.54	304.52	302.08
	LA	228.62	318.84	303.10	308.09
	MinPI↓	1.28	1.29	1.33	1.28
	MinPI↑	203.46	267.36	260.42	270.50
	MaxPI↓	1.29	1.29	1.29	1.31
	MaxPI↑	206.10	268.60	259.29	261.77
A-III	MA	2594.93	3240.10	3553.28	3546.93
	LA	2595.43	3138.61	3481.61	3622.81
	MinPI↓	1.00	390.37	0.87	0.89
	MinPI↑	2420.01	2926.05	3256.71	3316.82
	MaxPI↓	0.98	375.55	0.87	0.85
	MaxPI↑	2507.99	2938.20	3275.93	3321.35

Table 4 AIS $(n = 11)$ for different reboot with $HP = \langle HR, \overline{Q}, MinPI\downarrow\rangle$ and $width = 16$.

`reboot`	0	1	2	3	4	5	6	7	8	`Auto`	INF
Total	0%	14%	44%	54%	66%	75%	85%	87%	93%	49%	100%
CPU Time	5.99	7.29	6.24	7.85	6.72	8.79	9.14	8.77	12.38	6.31	16.12
Backtracks	2960	3682	2672	2848	2187	2280	1735	2030	633	2416	13

during the search, that is, reboots that were not cut short. The gains occurs when around $80 - 90\%$. By the time $reboot = 7$, 98% of reboots are full meaning any further increase is unlikely to improve refinements but may still add overhead. In the benchmarks, each `AllDifferent` constraint has at most 7, sometimes fewer, variables. Hence, the reboot *may benefit from staying within the scope of the constraint.* A tempting `Auto` strategy for limiting reboots for any variable x_i associated to layer $\mathcal{L}_i$ is as follows. As usual, let $vars(c)$ denote the set of variables appearing in c and $cstr(x)$ be the set of constraints mentioning variable x. Let $\mathcal{L}(x)$ be the layer of variable x. Then,

$$related(x_i) = \bigcup_{c \in cstr(x_i) \;\mid\; |vars(c)| \leq \frac{|X|}{2}} vars(c)$$

in $reboot(i) = \min_{y \in related(x_i)} index(\mathcal{L}(y))$ denotes the layer that the propagator should return to when refinement aborts early. The rationale is to consider the shallowest layer of variables directly related to x_i provided that the constraint connecting them does not cover a majority of the variables in the CSP. Figure 6 and Table 2 give the results. While the `Auto` strategy does not beat the best static reboot value shown, it performs quite well and avoids the risk of setting the maximum reboot too small or too large.

Experiment 5: Similarities across benchmarks. Last, we check how the heuristics behave across benchmarks. Table 3 gives results for different Y and W using the All Different benchmarks with a reboot of 6. While $MinPI\downarrow$ and $MaxPI\downarrow$ are again the clear favorites for W, HR appears to be the best option for Y. This differs from Nurse Rostering and underlines the usefulness of having programmable heuristics.

To assess whether `Auto` performs on other benchmarks, it is tested on the *All-Interval Series* problem (#007 on CSPLIB) measuring the time, number of backtracks, and percentage of full reboots when looking for all solutions. Table 4 shows the results with $n = 11$. `Auto` picks a good compromise somewhere between 2 and 3 which matches the arity of the absolute value constraints. Using an infinite reboot pays off in backtracks, but not in run time.

7 Conclusion

Heuristics can have a significant impact on the filtering ability of an MDD propagator and ultimately on the efficiency of a model. This paper introduces several heuristics that govern such behaviors, formalized their integration into a generic framework, and reported on the impact they have in practice. Interestingly it led to an *automatic* setting for the `reboot` heuristic. The keystone of the paper is the recognition that such heuristics should be user programmable to get the most out of decision diagram technologies.

References

1 Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33, 2005. `doi:10.1016/j.orl.2004.04.002`.

2 H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A Constraint Store Based on Multivalued Decision Diagrams. In *Proceedings of CP*, volume 4741 of *LNCS*, pages 118–132. Springer, 2007.

3 David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William Cook. Finding cuts in the tsp. *Annals of Physics*, 54, 1995.

4 N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1994.

5 Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290:405–421, April 2021. `doi:10.1016/J.EJOR.2020.07.063`.

6 D. Bergman, A. A. Cire, W.-J. van Hoeve, and J. N. Hooker. *Decision Diagrams for Optimization*. Springer, 2016.

7 Raphaël Boudreault and Claude-Guy Quimper. Improved cp-based lagrangian relaxation approach with an application to the tsp. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI*, volume 21, pages 1374–1380, 2021.

8 Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre. Revision ordering heuristics for the constraint satisfaction problem. In *First International Workshop: Constraint Propagation and Implementation*, 2004. URL: `http://www.cril.univ-artois.fr/~lecoutre/research/publications/2004/CPW2004.ps`.

9 T Fahle and M Sellman. Cost based filtering for the constrained knapsack problem. *Annals of Operations Research*, 115:73–93, 2002.

10 J. M. Gauthier and G. Ribière. Experiments in mixed-integer linear programming using pseudo-costs. *Mathematical Programming*, 12, 1977. `doi:10.1007/BF01593767`.

11 R. Gentzel, L. Michel, and W.-J. van Hoeve. Haddock: A language and architecture for decision diagram compilation. In *Principles and Practice of Constraint Programming. CP 2020*, volume 12333 of *Lecture Notes in Computer Science*, pages 531–547. Springer, Cham, 2020.

12 X. Gillard, P. Schaus, and Coppé. Ddo, a Generic and Efficient Framework for MDD-Based Optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2020.

13 Gilles Pesant Gilles, Claude Guy Quimper, and Alessandro Zanarini. Counting-based search: Branching heuristics for constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 43, 2012. `doi:10.1613/jair.3463`.

14 T. Hadžić, J. N. Hooker, B. O'Sullivan, and P. Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In P. J. Stuckey, editor, *Principles and Practice of Constraint Programming (CP 2008)*, volume 5202 of *Lecture Notes in Computer Science*, pages 448–462. Springer, 2008.

15 R M Haralick and G L Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

16 S. Hoda, W.-J. van Hoeve, and J. N. Hooker. A Systematic Approach to MDD-Based Constraint Programming. In *Proceedings of CP*, volume 6308 of *LNCS*, pages 266–280. Springer, 2010.

17 R. M. Keller. Formal Verification of Parallel Programs. *Communications of the ACM*, 19(7):371–384, 1976.

18 M. Lagerkvist and C. Schulte. Propagator groups. In Ian Gent, editor, *Fifteenth International Conference on Principles and Practice of Constraint Programming, Lisbon, Portugal*, volume 5732 of *Lecture Notes in Computer Science*, pages 524–538. Springer-Verlag, 2009.

19 Laurent Michel, Pierre Schaus, Pascal Van Hentenryck. MiniCP: A lightweight solver for constraint programming, 2018. Available from `https://minicp.bitbucket.io`.

20 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for sat solvers. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710, 2016. `doi:10.1007/978-3-319-40970-2_9`.

21 Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In Nicolas Beldiceanu, Narendra Jussien, and Éric Pinson, editors, *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems*, volume 7298 of *Lecture Notes in Computer Science*, pages 228–243. Springer Berlin Heidelberg, 2012. `doi:10.1007/978-3-642-29828-8_15`.

22 Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001. `doi:10.1145/378239.379017`.

23 Philippe Refalo. Impact-based search strategies for constraint programming. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 557–571. Springer, 2004. URL: `http://springerlink.metapress.com/openurl.asp?genre=article{&}issn=0302-9743{&}volume=3258{&}spage=557`.

24 Meinolf Sellmann, Thorsten Gellermann, and Robert Wright. Cost-based filtering for shorter path constraints. *Constraints*, 12, 2007. `doi:10.1007/s10601-006-9006-4`.

25 Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219, 2015. `doi:10.1016/j.artint.2014.11.006`.

An Auditable Constraint Programming Solver

Stephan Gocht
Lund University, Sweden
University of Copenhagen, Denmark

Ciaran McCreesh
University of Glasgow, UK

Jakob Nordström
University of Copenhagen, Denmark
Lund University, Sweden

Abstract

We describe the design and implementation of a new constraint programming solver that can produce an auditable record of what problem was solved and how the solution was reached. As well as a solution, this solver provides an independently verifiable proof log demonstrating that the solution is correct. This proof log uses the VeriPB proof system, which is based upon cutting planes reasoning with extension variables. We explain how this system can support global constraints, variables with large domains, and reformulation, despite not natively understanding any of these concepts.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Discrete optimization

Keywords and phrases Constraint programming, proof logging, auditable solving

Digital Object Identifier 10.4230/LIPIcs.CP.2022.25

Supplementary Material Source code for the solver described in this paper can be found here:
Software (Source Code): `https://doi.org/10.5281/zenodo.6514093`

Funding *Stephan Gocht*: Supported by the Swedish Research Council grant 2016-00782.
Ciaran McCreesh: Supported by a Royal Academy of Engineering research fellowship.
Jakob Nordström: Supported by the Swedish Research Council grant 2016-00782 and the Independent Research Fund Denmark grant 9040-00389B.

1 Why Trust a Constraint Programming Solver?

Proof logging is now a standard practice in the Boolean satisfiability (SAT) community: alongside a solution, solvers are expected to produce a proof, in a standard format called DRAT [19, 18, 33], that can be verified independently to ensure that the correct answer was reached through legitimate reasoning. As well as reducing the number of bugs in solvers, this has been vital for the social acceptability of computer-generated mathematical proofs [21]. These successes mean that proof logging is now being considered in other areas, including mixed integer programming [9] and subgraph-finding algorithms [14, 13], and a similar paradigm known as *certifying algorithms* exists for polynomial-time solvable problems [23].

We believe that a practical proof logging system would also be extremely useful for the constraint programming (CP) community. In the 2021 MiniZinc challenge, at least 45 out of 3,500 claimed solutions were incorrect (either through falsely claiming unsatisfiability or optimality, or by providing infeasible "solutions"), and previous years saw similar rates. Furthermore, this was not limited to one solver, one problem, or one global constraint. Although this high error rate does not necessarily reflect what we might see in practice, it strongly suggests that we should not be complacent. And even if we are completely convinced that our solvers are correct, thanks to extensive testing using domain-specific methods [1, 12],

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 25; pp. 25:1–25:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

there are still benefits to be had from proof logging. When CP is used for life-affecting decision-making, having a solver that can produce an independently verifiable record of what the problem was and how it was solved would be much better for public confidence in algorithms than saying "trust us, we tested it carefully". In effect, we would be making the solving process *auditable*, and removing the need for trust.

In some applications, compiling a CP model to SAT and re-using SAT proof logging might be a viable approach for auditability. However, this is not a universal solution: even if the loss of solving power from switching representations is not a problem, why should we trust that a complex compilation process is correct? And what if we need to solve enumeration or optimisation problems, neither of which are supported by DRAT? Nor is it practical to make CP solvers output DRAT proofs, even for decision problems: attempts at expressing the strong reasoning carried out by simple global constraints like all-different have introduced intolerable overheads [28, 10], and DRAT does not seem well-suited even for the parity reasoning done by some modern SAT solvers [15]. One alternative would be to introduce a much stronger and more complex proof format, that is aware of the meaning of every global constraint and every kind of propagation that could be performed [31] and every kind of reformulation ever invented – but why should we trust such a complicated proof logging system, and how would we even know that it is consistent? This is not a trivial concern: even the relatively simple DRAT format has had issues in this respect [26].

This paper describes an alternative approach to proof logging which addresses all of these problems. It uses an existing proof verifier, VeriPB, which was designed for pseudo-Boolean models. VeriPB's proof format uses cutting planes reasoning [5] and redundance-based strengthening [15], which is only a small step up in complexity from the DRAT approach of using Boolean models and extended resolution. However, this small change in underlying proof system suddenly means that proof logging for many kinds of constraint becomes both efficient and easy to implement, despite the system not having any explicit notion of global constraints or even non-binary variables. Thanks to this, we have been able to implement a new prototype constraint programming solver which can produce proof logs for all of its reasoning, with support for global constraints like all-different, integer linear inequality (including for variables with very large domains), table, minimum / maximum of an array, element, and absolute value, as well as some simple automatic reformulation.

Our aim in this work is not to produce the world's fastest solver, but rather to explore the design decisions necessary to provide auditable solving when operating with diverse constraints, and to explain how to understand and adopt proof logging technology. The main differences between our solver and a basic conventional solver such as MiniCP [24] are:

- The solver can describe the semantics of variables and each constraint in a low-level format, which is discussed in Section 2.1. We give examples in Section 3. This is the only part of the process that is not directly auditable: we discuss this further in Section 5.1.
- Whenever the solver backtracks during search, it creates a proof step asserting that its current sequence of guesses "obviously" (but verifiably) implies a contradiction, as described in Section 2.3. When enumerating or optimising, solutions must also be logged.
- Any piece of code that potentially removes a value from a variable's domain (or instantiates it, or changes its bounds) must be able to provide a justification that can be added to the proof log. This justification can be "this is immediately obvious", "use reverse unit propagation" (Section 2.3), or occasionally, "use the following explicit proof steps" (Section 2.2). In many ways this resembles lazy clause generation solvers [25], except that justifications must be derived in a sound and verifiable manner, rather than being introduced from nowhere. We give examples in Section 4.
- Finally, some constraints make use of reformulations, which must also be justified; examples are in Section 4.5.

Together, these additions mean that if the solver ever produces an incorrect answer, this can be detected – even if it is due to a compiler or hardware fault rather than a solver bug. Our results demonstrate that proof logging for constraint programming is rapidly becoming a technologically viable approach, and one that will be worth adopting in other solvers. We conclude by outlining how we might realise this goal of auditable constraint solving.

2 The VeriPB Proof System

We begin with an overview of the relevant parts of the model and proof system used by VeriPB.[1] It is important to stress immediately that solvers do not need to understand or implement this proof system (in the same way that most SAT solvers do not "know" that they are searching for resolution proofs). Indeed, the prototype solver we describe later in this paper produces proofs through templates, never manipulates proof steps directly, and does not know enough about the VeriPB proof system to be able to verify its own proofs.

In this section we will primarily be talking about proofs of unsatisfiability. Both VeriPB and our solver also support optimisation, enumeration, and satisfiable decision problems, but the core of the proof system concerns unsatisfiability. The idea behind a proof is that we start off with known facts, which come from the input model. Then, at each step in the proof, we derive a new fact which is "obviously" a consequence of some combination of previous facts. We finish by deriving a fact which is clearly a contradiction, which in turn means it must be the case that the input is unsatisfiable.

2.1 Pseudo-Boolean Models

VeriPB takes as input a pseudo-Boolean model, which is a very restricted kind of constraint programming model. A pseudo-Boolean model is defined by a set $\{x_i\}$ of $\{0, 1\}$ integer variables, and a set of integer linear inequalities $\sum_i c_i x_i \geq n$ for integers c_i and n. In this paper we will use lowercase variable names to refer to pseudo-Boolean variables, and uppercase variable names to refer to constraint programming variables. We will also write some constraints using $\leq$ instead of $\geq$, and will write $\sum_i c_i x_i = n$ as shorthand for two inequalities. We use the convention that $x = 0$ means false, and $x = 1$ means true; we write $\overline{x}$ to mean $1 - x$. Observe that Boolean satisfiability constraints in conjunctive normal form (CNF) can easily be written as pseudo-Boolean constraints, because e.g. $(x_1 \vee \overline{x}_2 \vee x_3)$ holds if and only if $(x_1 + \overline{x}_2 + x_3 \geq 1)$. For clarity we will sometimes mix logical and pseudo-Boolean notation, and write expressions like $(x_1 \wedge x_2 \wedge x_3) \rightarrow (2x_4 + 3x_5 + -4x_6 \geq 7)$ rather than the more cumbersome $11\overline{x}_1 + 11\overline{x}_2 + 11\overline{x}_3 + 2x_4 + 3x_5 + -4x_6 \geq 7$.

There is a standard textual file format for pseudo-Boolean models, known as OPB [27]. VeriPB supports this format, with extensions: for example, it allows variables to have descriptive names, which is convenient for readability, and can include implications to avoid the need for solver authors to calculate appropriate coefficients manually.

2.2 Cutting Planes

Alongside an OPB file, VeriPB takes a proof log file that claims to show that the pseudo-Boolean model is unsatisfiable, and checks the proof's validity. This proof log is a text file, which describes a sequence of steps using the cutting planes proof system [5]. In cutting

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

planes, we can add two constraints together, multiply a constraint by a non-negative integer constant, and divide existing constraints by a positive integer constant (with rounding); we may also assert that any literal is non-negative. The aim is to derive a constraint saying that $0 \geq 1$, which serves as a contradiction. The cutting planes proof system is complete for pseudo-Boolean models, in the same way that resolution is complete for Boolean models. However, it is exponentially stronger than resolution: for example, resolution requires exponential length proofs for all-different constraints, whereas cutting planes can justify Hall set reasoning in (small) polynomial length [10]. For more details on the theoretical background, see, e.g., the survey by Buss and Nordström [3].

2.3 Unit Propagation and Reverse Unit Propagation

For solver authors, working directly with cutting planes can be difficult, and would require every part of a solver to keep careful track of every operation carried out. This difficulty can be avoided through the use of *reverse unit propagation* (RUP) proof steps [16, 30, 10], which are in effect shorthand for a sequence of cutting planes steps.

For CNF clauses, *unit propagation* means identifying any clause where all but one of its literals has already been set the wrong way, and propagating the remaining literal to the value that avoids violating the clause, repeating until either a contradiction is reached or no further unit clauses exist. This notion generalises to pseudo-Boolean constraints, where unit propagation means achieving integer bounds consistency [4]. A constraint C is said to be RUP if asserting its negation leads to a contradiction via unit propagation; in such a case, it is obviously permissible to introduce C as a new constraint without altering whether the underlying model is satisfiable.

RUP steps in a Boolean setting form the core part of the DRAT proof format. This is useful for solver authors because for a typical CDCL SAT solver, every learned clause is RUP, and so writing a proof log requires only that a solver output every clause it learns in turn. In our constraint programming setting, RUP clauses will similarly form the backbone of the proofs we generate, with a RUP clause being written every time a solver backtracks. However, we will also use explicit cutting planes steps where necessary, to justify complex propagations. In one sense, RUP is purely a convenience for solver authors, in that with more work, cutting planes steps could be used instead; however, this would require substantially more book-keeping in the solver.

The following pieces of intuition may be helpful in what follows: a fact follows from unit propagation if it is so immediately obvious that it is not worth stating. A fact follows from reverse unit propagation if, once you have been told that it is a fact, it is obviously true (but that it might not be immediately obvious if you are not told). In some ways this resembles failed literal probing, or the difference between generalised arc consistency and singleton arc consistency; this intuition may become clearer following the example in Section 4.1.

2.4 Extension Variables and Redundance-Based Strengthening

An *extension variable* is a new variable introduced as part of a proof. In VeriPB, extension variables are supported using a rule called *redundance-based strengthening* (which, for readers familiar with SAT proof logging, is the natural analogue of the RAT rule in DRAT) [15]. We do not need the full power or definition of that rule for this paper. It suffices to say that, for an arbitrary pseudo-Boolean constraint C and a new variable y that has not previously appeared in the model or proof, we are allowed to introduce the reified constraints $y \leftrightarrow C$ at any point during the proof. As well as being extremely convenient for solver authors, extension variables also give an exponential increase in reasoning power [3].

2.5 Satisfiable Instances, Enumeration, Optimisation, and Deletions

For satisfiable decision problems, VeriPB supports solution checking by including a solution in a proof log. Enumeration problems may also be verified this way: whenever a solution is logged, VeriPB treats this as introducing a new constraint saying "but not this solution", and so a proof is effectively a proof by contradiction that there are no solutions other than the ones listed. Optimisation is handled similarly, via an optional objective expression in the OPB file. Finally, in practice it is important to delete constraints from the proof that will not be re-used later on. This is straightforward, but will not be discussed in this paper.

3 Encoding Constraint Programming Models

In the previous section, we learned that if we wish to use VeriPB to verify constraint programming proofs then we must provide two things: a pseudo-Boolean model in OPB format, and a proof log. We now discuss how the first of these two steps may be generated by a CP solver, looking first at how we turn CP variables into pseudo-Boolean variables, and then at how we represent constraints. When compiling CP to a lower level format for solving, selecting a good encoding involves considering propagation and consistency; in contrast, for proof logging we need only something that is simple and reasonably compact.

3.1 Variables

The most straightforward way of encoding an integer variable X with domain $\ell \dots u$ is to create $u - \ell + 1$ pseudo-Boolean variables $x_{=i}$, where $x_{=i}$ is true if and only if $X = i$, together with supporting constraints saying that $\sum x_{=i} = 1$. Such an approach was used for proof logging the all-different constraint by Elffers et al. [10]. However, this is impractical for variables with large domains that are only involved in bounds-consistent constraints such as integer linear inequalities. Instead, we define a binary encoding. Let h be the least strictly positive number such that $2^{h-1} \geq \max(1, |u|, |\ell|)$. Then we introduce variables $x_{\mathrm{b}i}$ for $i \in 0 \dots h-1$, and, if $\ell < 0$, we additionally introduce an x_{neg} variable to give us a two's complement style representation. The two constraints $\ell \leq -2^h x_{\mathrm{neg}} + \sum_{i=0}^{h-1} 2^i x_{\mathrm{b}i} \leq u$ then define the meaning and bounds of these variables (with the leading sum term omitted if $\ell \geq 0$). For example, if we have a constraint programming variable A with domain $\{-3 \dots 9\}$, we would define

$$\begin{aligned} -32a_{\mathrm{neg}} + 1a_{\mathrm{b}0} + 2a_{\mathrm{b}1} + 4a_{\mathrm{b}2} + 8a_{\mathrm{b}3} + 16a_{\mathrm{b}4} &\geq -3 \text{ and} \\ 32a_{\mathrm{neg}} + -1a_{\mathrm{b}0} + -2a_{\mathrm{b}1} + -4a_{\mathrm{b}2} + -8a_{\mathrm{b}3} + -16a_{\mathrm{b}4} &\geq -9. \end{aligned}$$

Although compact, experienced modellers know that such an encoding often leads to extremely poor propagation. This is a problem if the encoding is to be used for solving, but for proof logging this is not an issue because the encoding only restricts how we write a proof, not how a solution is reached. However, when expressing constraints or propagation, it is often useful to be able to use variables $x_{=i}$ and $x_{\geq i}$ for selected values of i. If these variables are used when the constraints appear in the pseudo-Boolean model, we can define them immediately. We have found the most convenient way of expressing these variables to be

$$x_{\geq i} \leftrightarrow -2^h x_{\mathrm{neg}} + \sum_{i=0}^{h-1} x_{\mathrm{b}i} \geq i$$

and similarly for $x_{\geq i+1}$. Additionally, we constrain that $x_{\geq i+1} \rightarrow x_{\geq i}$, and force $x_{\geq i}$ to be true or false respectively if i defines a lower or upper bound. We then define $x_{=i} \leftrightarrow x_{\geq i} \wedge \overline{x}_{\geq i+1}$.

However, what if these values are only used for branching or propagation, such as when dealing with linear inequalities (discussed below)? The whole point of using a binary encoding was to avoid having to define variables for values that never appear in a constraint or a proof. Fortunately, it is possible to introduce these additional variables as extension variables with the same defining constraints, so long as it is done in exactly the order described above. In such a case, we also introduce the RUP constraints $x_{\geq i} \rightarrow x_{\geq j}$ and $x_{\geq h} \rightarrow x_{\geq i}$ for the closest two values h and j that already have equality variables, if they exist. Finally, when propagating certain constraints such as all-different, it is also convenient to have an at-least-one constraint $\sum_{i=\ell}^{u} x_{=i} \geq 1$. If all the $x_{=i}$ variables are defined, then this constraint can also be introduced via RUP as needed, and does not need to be defined in the model. This can make the pseudo-Boolean model much more manageable: for example, for the implementation of the "cake" problem discussed in Section 5, our solver introduces a total of one hundred $x_{=i}$ or $x_{\geq i}$ variables in the proof, rather than defining several hundred thousand of them in the OPB file.

Note finally that the details of this encoding are largely irrelevant to most constraints. In particular, it is possible for the part of a solver that deals with proof logging to treat 0/1 variables separately with almost no impact on the rest of its code.

3.2 Constraints

Next, we must represent every constraint in pseudo-Boolean form. This topic is relatively well-understood, because pseudo-Boolean constraints are a superset of CNF – and again, it is not necessary to find a *good* encoding, only a simple and correct one. We now give some examples that illustrate general concepts.

Integer linear inequalities. Integer linear inequalities can easily be expressed in pseudo-Boolean form by adding multiples of the bit encodings together. For example, suppose we have the CP constraint $2A+3B+4C \leq 42$, where each of the variables has domain $\{-3 \ldots 9\}$. This would be translated into

$$\begin{aligned}
&-64a_{\text{neg}} + 2a_{\text{b0}} + 4a_{\text{b1}} + 8a_{\text{b2}} + 16a_{\text{b3}} + 32a_{\text{b4}} \\
&+ -96b_{\text{neg}} + 3b_{\text{b0}} + 6b_{\text{b1}} + 12b_{\text{b2}} + 24b_{\text{b3}} + 48b_{\text{b4}} \\
&+ -128c_{\text{neg}} + 4c_{\text{b0}} + 8c_{\text{b1}} + 16c_{\text{b2}} + 32c_{\text{b3}} + 64c_{\text{b4}} >= 42.
\end{aligned}$$

We may use a pair of such constraints to define equality and sum constraints. If we were solving using these constraints, we would get very weak propagation, but we will explain why this does not matter in the following section.

Not equals. Not equals constraints can be expressed using two *half-reified* linear inequalities: we introduce a Boolean flag f, and define the constraints $f \rightarrow (A-B > 0)$ and $\overline{f} \rightarrow (B-A > 0)$. These can be expressed in pseudo-Boolean form as integer linear inequalities with the addition of a suitably large coefficient on the negation of the flag to handle the implication. A similar encoding can be used for the absolute value constraint.

All-different. All-different can be expressed by a set of at-most-one constraints, such as $a_{=2} + b_{=2} + c_{=2} \leq 1$, or by a clique of not-equals constraints. Again, solving using either kind of constraint would give weaker propagation than the usual GAC all-different constraint, but this is not a concern for proof logging.

Table constraints. Table constraints can be expressed in terms of an auxiliary variable, which selects which tuple is matched. For example, given the tuple sequence $[(1,2,3),(1,3,4),(2,2,5)]$ applied as a table constraint to the variables $[A,B,C]$, we could express this by adding an auxiliary variable $T \in \{0\dots 2\}$ (called the *tuple selector variable*), and using implication constraints $t_{=0} \to (a_{=1} \wedge b_{=2} \wedge c_{=3})$ etc.

Element constraints. Element constraints come in a variety of forms. For example, consider a 2D element from constants constraint, which says that a variable V takes the $[I,J]$th entry in a two dimensional $m \times n$ array A of constants. This shows up in various places, such as the MiniCP quadratic assignment problem benchmark discussed in Section 5 (where solvers are expected to achieve generalised arc consistency on the two index variables, and bounds consistency on the assigned variable) [24]. The only constraints we will define are the unary constraints $i_{\geq 0}$, $\overline{i}_{\geq m}$, $j_{\geq 0}$ and $\overline{j}_{\geq n}$, and then $(i_{=x} \wedge j_{=y}) \to v_{=A[x,y]}$ for each array entry. Such constraints, on their own, obviously do not enforce the desired consistency levels, but they have the advantage of being simple. This technique also generalises. For example, if the array A is not constant then the implication constraints can become half-reified equalities instead – and this in turn makes it easy to define array minimum and array maximum constraints.

Other constraints. Other constraints are usually similarly easy to express. The critical point is that encodings need only be correct, not good, and so if we know how to express the constraint at all in CNF or as integer linear inequalities then that is sufficient. Similarly, if a constraint easily fits in a table, then it can be handled that way. Combined with the ability to use auxiliary variables, even constraints like “forms a connected subgraph” are manageable [13].

4 Proofs for Search and Propagation

The core of a proof for an unsatisfiable constraint satisfaction problem is a description of the solver’s search tree. This is expressed as a RUP statement for every backtrack, and ends with a contradiction when we backtrack from the root node. The idea is that whenever the constraint solver backtracks, it should be “obvious” that the sequence of guesses made leads to a dead end, and is thus a RUP clause. Gocht et al. [13] provide a worked example of this process in a branch and bound setting for a clique algorithm.

In order to make this process work with global constraints, we will need to include additional proof statements to justify non-obvious propagations (in the same way that Gocht et al. had to justify the clique algorithm’s bounds). The core invariant we use is that at every backtrack, any variable-value deletion that is known to the CP solver (and thus part of the decision to backtrack) must be visible to the proof verifier either through unit propagation, or through reverse unit propagation of the backtrack clause. This section elaborates on what this means for various global constraints.

4.1 RUP Justifications and Table Constraints

Achieving generalised arc consistency for table constraints involves two kinds of inference: detecting when a tuple becomes infeasible, and detecting when a variable’s value is no longer supported by any feasible tuple. There are several different propagation algorithms for performing this inference [29, 22, 7, 20, 32], but from a proof logging perspective it does not matter how the inference is performed, only what is inferred.

A tuple becoming infeasible requires no justification. Recall that tuples are defined with constraints like $t_{=0} \rightarrow (a_{=1} \wedge b_{=2} \wedge c_{=3})$. If, for example, A loses the value 1 from its domain, this constraint will unit propagate, setting $t_{=0}$ to false. This also holds when assignments are guessed: by the core invariant, asserting through RUP that the guessed assignments imply false will propagate the value loss, which in turn propagates the tuple becoming infeasible.

In contrast, suppose we have only two tuples supporting $A = 1$, and these are both made infeasible by other variables, so a solver infers $A \neq 1$. Let $\mathcal{G}$ be the set of equality variables corresponding to our current set of guessed assignments (for example, $\{b_{=2}, c_{=5}\}$). Then the assignment $\overline{a}_{=1}$ does *not* follow by unit propagation in the proof under the assertion of $\wedge\mathcal{G}$, which means it must be justified in some way. Fortunately, this is very simple, and we need only give a small hint to the proof verifier: we claim that $\wedge\mathcal{G} \rightarrow \overline{a}_{=1}$ will be a RUP constraint. Indeed, its negation is $(\wedge\mathcal{G}) \wedge a_{=1}$. Now consider each tuple t supporting $a_{=1}$ in turn. There must be some constraint derived that, under $\wedge\mathcal{G}$, falsifies a different variable in this tuple, which in turn forces the tuple selector variable to not equal t. Additionally, we know that A must take at most one value, and so for each $i \neq 1$, $a_{=i}$ will propagate to false; this in turn propagates every other tuple selector variable to false. Finally, we know that the tuple selector variable must take at least one value, but we have ruled out every value it could take, giving the desired contradiction.

Putting these facts together, the only proof logging needed for a table constraint is to log one RUP step whenever a variable loses a value due to lack of support. Intuitively, infeasibility of tuples is so obvious that we need not mention it. In contrast, loss of support is not immediately obvious to detect, but if we tell the proof verifier that it has in fact occurred then it is easy to check that we are telling the truth.

4.2 Explicit Justifications and Integer Linear Inequalities

Some constraints require more work. Elffers et al. [10] have already shown how both propagation and failure detection for the all-different constraint can be justified using cutting planes proofs. Their approach works in our setting, with one caveat: they require at-least-one constraints for certain CP variables, which we do not have in our model. However, recall that these constraints can be introduced using RUP where needed.

Integer linear inequalities are a similar case. Suppose we have a constraint $2A+3B+4C \leq 42$, with all three variables non-negative. In a typical CP solver, this constraint will achieve integer bounds consistency [17, 4]. As an example, suppose we know that under some set of guessed assignments $\mathcal{G}$ that $A \geq 5$ and $C \geq 3$, then a CP solver will infer that $\wedge\mathcal{G} \rightarrow B \leq 6$. We can derive this fact in a proof as follows. By assumption, we either have or can introduce RUP constraints that $\wedge\mathcal{G} \rightarrow a_{\geq 5}$ and $\wedge\mathcal{G} \rightarrow c_{\geq 3}$. This in turn means we either have or can introduce RUP constraints for the binary representation, saying that $\wedge\mathcal{G} \rightarrow \sum_{i=0}^{h-1} 2^i a_{\mathrm{b}i} \geq 5$ and $\wedge\mathcal{G} \rightarrow \sum_{i=0}^{h'-1} 2^i c_{\mathrm{b}i} \geq 3$ for appropriate values of h and h'. Now using cutting planes steps, we can multiply the first of these by 2 and the second by 4 (their coefficients in the original linear inequality), add both to the constraint defining the linear inequality, and divide the result by the coefficient of B, 3. It can be verified that the resulting mess is now sufficient to make $\wedge\mathcal{G} \rightarrow \overline{b}_{\geq 7}$ a RUP constraint. It is also routine to prove that this example generalises to arbitrary integer linear inequalities (although negative variables and / or coefficients require several awkward case by case analyses).

4.3 Element Constraints

Recall the special case of a two-dimensional element from constants constraint, where a variable V takes the $[I, J]$th entry in a two dimensional $m \times n$ array A of constants. In the interests of having a simple pseudo-Boolean encoding, we defined this using $(i_{=x} \wedge j_{=y}) \rightarrow v_{=A[x,y]}$ constraints. However, we wish for our solver to achieve generalised arc consistency on I and J, and bounds consistency on V. One way to proof log this reasoning is as follows. As a one-time operation at the start of search, we will use extension variables to turn this into a one-dimensional element constraint. We will introduce $m \times n$ new variables s_k, each of which is true if and only if a different $i_{=x} \wedge j_{=y}$ holds. We will then build an at-least-one constraint over the s_k variables via a sequence of $O(m \times n)$ RUP steps, as follows. For each value x for the first index variable I, we are going to derive via RUP that $\sum_k s_k + \overline{i}_{=x} \geq 1$. For this to hold, we first derive via RUP that for each value y for the second index variable J, that $\sum_k s_k + \overline{i}_{=x} + \overline{j}_{=y} \geq 1$. The desired at-least-one constraint now follows via RUP; in effect, we have performed an exhaustive backtracking search over the pair of variables I and J under the assertion that the desired at-least-one constraint does not hold, and shown that no solution satisfying I and J exists. From this point forwards, we are effectively dealing with a one-dimensional element constraint.

(Of course, one could ask why we convert from two dimensions to one dimension in the proof, and not in the model when we define the element constraint. We could certainly do things this way. However, our point here is to demonstrate that *we don't have to* handle model reformulations by changing the model: instead, we can use the most straightforward low-level model imaginable, and then prove that our reformulations are valid as part of the proof. We will explore this further below.)

We can also view our new encoding as being like a table constraint, but where the tuple selector variable is channeled to the two index variables. If we wished to achieve generalised arc consistency on the assigned variable V, we would simply reuse the inference techniques discussed in Section 4.1. However, this would require introducing a pseudo-Boolean equality variable $v_{=n}$ for each value in V's domain. This is potentially not what is desired, if the range of the constants is large and V is only otherwise used in a bounds-consistent manner. Therefore, instead of justifying that V does not take each value no longer present in feasible parts of A in turn, we would like to assert a bounds change using a $v_{\geq n}$ variable. This does not immediately follow by RUP on its own, although it will if we repeat the iteration technique used in creating the index variable, but iterating only over feasible array entries.

A downside to this approach is that it produces a proof containing $O(m \times n)$ steps to justify each bounds propagation. As Michel et al. [24] explain, by storing the array in a sorted manner, it is possible for a propagator to avoid looking at most array entries most of the time, and so have better than $O(m \times n)$ performance in the typical case. We suspect that this algorithm can be replicated in a proof efficiently, if we are prepared to establish a set of ordering constraints at the start of the proof.

Finally, an observant reader might have noticed that deletions on the one-dimensional array index will not unit propagate backwards to I and J. In fact, these deletions are RUP.

4.4 Not Equals

At this point, it should be clear how the not equals constraint can be handled: when one variable A is instantiated to a value v, it follows using RUP that the other variable B cannot also be v since the flag f would have to be both true and false to allow $f \rightarrow (A - B > 0)$ and $\overline{f} \rightarrow (B - A > 0)$ to hold simultaneously. However, there is another alternative, which we will see is more efficient in some scenarios. Instead of deriving under a sequence of guesses $\mathcal{G}$ that $\wedge\mathcal{G} \vee a_{=v} \rightarrow \overline{b}_{=v}$, we could simply introduce the RUP constraint $\overline{a}_{=v} + \overline{b}_{=v} \geq 1$, independently of the guesses. Propagation of the not equals constraint for v would then follow by simple unit propagation.

4.5 Autotabulation and Other Reformulations

Linear equality constraints can be defined and propagated as two linear inequalities. However, sometimes a solver may wish to achieve generalised arc consistency on a linear equality. This is NP-hard, but is still a good idea sometimes for small variables – for example, if $2X + 2Y + Z = 7$ then Z must be odd, but this will not be inferred from the inequalities. One way a solver might handle such constraints is by automatically turning the two linear inequalities into a table constraint. An implementation of this process might, of course, be buggy, and so we would like to prove that this tabulation is valid, rather than simply defining the table constraint in the pseudo-Boolean model. This is indeed possible, using a more advanced form of the kind of argument previously used to turn a 2D element constraint into a 1D element constraint.

Let us start by finding the set of solutions to the constraint. For each solution, we introduce an extension variable t_s which is true if and only if that solution is selected, in the same way as for a table constraint. We also introduce an extension variable g which is true if and only if at least one of these t_s variables is true. Next we perform and log a backtracking search to find all of the solutions to the constraint, except that we use g as an additional guessed assignment at every stage. At the end of the search, we have a proof that g must be true, which in turn gives us an at-least-one constraint over the t_s variables. We have now created all the constraints we need to define a table constraint.

We expect that similar techniques will be useful for many other kinds of reformulation as well, re-emphasising our ability to prove more than just the core solving process. One modelling technique that likely *cannot* easily be handled this way is symmetry breaking constraints. However, Bogaerts et al. [2] show that a slight extension to the VeriPB proof system would make this possible: this raises the intriguing possibility of taking a symmetry breaking lex or ordering constraint that is defined in a high level model, omitting it from the pseudo-Boolean model, and then efficiently proving that the constraint is in fact valid.

5 An Implementation

We have implemented a basic constraint programming solver which supports proof logging (see supplementary material). Our solver generally follows a conventional design, similar to MiniCP [24], although we have chosen for novelty reasons to make use of some modern C++ features like lambdas and variant types instead of a pure object-oriented design. We were not aiming for sheer speed, and so our solver does not include optimisations like multiple propagation queues, backtrackable variables, stateful or support-tracking propagators, or special handling of binary variables. The solver supports only integer variables, and implements the absolute value, all different, comparison (with half and full reification), element, linear equality and inequality, minimum and maximum, and table constraints. We include example programs implementing four of the five MiniCP benchmarks (a quadratic assignment problem, n-queens, magic series, and magic square; the TSP example is not included because we do not yet have a circuit constraint), as well as the MiniZinc cake optimisation problem[2], the classic "send more money" and Crystal Maze problems, the world's hardest Sudoku puzzle[3], and an odd-even sum problem using an auto-tabulated GAC sum constraint.

Throughout the development process, we have *not* tried particularly hard to produce a solver which is free from bugs. Instead, our goal is to produce a solver that will not produce undetectable incorrect outputs. The rest of this section describes the key aspects of the solver design that involve proof logging, discusses what we have learned from using the VeriPB system in a constraint programming setting, and evaluates its performance.

5.1 Constraint Compilation, or Why Trust the OPB File?

To create the OPB file, we use a single pass approach, outputting definitions as soon as variables and constraints are generated. Variables are handled centrally, whilst each constraint is responsible for providing its own pseudo-Boolean encoding. OPB creation is done purely using text, and the solver stores only the model line numbers for certain constraints – it does not explicitly store any pseudo-Boolean information.

An obvious difficulty with our proposed process is that this compilation from a CP model to an OPB model is not verified. This is somewhat offset by the deliberate use of extremely simple encodings, but one must ask: "why are the authors so sure that they have designed and implemented the encoding correctly, particularly for fiddly global constraints?". The answer to this question is that we are not sure at all, and so we rely upon a special test system, as follows. For a given constraint, we generate many different possible input domains for its variables. For each set of inputs, we use a small generate-and-test program that provides the full set of solutions to the constraint, making no use of the constraint programming solver or any clever logic or programming. (This can be moderately slow, for example for the element constraint.) We then use the constraint solver to solve the problem consisting of just that constraint on those inputs, and verify that the set of solutions found this way is identical. Finally, we verify the proof produced by this solving process: because this is an enumeration problem, this verifies that the OPB file also has exactly the same set of solutions (and additionally that the propagator found them all legitimately, although this is not the main point of the test).

This process is not perfect, but it does severely reduce the scope for errors: for example, it immediately flagged a typo where a reified greater than or equal constraint had accidentally been implemented as a reified greater than constraint, and a bug when the index constraint for an element constraint contained only out-of-range values.

5.2 Producing the Proofs

Recall that to produce a proof, we need to log our backtracking search, and certain variable-value eliminations. In the design of our solver, we opted for a careful separation of the notion of a variable and its current state: the former we represent as a handle, whilst the latter is stored separately in a central location to allow for easy backtracking. It was therefore natural to force every modification to a variable's values to go through a common set of functions, and to make these functions take a mandatory argument that can be either "no justification needed", "output a RUP statement for this", or "call the following piece of code to produce an explicit set of proof steps". Making this argument mandatory forces constraint authors to think explicitly about justifications, and avoids the potential for illicit modifications to be hiding in places where they can not easily be found by inspecting a proof log.

[2] https://www.minizinc.org/doc-2.5.5/en/modelling.html#an-arithmetic-optimisation-example
[3] https://abcnews.go.com/blogs/headlines/2012/06/can-you-solve-the-hardest-ever-sudoku

For backtracking search, we treat guessing on a branching variable to be a special kind of inference. Outputting the proof log then simply consists of tracing the search as backtracks are performed. Again, at no point was it necessary to manipulate pseudo-Boolean constraints or proof steps as anything other than simple strings created using a template.

5.3 Identifying Solver Bugs

Our experience has been that once the core solver is working and producing proofs for simple problems, it is somewhat more common to have bugs in the proof-producing code for new propagators than in the propagators themselves. Usually these bugs are extremely easy to fix, because VeriPB immediately flags the faulty line of the proof, and our solver can include a comment line immediately above any proof line saying exactly where in its source code that line originated. Similarly, propagator bugs are usually obvious from proof logs. For example, when we first implemented propagation for linear inequalities, we did not yet have a full proof logging setup for variables with large domains. We therefore used a VeriPB feature which allows for unchecked assertions to be included in the proof log (subject to an angry warning being issued at the end of the verification process) so that the remainder of the proofs could be verified. However, our implementation contained a bug, because one of the authors did not realise they did not understand the rules for rounding and integer division when both a variable and its coefficient are negative. Throughout conventional testing on the remainder of the solver, we never saw a single wrong answer being produced by this bug – but as soon as proof logging was implemented, we were told the exact line of code in our solver that was incorrect, even though correct sets of solutions were still being produced. Of course, one could claim that better testing would have identified this, but this relies upon the tester having intimate knowledge of how the propagator works and remembering that integer division of negative numbers could be a source of errors.

It can sometimes be harder to understand the problem when faulty proofs arise from insufficient justifications. For example, for the absolute value constraint, one of the authors had originally lazily assumed that its propagations would follow by a single RUP step in the same way as for not equals – and indeed this is often but not quite always the case. (This experience has left us extremely envious of the skills of authors of lazy clause generation solvers, who are able to write similar propagators without the benefit of machine verification.) This can lead to a proof verification error that only occurs several propagation steps later than the actual bug: the verifier always tells us if something is wrong, but does not always make it trivial to figure out where. However, because our solver forces all propagations to go through a central function call, it is easy to change the way proof logs are written so that all propagations are checked, including those which would usually be implicit.

5.4 Performance and Overheads

Having discussed the design and implementation of proof logging, we now talk about actually using it. This section answers two questions: "does proof logging work at all?", and "how expensive is proof logging in practice when used on large problems?".

To answer the second question, we must first establish whether our solver is "fast enough" that its results are likely representative of what would be seen if proof logging were introduced into a mature solver. For MiniCP, Michel et al. [24] include five benchmarks that are designed to test solver speed: they specify an exact search order, and propagation strength for global constraints, so that every solver is producing the same search tree for a fair speed comparison. Their aim was not to have the best model or search for a problem, but rather to benchmark

Table 1 Experimental results from our anonymous solver on six different problem instances. The first four problems are from the MiniCP benchmark suite and have a fixed model, search order, and propagation strength, to allow for a fair comparison between solvers. The final two problems are relatively simple, but use further global constraints that are not supported in MiniCP.

QAP: a quadratic assignment optimisation problem with linear inequalities, not equal constraints, a 2D element constraint, and large variables.						
Runtimes:	MiniCP:	16.9s	OscaR:	7.1s	Choco:	11.3s
	Anon:	5.6s	logging:	149.5s	VeriPB:	232,655.1s
Statistics:	propagators:	355	recursions:	125,805	inferences:	4,521,801
	OPB size:	6.4MBytes	log size:	19GBytes		
	RUP steps:	39,170,568	RPN steps:	413,295	red steps:	101,394
Magic Series: finding the only magic series of length 300, and proving it is unique. Uses linear equality and reified equality constraints.						
Runtimes:	MiniCP:	29.6s	OscaR:	8.8s	Choco:	29.8s
	Anon:	8.2s	logging:	425.2s	VeriPB:	est. 39 days
Statistics:	propagators:	90,301	recursions:	1,193	inferences:	15,584,073
	OPB size:	108MBytes	log size:	12GBytes		
	RUP steps:	7,923,342	RPN steps:	342,401	red steps:	358,800
Magic Square: finding the first 10,000 magic squares of size 5. Uses sum, not equal, and less than constraints.						
Runtimes:	MiniCP:	61.1s	OscaR:	32.3s	Choco:	32.9s
	Anon:	31.0s	logging:	1894.1s	VeriPB:	108,772.8s
Statistics:	propagators:	315	recursions:	6,042,079	inferences:	92,891,165
	OPB size:	145KBytes	log size:	100GBytes		
	RUP steps:	141,528,806	RPN steps:	70,946,952	red steps:	2,550
Queens: finding the first solution to the 88 queens problem. Uses not equals constraints.						
Runtimes:	MiniCP:	876.2s	OscaR:	477.8s	Choco:	438.8s
	Anon:	410.0s	logging:	3450.5s	VeriPB:	60,643.7s
Statistics:	propagators:	11,484	recursions:	49,339,390	inferences:	535,852,330
	OPB size:	8.9M	log size:	104GBytes		
	RUP steps:	50,130,687	RPN steps:	0	red steps:	31,152
Crystal Maze on the usual 8-vertex graph, all solutions. Uses GAC all-different, absolute value, and sum constraints.						
Runtimes:	Anon:	0.01s	logging:	0.13s	VeriPB:	6.3s
Statistics:	propagators:	35	recursions:	259	inferences:	8,737
	OPB size:	60K	log size:	2.6MBytes		
	RUP steps:	32,903	RPN steps:	6,685	red steps:	1,496
With autotabulation and GAC propagation on the sum constraints:						
Runtimes:	Anon:	0.01s	logging:	0.06s	VeriPB:	3.9s
Statistics:	propagators:	52	recursions:	139	inferences:	2,601
	OPB size:	60K	log size:	2.0MBytes		
	RUP steps:	29,467	RPN steps:	102	red steps:	2,958
Sudoku on Arto Inkala's "world's hardest Sudoku puzzle", all solutions. Uses GAC all-different and equals constraints.						
Runtimes:	Anon:	0.03s	logging:	0.05s	VeriPB:	0.52s
Statistics:	propagators:	48	recursions:	103	inferences:	1,388
	OPB size:	320K	log size:	484KBytes		
	RUP steps:	4,561	RPN steps:	460	red steps:	0

solvers performing the same well-defined task. We support enough global constraints (linear inequalities, sum, not equals, reified equals, a special element constraint, and less than) to implement four of these five benchmarks; we do not yet support the circuit global constraint for the fifth. In the first four rows of Table 1 we present computational results from a machine with dual Intel Xeon E5-2697A v4 CPUs, 512GBytes RAM, and a pair of solid state drives in a RAID 0 configuration, running Ubuntu 20.04.3 LTS, and benchmarking against the versions of the other solvers included in the supplementary material provided by Michel et al. In each case our solver is faster than the fastest of MiniCP, OscaR, and Choco, although sometimes only by a few percent. We therefore believe that the results that follow cannot be said to be unfairly optimistic due to the use of a slow solver.

Table 1 also shows runtimes for running our solver with proof logging enabled, together with statistics showing the size of the OPB models and VeriPB proof logs produced, and the number of RUP steps, groups of cutting planes steps (VeriPB works with sequences of cutting planes steps in reverse Polish notation, RPN, rather than one step per line), and redundance-based strengthening steps (red; two such steps are used to introduce an extension variable). On the four MiniCP benchmarks we see a slowdown of between 8.4 and 61.1 from proof logging. This should not be particularly surprising: without proof logging, our solver is making between eight hundred thousand and three million successful inferences per second, and the proof logs to justify these inferences range from ten to over a hundred GBytes in size. Furthermore, our implementation of proof logging is deliberately pessimal. We make use of C++ text output streams for file writing, which are notoriously inefficient. We write comments for most proof log lines generated, we make use of expressive variable names (which require several string concatenation operations and a hash table lookup for each variable written out), and proof lines are manipulated as strings for ease of implementation; all of these decisions are extremely helpful for exploratory research, but not for performance. Finally, these MiniCP benchmarks make use of only relatively simple and extremely fast propagators, which is where proof logging is most expensive. We therefore consider these performance results to be close to a worst case scenario, and would not be surprised if the overheads could be cut by at least a factor of five for some problems if implemented in a production solver that aimed purely for performance rather than for research and teaching.

Returning to the first question, we were able to verify the entire proofs for three of the four MiniCP problems; based upon the first ten percent of the proof for the remaining magic series problem, VeriPB estimates it will take 39 days to verify. We were able to verify entire proofs for smaller instances of the magic series problem. We have also produced and verified proofs for a range of other problems that make heavier use of global constraints – we show two of these in the bottom of Table 1, and other example problems and per-constraint tests are included in our supplementary material. Considering these results, and all the bugs that have been identified during development, we are comfortable in claiming that proof logging can be effective in practice. Although it may not (yet) scale practically to some of the more challenging combinatorial benchmark instances, it is already able to handle moderately sized problems involving several different global constraints, large variables, and reformulation.

6 Conclusion and Future Work

Proof logging gives us a way to trust outputs, not solvers. Trusting solvers seem to be a long way from being a practical reality for for constraint programming: even relatively simple propagators like all different have resisted attempts at formally verified implementation [8] even without the extensive optimisations used by modern solvers [11]. In contrast, we have shown that, with the right proof format, it is relatively easy to add proof logging to a wide variety of propagators, without requiring the proof verifier to understand anything about constraint programming – and this does not stand in the way of propagator optimisations such as greediness and incrementality.

There is still a lot of work to do before proof logging can be used by everyone all of the time. Firstly, there are many more global constraints and propagators to consider. Most of these will be straightforward, and will re-use existing strategies for proof logging in familiar ways. However, some will not be, and it is an open question as to whether cutting planes with extension variables give a sufficiently strong system to provide practical proof logging in every situation. We expect that recent work in proof logging for symmetry and dominance relations [2] might be necessary to justify certain propagators, as well as for reformulations involving symmetries, and would be interested in a deeper investigation into the relationship between constraint programming propagators and proof systems (with the caveat that "this system polynomially simulates natural deduction and so it can do everything" is not a helpful answer unless the polynomial is of very low order and with small constants).

Secondly, we must think about performance. Using formatted text output and string lookups to produce proof logs is useful for development and exploratory purposes, but for a production solver a better approach is probably needed. Verification performance is also a concern, although we have many reasons to be optimistic that this will improve. For example, very small changes to how proofs are written can give a huge improvement to verification times. We discussed two different ways of proof logging the not equals constraint, one of which involved justifying every propagation subject to the current guessed assignments, and the other which produced new clauses to assist unit propagation. On the MiniCP queens benchmark, using the former would have produced a 1.1TByte proof log that would take an estimated 138 days to verify, rather than a 100GByte proof that could be verified in under a day. If we are prepared to put slightly more cleverness into a solver, and abandon the gratuitous use of RUP steps in favour of a little more logic, we expect that proof sizes for some other constraints can be reduced by a similar factor.

An automated tool that performs proof minimisation would also be useful in this respect. Although potentially expensive to run, this would be very useful for auditability where proofs are to be stored, shared, and potentially verified more than once by different people. Such a tool could also provide annotations that would make RUP steps much quicker to verify – such an approach is already used for formally verified verifiers for DRAT, which actually verify a simplified format called LRAT [6].

On the other hand, relatively slow verification is not a fatal flaw. Proof logging is very good at catching solver bugs that will not be detected by conventional testing, even on relatively small instances. Because the same logic and code paths in a solver can be used whether or not proof logging is enabled, it is a useful feature to support even if it is not enabled all of the time. And, of course, many useful problems with serious real-world consequences derive most of their difficulty from the variety of constraints involved, rather than from being close to the limit of what we can solve computationally.

And thirdly, although we have a reasonably good solution for being confident in our translation from a constraint programming model into OPB, we have not discussed the further difficulty of verifying compilation from high level languages like Essence or MiniZinc. Perhaps it would be worth investigating techniques from formally-verified compilers to help with this translation.

References

1 Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 727–736. Springer, 2018. doi:10.1007/978-3-319-98334-9_46.

2 Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022*, 2022.

3 Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 7, pages 233–350. IOS Press, 2nd edition, February 2021.

4 Chiu Wo Choi, Warwick Harvey, J. H. M. Lee, and Peter J. Stuckey. Finite domain bounds consistency revisited. In *AI 2006: Advances in Artificial Intelligence, 19th Australian Joint Conference on Artificial Intelligence, Hobart, Australia, December 4-8, 2006, Proceedings*, pages 49–58, 2006. doi:10.1007/11941439_9.

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

6 Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2017. doi:10.1007/978-3-319-63046-5_14.

7 Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In Michel Rueher, editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 207–223. Springer, 2016. doi:10.1007/978-3-319-44953-1_14.

8 Catherine Dubois. Formally verified constraints solvers: a guided tour. CICM. Invited talk, 2020.

9 Leon Eifler and Ambros M. Gleixner. A computational status update for exact rational mixed integer programming. In Mohit Singh and David P. Williamson, editors, *Integer Programming and Combinatorial Optimization - 22nd International Conference, IPCO 2021, Atlanta, GA, USA, May 19-21, 2021, Proceedings*, volume 12707 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2021. doi:10.1007/978-3-030-73879-2_12.

10 Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 1486–1494. AAAI Press, 2020. URL: https://aaai.org/ojs/index.php/AAAI/article/view/5507.

11 Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artif. Intell.*, 172(18):1973–2000, 2008. doi:10.1016/j.artint.2008.10.006.

12 Xavier Gillard, Pierre Schaus, and Yves Deville. SolverCheck: Declarative testing of constraints. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, volume 11802 of *Lecture Notes in Computer Science*, pages 565–582. Springer, 2019. doi:10.1007/978-3-030-30048-7_33.

13 Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, 2020. doi:10.1007/978-3-030-58475-7_20.

14 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 1134–1140. ijcai.org, 2020. doi:10.24963/ijcai.2020/158.

15 Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3768–3777. AAAI Press, 2021. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16494.

16 Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe Conference (DATE)*, pages 10886–10891. IEEE Computer Society, 2003.

17 Warwick Harvey and Joachim Schimpf. Bounds consistency techniques for long linear constraints. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems*, pages 39–46, 2002.

18 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 181–188. IEEE, 2013. URL: http://ieeexplore.ieee.org/document/6679408/.

19 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2013. doi:10.1007/978-3-642-38574-2_24.

20 Linnea Ingmar and Christian Schulte. Making compact-table compact. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 210–218. Springer, 2018. doi:10.1007/978-3-319-98334-9_14.

21 Evelyn Lamb. Two-hundred-terabyte maths proof is largest ever. *Nature*, 545:17–18, 2016.

22 Christophe Lecoutre. STR2: optimized simple tabular reduction for table constraints. *Constraints An Int. J.*, 16(4):341–371, 2011. doi:10.1007/s10601-011-9107-6.

23 Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Comput. Sci. Rev.*, 5(2):119–161, 2011. doi:10.1016/j.cosrev.2010.09.009.

24 Laurent D. Michel, Pierre Schaus, and Pascal Van Hentenryck. MiniCP: a lightweight solver for constraint programming. *Math. Program. Comput.*, 13(1):133–184, 2021. doi:10.1007/s12532-020-00190-7.

25 Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 544–558. Springer, 2007. doi:10.1007/978-3-540-74970-7_39.

26 Adrian Rebola-Pardo and Luís Cruz-Filipe. Complete and efficient DRAT proof checking. In Nikolaj Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018. doi:10.23919/FMCAD.2018.8602993.

27 Olivier Roussel and Vasco M. Manquinho. Input/output format and solver requirements for the competitions of pseudo-Boolean solvers. Revision 2324. Available at `http://www.cril.univ-artois.fr/PB16/format.pdf`, January 2016.

28 Peter J. Stuckey. Certifying optimality in constraint programming, February 2019. Talk at KTH Royal Institute of Technology.

29 Julian R. Ullmann. Partition search for non-binary constraint satisfaction. *Inf. Sci.*, 177(18):3639–3678, 2007. `doi:10.1016/j.ins.2007.03.030`.

30 Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *10th International Symposium on Artificial Intelligence and Mathematics (ISAIM)*, 2008. http://isaim2008.unl.edu/index.php?page=proceedings.

31 Michael Veksler and Ofer Strichman. A proof-producing CSP solver. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1754`.

32 Hélène Verhaeghe. *The extensional constraint*. PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2021. URL: `http://hdl.handle.net/2078.1/252859`.

33 Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014. `doi:10.1007/978-3-319-09284-3_31`.

From Cliques to Colorings and Back Again

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

Anthony Karahalios
Carnegie Mellon University, Pittsburgh, PA, USA

Willem-Jan van Hoeve
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

We present an exact algorithm for graph coloring and maximum clique problems based on SAT technology. It relies on four sub-algorithms that alternatingly compute cliques of larger size and colorings with fewer colors. We show how these techniques can mutually help each other: larger cliques facilitate finding smaller colorings, which in turn can boost finding larger cliques. We evaluate our approach on the DIMACS graph coloring suite. For finding maximum cliques, we show that our algorithm can improve the state-of-the-art MaxSAT-based solver IncMaxCLQ, and for the graph coloring problem, we close two open instances, decrease two upper bounds, and increase one lower bound.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Mathematics of computing → Graph coloring

Keywords and phrases Graph coloring, maximum clique, Boolean satisfiability

Digital Object Identifier 10.4230/LIPIcs.CP.2022.26

Supplementary Material *Software (Source Code and Log Files of Experiments)*: `https://github.com/marijnheule/clicolcom`
archived at `swh:1:dir:2e1f585e920ed0805002916e31263acdd04746ea`

Funding *Marijn J. H. Heule*: Partially supported by NSF under grant CCF-2006363.
Anthony Karahalios: Partially supported by Office of Naval Research Grant No. N00014-21-1-2240.
Willem-Jan van Hoeve: Partially supported by Office of Naval Research Grant No. N00014-21-1-2240 and National Science Foundation Award #1918102.

1 Introduction

Given a graph, the *vertex coloring problem* asks to label each vertex of the graph with a color such that adjacent vertices have different labels, using the minimum number of colors (the *coloring number*). A closely related problem is the *maximum clique problem*, which asks to find a subset of vertices that are pairwise adjacent, of maximum size (the *clique number*). Both are NP-hard combinatorial optimization problems at the heart of practical applications including scheduling, timetabling, and network analysis [11, 36].

Many different algorithms have been proposed to solve vertex coloring and maximum clique problems in practice. One stream of research focuses on dedicated exact and heuristic algorithms (e.g., Cliquer [19] and DSATUR [1]), while another stream uses generic methodologies, such as integer programming and column generation (e.g., [7, 17, 18]), constraint programming (e.g., [4, 21]), or Boolean satisfiability (e.g., [6, 13, 32]). An important milestone for these developments was the second DIMACS challenge on cliques, coloring, and satisfiability that was launched in 1993 [11]. To our knowledge, for the DIMACS graph coloring challenge, several instances remain unsolved and in the past eight years only a few instances were closed: wap01a in 2021 [26], 5-FullIns_4 in 2021 [31], and 4-FullIns_5 in 2014 [14, 38]. Our method solves these instances as well (and quickly). Similarly, only a few improved bounds have been found that do not close instances: C2000.9 in 2021 [33] and DSJC250.1 in 2020 [20]. Before this, many instances were closed around 2012 [4, 8, 15, 27–29, 35] and earlier.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 26; pp. 26:1–26:10
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

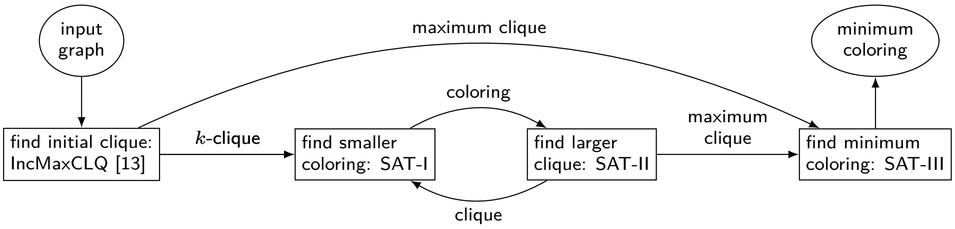

input graph
maximum clique
minimum coloring
coloring
maximum clique
find initial clique: IncMaxCLQ [13]
k-clique
find smaller coloring: SAT-I
find larger clique: SAT-II
find minimum coloring: SAT-III
clique

Figure 1 Illustration of the CliColCom algorithm to find a maximum clique and a minimum vertex coloring. The solver IncMaxCLQ is based on MaxSAT. The SAT-I encoding uses a given clique to quickly find colorings. The SAT-II encoding uses these colorings to find a larger clique. Once the maximum clique is found, encoding SAT-III is used to find the minimum coloring.

In this work, we first revisit the performance of Boolean satisfiability (SAT) solvers on graph coloring and maximum clique problems. The best known maximum clique solver called IncMaxCLQ [13] is based on MaxSAT technology, which is able to close all but four instances of the DIMACS Clique benchmark suite and find maximum cliques for all but eight instances of the DIMACS Coloring benchmark suite. For graph coloring, the best known solver is the branch-and-bound hybrid CP/SAT solver gc-cdcl [6]. We show that a direct encoding coupled with either the local search SAT solver DDFW [9] or CaDiCaL [3] provides surprisingly strong results. For the 69 DIMACS coloring instances where the coloring number equals the clique number, combining IncMaxCLQ and one of DDFW or CaDiCaL solves 54 instances in under ten minutes.

We therefore concentrate on two cases – finding stronger colorings for instances where we quickly have a maximum clique, and improving both cliques and colorings for instances where we do not quickly find a maximum clique, which are often those where the coloring number does not equal the clique number. We propose an algorithm, named *CliColCom*, (derived from "cliques, colorings, and communication") that consists of four sub-algorithms with an inner loop that alternates between finding cliques of larger size and colorings with fewer colors (see Fig. 1). Specifically, we use cliques to define a symmetry-breaking predicate based on a variable ordering for the coloring problem (using encoding SAT-I), similar to ones used by Van Gelder [32] and Velev [34]. Conversely, we use colorings to formulate the maximum clique problem (using encoding SAT-II), similar to the MaxSAT encoding by Li [13]. We continue this alternating process until a maximum clique is found, which serves as input to the final sub-algorithm that finds a minimum coloring (using encoding SAT-III). This approach can be viewed as a new form of communication between SAT solvers. While one way that SAT solvers communicate is through exchanging learned clauses like in portfolio-based parallel SAT [5], we demonstrate how SAT solvers can also pass solutions back and forth, using the other solver's solution in its problem's clauses.

We show that CliColCom can find larger cliques than IncMaxCLQ for two of the eight DIMACS Coloring instances that IncMaxCLQ cannot solve, and for the vertex coloring problem closes two open instances (wap02a, wap08a), improves one lower bound (r1000.1c), and improves two upper bounds (wap03a, wap04a).

The rest of this paper is organized as follows. In Section 2 we provide formal definitions and notation for graph coloring and maximum clique problems. Section 3 presents the details of our algorithm. In Section 4 we provide an overview of the used tools. The experimental evaluation is presented in Section 5, and we conclude in Section 6.

2 Graph Coloring and Maximum Clique Problems

We first recall the definitions of cliques and colorings [22]. Let $G = (V, E)$ be an undirected graph with vertex set V and edge set E. A *k-clique* is a subset of k vertices that are pairwise adjacent. A *maximum clique* of G is a clique in G of maximum size. The size of a maximum clique is called the *clique number* of G.

An *independent set* is a subset of vertices that are pairwise non-adjacent. A *k-coloring* of G is a partition of V into k independent sets $V_1, V_2, \dots, V_k$. The independent sets represent the *color classes* of the coloring. The *coloring number* of G is the size of a coloring that uses the minimum number of colors.

The existence of a k-clique proves a lower bound of k on the clique number, and a k-coloring proves an upper bound of k on the coloring number. To prove the dual bounds, one must show that a $k+1$-clique and $k-1$-coloring do not exist. The existence of a k-clique proves a lower bound of k for the coloring number. Both the vertex coloring and max clique problems are NP-hard, so computational results of algorithms are of interest [12, 16, 36].

3 CliColCom Algorithm

In this section we present an exact algorithm for graph coloring that also contains an exact algorithm for the maximum clique problem. It consists of four sub-algorithms as in Fig. 1. The algorithm for the maximum clique problem is obtained by omitting sub-problem SAT-III. We next describe each of the sub-algorithms below.

3.1 IncMaxCLQ: Find an Initial Clique

The input to the first sub-algorithm is a graph $G = (V, E)$. To obtain an initial clique, we run an exact MaxSAT solver called IncMaxCLQ [13] with a time limit; we use one second in our experiments. If IncMaxCLQ finds a maximum clique and proves optimality, then we immediately go to step SAT-III using this maximum clique. Otherwise the clique returned by IncMaxCLQ will be used for the SAT-I encoding.

3.2 SAT-I: Find a Coloring

The next sub-algorithm called SAT-I takes as input a graph $G = (V, E)$, a k-clique C, and an upper bound $b \geq k$. Its purpose is to find a coloring of good quality. When we first enter this sub-algorithm, we determine b by running the DSATUR graph coloring heuristic. In subsequent calls, b will be the best known coloring number.

The SAT-I encoding is optimized for local search solvers and asks for the existence of a b-coloring of G. It has two sets of constraints: 1) each vertex has at least one color; and 2) adjacent vertices are colored differently. The direct encoding uses color variables $x_{v,i}$ which denote that vertex $v \in V$ has color $i \in \{1, \dots, b\}$. The first constraint consists simply of a single clause of b literals per vertex:

$$(x_{v,1} \vee \cdots \vee x_{v,b}) \text{ for } v \in V.$$

Note that this only enforces that there is at least one color per vertex instead of exactly one color per vertex. The latter would include clauses of the form $(\overline{x}_{v,i} \vee \overline{x}_{v,j})$ for $1 \leq i < j \leq b$. These clauses however are known to be "blocked" and top-tier SAT solvers eliminate them [10].

The second constraint uses the following clauses:

$$(\overline{x}_{u,i} \vee \overline{x}_{v,i}) \text{ for } (u, v) \in E, i \in \{1, \dots, b\}.$$

To break the color symmetry, we add unit clauses that assign a different color to each vertex in the given clique C, which is a common practice [32].

The encoding is used as follows. We start with bound $b-1$ and run a local search solver for a limited time (or number of flips). If no coloring is found within the limit, we report the previously found b-coloring. Otherwise, we decrease b by one unit and repeat. We thus return the best coloring we can find within a limited time. Note that if the encoding for $b = |C|$ is satisfiable, then we have found the coloring number of G.

3.3 SAT-II: Find a Larger Clique

The third sub-algorithm uses the fact that a vertex coloring is a partition of the graph into independent sets. The coloring from SAT-I is used to define these independent sets. For a graph $G = (V, E)$ and a p-partition $\{V_1, \dots, V_p\}$ of V into independent sets, the encoding SAT-II asks whether there exists a clique of size c, where $c \le p$.

This encoding uses clique variables $v_{i,s}$, which denote that the s-th vertex in V_i is part of the clique. Apart from the clique variables, the encoding uses relaxation variables r_i denoting that no vertex from partition V_i is used in the clique. The clauses have the following form:

$$(r_i \lor v_{i,1} \lor \cdots \lor v_{i,|V_i|}) \text{ for } i \in \{1, \dots, p\}.$$

Additionally we have constraints between partitions enforcing that two vertices from different partitions cannot be in a clique if there is no edge between them in the graph:

$$(\overline{v}_{i,s} \lor \overline{v}_{j,t}) \text{ for } 1 \le i < j \le p, s \in \{1 \dots, |V_i|\}, t \in \{1, \dots, |V_j|\}, (v_{i,s}, v_{j,t}) \notin E.$$

We could have included similar clauses for pairs of vertices within a partition. However, these clauses are blocked as well and top-tier solvers would remove them.

Finally, we have a constraint stating that at most $k = p - c$ of the relaxation variables can be assigned to true. We use the sequential counter encoding proposed by Sinz to enforce the "at most $p - c$" constraint [25]. This encoding introduces $O(pk)$ auxiliary variables and $O(pk)$ additional clauses.

The sub-algorithm starts with $c = |C| + 1$ where C is the largest clique found so far by either IncMaxCLQ or SAT-II itself. We solve the encoding with an exact CDCL solver (see Section 4). If the formula is unsatisfiable, then the largest clique has size $c-1$. Otherwise, we have found a clique C' of size c and continue by increasing the bound $c \mathrel{+}= 1$ and repeating this sub-algorithm. If the SAT-II encoding cannot be solved within a certain time limit, we return to sub-algorithm SAT-I to find a smaller vertex coloring, using the improved clique C'. Due the time limits imposed on SAT-I and SAT-II, we could in theory repeatedly solve them with the same clique and the same coloring. For that reason, we increase the time limit for SAT-II with a multiplicative factor when the coloring and the clique have not changed, so that sub-algorithm SAT-II becomes exact. Therefore, SAT-II will eventually return a maximum clique, unless a global time limit on the overall algorithm is exceeded.

3.4 SAT-III: Find an Optimal Coloring

The final sub-algorithm uses encoding SAT-III, which generalizes SAT-I and is optimized for CDCL solvers. The sub-algorithm takes as input a graph $G = (V, E)$, a k-clique C, and a lower bound b. The first part of the encoding is identical to the SAT-I encoding with G, C, and b as input.

We additionally add clauses to break color symmetries. To this end, we first construct a vertex ordering O, by starting with the vertices in C in arbitrary order. We then iteratively extend the ordering by adding the vertex with the most neighbors in O, breaking ties by highest degree. We break the color symmetries for the vertices from $k+1$ to $|O|$ in the ordering. Let v_i denote the i-th vertex in ordering O. The encoding enforces that if none of the first $i-1$ vertices in O uses the color c, then vertex v_i must have a color less or equal to c. The clauses have the following form:

$$(x_{v_1,c} \lor x_{v_2,c} \lor \cdots \lor x_{v_{i-1},c} \lor \overline{x}_{v_i,d}) \text{ for } d \in \{c+1,\ldots,b\}, i \in \{k+1,\ldots,|O|\}.$$

The sub-algorithm uses this encoding as follows: starting with $b = k$, we solve the formula using an exact CDCL solver. If the formula is found to be unsatisfiable, meaning that G requires more than b colors, the bound is increased $b \mathrel{+}= 1$ and we repeat. This is continued until the formula is satisfiable. The final bound equals the coloring number of G.

3.5 Example Run

We illustrate the flow between the sub-algorithms using graph r1000.1c. This instance has clique number 92, while the best known lower and upper bound on the coloring number are 96 and 98, respectively [4]. We first run IncMaxCLQ, which returns a clique of size 82 within one second (which it cannot improve within reasonable time).

We next run the SAT-I encoding with bound $b = 110$ (from the coloring found by DSATUR). The local search solver UBCSAT with the WalkSAT algorithm lowers the upper bound one by one until it reaches $b = 102$ and times out (i.e., reaching a million flips without finding a coloring). Each step takes a few seconds. The 103-coloring is used in the SAT-II encoding. We start with $c = 83$ (the size of the clique + 1). The solver CaDiCaL finds a satisfying assignment in a fraction of a second. This also holds for the bounds $c \in \{84,\ldots,92\}$. The bound $c = 93$ times out (reaches a million conflicts).

We return to SAT-I using the 92-clique. This helps the local search solver and now it can find a coloring for $b = 102$ and can even lower it to 98 before timing out on $b = 97$. The 98-coloring is used in SAT-II. This time CaDiCaL can prove optimality of $c = 92$ (the $c = 93$ instance is unsatisfiable). Now that the maximum clique has been determined, we switch to SAT-III to determine the coloring number. The clique of size 92 is extended to a vertex ordering. The solver CaDiCaL is used to solve the instances with bounds $b \in \{92,\ldots,97\}$. The bounds up to $b = 96$ are unsatisfiable, while $b = 97$ times out (24 hours). Therefore, we report an improved lower bound of 97 on the coloring number for this open instance.

4 SAT Solving Paradigms

The best SAT-solving paradigm differs for each of the encodings proposed in the prior section. Because some sub-algorithms work by solving a sequence of SAT instances, the use of MaxSAT solvers could be also be explored. Below we will discuss the ones used during our experiments.

Conflict-Driven Clause Learning. The most effective and well-known exact SAT-solving paradigm is conflict-driven clause learning (CDCL) [24]. In the context of maximum clique and graph coloring, CDCL is mostly effective for unsatisfiability results. In particular, we use this paradigm to increase the lower bound results after the maximum clique was determined. Although the default heuristics in CDCL solvers are in general effective on a broad range of formulas, we observed that using negative branching instead of phase saving improves performance on graph coloring instances. We will use the CDCL solver CaDiCaL during the experiments and turn on negative branching (options `-forcephase=1 -phase=0`).

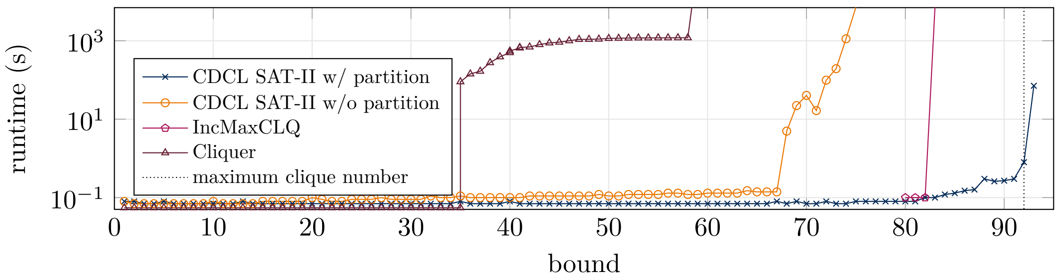


Figure 2 Performance of different maximum clique techniques to compute a large clique of r1000.1c. The method SAT-II uses the partition obtained from a 98-coloring obtained by SAT-I.

Local Search. An almost obscure, yet quite effective local search solving paradigm is called Divide and Distribute Fixed Weights (DDFW) [9]. In DDFW, all clauses have weights. The algorithm flips variable assignments if the weighted sum of the satisfied clauses improves. If no such variable exists (i.e., a local minimum is reached), then a random falsified clause is selected which is increased in weight by pulling weight of its neighboring satisfiable clauses. This is repeated until a satisfying assignment is found. We use the implementation of DDFW in UBCSAT [30] for the experiments with SAT-III.

A well-known local search algorithm is WalkSAT [23]. Given a random assignment, the algorithm picks a random falsified clause and flips one of its literals to satisfy the clause. This is repeated until a satisfying assignment is found. WalkSAT is much more greedy compared to DDFW. This is helpful to reduce upper bounds in SAT-I. However, it is not effective to find a coloring for graphs when the coloring number equals the clique number. We use the implementation of WalkSAT in UBCSAT.

5 Experiments

We tested the performance of our method on solving both the maximum clique and vertex coloring problems on the 137 DIMACS Graph Coloring instances. This benchmark consists of a variety of instances with different sizes and densities – some random graphs and some from real world problems. We chose this sets of instances even for maximum clique performance because the DIMACS Maximum Clique and BHOSLIB [37] instances are almost all solved. The source code and log files of the experiments are available in the repository `https://github.com/marijnheule/clicolcom`. We will note when we run experiments on one of two different CPUs: Intel Xeon 2.33GHz CPU or AMD EPYC 7742 CPU [2].

5.1 Maximum Clique Results

As a baseline, we first ran IncMaxCLQ which solved 129 instances to optimality within one hour on the Intel Xeon 2.33GHz CPU. IncMaxCLQ failed to produce and prove a maximum clique for only eight graphs: C2000.5, C2000.9, C4000.5, latin_square_10, DSJC500.9, DSJC1000.9, DSJR500.1c, and r1000.1c. For the last two instances, the largest found cliques were of size 78 and 82, respectively. Our method is able to compute the maximum clique of them in a few minutes: 83 and 92, respectively. We are not aware of any other tool that can compute (and prove optimality) of the maximum cliques for these two instances.

Figure 2 illustrates the effectiveness of the SAT-II encoding. It shows for the open instance r1000.1c the runtimes of various techniques to find a large clique. We only find the maximum clique of size 92 with the SAT-II encoding that uses a 98-coloring obtained via SAT-I using

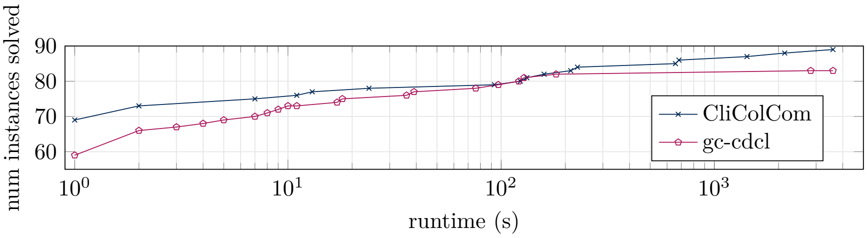


Figure 3 Performance profile of the number of DIMACS graph coloring instances gc-cdcl and CliColCom can prove to optimality over time.

local search. The instance with $b = 93$ is unsatisfiable, hence the larger runtime. IncMaxCLQ can compute a clique of size 82, while Cliquer gets only to 58. Without a coloring (i.e., each vertex is an independent set), SAT-II performance poorly and finds cliques up until size 74.

5.2 Comparison with State-of-the-Art Graph Coloring

We run these experiments on an Intel Xeon 2.33GHz CPU. We use gc-cdcl as a baseline because it is the SAT-based solver that performs best on this problem domain.[1]

We ran gc-cdcl for 1 hour and it proved the optimal solution for 83 instances. We compare this to our method in Fig. 3, which shows that we can solve 88 instances in 1 hour. The differences are as follows: gc-cdcl solves myciel7, queens9_9, and qg.order60 and CliColCom does not. CliColCom solves 4-FullIns_5, 1-Insertions_4, DSJR500.1c, le450_15c, le450_15d, wap01a, wap02a, wap06a and gc-cdcl does not.

We observed strong performance of our setup on the wap0* graphs. We therefore ran each instance on a cluster of AMD EPYC 7742 CPUs with 128 seeds. The results are reported in Table 1. We improve the upper bound on four graphs, which includes closing two instances. The DDFW algorithm was crucial to obtain these results. The wap01 instance was recently closed with a method that requires significantly more time [26].

Table 1 DDFW runtimes in seconds for wap0* instances using 128 seeds (no timeout). The second and third column show the lower and upper bounds. The bold bounds are improvements.

instance	LB	UB	min	mean	max
wap01a	41	41	291.19	736.01	1855.56
wap02a	40	**40**	195.45	382.85	883.02
wap03a	40	**43**	9612.49	15865.50	21963.13
wap04a	40	**41**	29757.11	65609.40	91501.84
wap05a	50	50	1.37	1.59	2.11
wap06a	40	40	9.21	26.44	92.54
wap07a	40	41	211.26	632.63	2207.33
wap08a	40	**40**	1016.65	6742.98	12096.61

[1] The paper “An Incremental SAT-Based Approach to the Graph Colouring Problem” published in CP 2019 claims strong computational results. Although these are reported in an aggregated form, they imply that several challenging open instances would have been solved. The GitHub repository linked in the paper was deleted. We contacted the authors, who were unfortunately unable to share the code or reproduce the published results. We therefore omit a comparison with that work.

5.3 Robustness, Variations, and Discussion

Naturally, our algorithm is sensitive to variations in its design. Below we discuss some extensions and variants to provide additional insights.

Robustness. Replacing the CDCL solver by any modern CDCL solver would hardly change runtime. The use of negative branching in CDCL slightly improves performance and is available in most SAT solvers. Increasing the timeout has little to no impact.

For the improved upper bounds of the wap graphs, we tried many local search solvers and only the implementation of DDFW in UBCSAT seems to be able to obtain them. The key aspects that impact the performance are: 1) fix only the clique for local search (full symmetry breaking significantly hurts local search solvers); 2) use full symmetry breaking for CDCL (otherwise unsatisfiable instances become impossible to solve); and 3) use the communication (otherwise hard problems cannot be solved).

Multiple colorings for SAT-II. The presented SAT-II encoding for finding a larger clique uses one vertex coloring in its clauses. This encoding can be extended to use multiple colorings by introducing corresponding sets of literals and clauses for each coloring. Taking DSJC250.9 as an example, we show that using multiple colorings can be beneficial to the runtime. We ran experiments using CaDiCaL for SAT-II that used either 1, 2, or 5 74-colorings to solve for a 43-clique. Using 40 trials for each number of colorings, the mean runtimes were 450, 62, and 202 respectively. This indicates that using two colorings may improve runtimes compared to one coloring, but using five colorings can perform worse than two colorings.

Vertex ordering for SAT-III. Encoding SAT-III for finding a minimum coloring uses a vertex ordering that begins with a maximum clique, assuming that starting with a maximum clique is effective. Although useful in most cases, this heuristic does not always result in the most effective ordering. For example, consider the graph coloring instance `queen9_9`, i.e., the n-queens instance of size 9. Its largest clique has size 9 while its coloring number is 10. Finding a clique of size 9 and a coloring of size 10 is easy. However, showing the absence of a 9-coloring (unsatisfiable, thus CaDiCaL was used) is hard: solving the SAT-III encoding starting with a border 9-clique (e.g., the top row) requires roughly 3 hours solving. Starting with a diagonal 9-clique reduces the runtime to 400 seconds. But, if one starts with a *non-optimal* 5-clique in the center (the + shaped clique that cannot be extended to a 6-clique), the runtime reduces to 100 seconds.

Heavy cliques for SAT-III. IncMaxCLQ is very effective in finding a maximum clique. However, we observed that when using a clique to generate a vertex ordering for SAT-III, performance of SAT-III may be enhanced by using the *heaviest* maximum clique, i.e., the maximum clique with the maximum sum of the vertex degrees. For example, for the queen instances the heaviest maximum clique is a diagonal, which we find to generate an ordering leading to better runtime when solving SAT-III compared to using a border row or column. Enhancing IncMaxCLQ to produce such a clique would further strengthen the results.

6 Conclusion

We were able to achieve state-of-the-art performance on the well-known DIMACS Coloring benchmark suite by combining off-the-shelf (Max)SAT-solving tools and a combination of three SAT encodings. Our algorithm, called CliColCom, uses the encodings to alternate between finding larger cliques and smaller colorings until a maximum clique and minimum coloring is found. We closed two open instances of the DIMACS benchmark suite and improved bounds on three others.

References

1 D. Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.

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

3 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.

4 Stefano Gualandi and Federico Malucelli. Exact solution of graph coloring problems via constraint programming and column generation. *INFORMS Journal on Computing*, 24(1):81–100, 2012.

5 Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2010.

6 Emmanuel Hébrard and George Katsirelos. Constraint and satisfiability reasoning for graph coloring. *Journal of Artificial Intelligence Research*, 69:33–65, 2020.

7 S. Held, W. Cook, and E. C. Sewell. Maximum-weight stable sets and safe lower bounds for graph coloring. *Mathematical Programming Computation*, 4(4):363–381, 2012.

8 Stephan Held, William Cook, and Edward C Sewell. Safe lower bounds for graph coloring. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 261–273. Springer, 2011.

9 Abdelraouf Ishtaiwi, John Thornton, Abdul Sattar, and Duc Nghia Pham. Neighbourhood clause weight redistribution in local search for sat. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, pages 772–776, Berlin, Heidelberg, 2005. Springer.

10 Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010.

11 David S Johnson and Michael A Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc., 1996.

12 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

13 Chu-Min Li, Zhiwen Fang, and Ke Xu. Combining maxsat reasoning and incremental upper bound for the maximum clique problem. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 939–946. IEEE, 2013.

14 Shadi Mahmoudi and Shahriar Lotfi. Modified cuckoo optimization algorithm (mcoa) to solve graph coloring problem. *Applied soft computing*, 33:48–64, 2015.

15 Enrico Malaguti, Michele Monaci, and Paolo Toth. An exact approach for the vertex coloring problem. *Discrete Optimization*, 8(2):174–190, 2011.

16 Enrico Malaguti and Paolo Toth. A survey on vertex coloring problems. *International transactions in operational research*, 17(1):1–34, 2010.

17 A. Mehrotra and M. A. Trick. A Column Generation Approach for Graph Coloring. *INFORMS Journal on Computing*, 8(4):344–354, 1996.

18 I. Méndez-Díaz and P. Zabala. A Branch-and-Cut algorithm for graph coloring. *Discrete Applied Mathematics*, 154:826–847, 2006.

19 Patric RJ Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1-3):197–207, 2002.

20 Daniel Porumbel. Projective cutting-planes. *SIAM Journal on Optimization*, 30(1):1007–1032, 2020.

21 Jean-Charles Régin. Using Constraint Programming to Solve the Maximum Clique Problem. In *Proceedings of CP*, pages 634–648, 2003.

22 Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency.* Springer, 2003.

23 Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In Barbara Hayes-Roth and Richard E. Korf, editors, *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1*, pages 337–343. AAAI Press / The MIT Press, 1994.

24 João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

25 Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *International conference on principles and practice of constraint programming*, pages 827–831. Springer, 2005.

26 Wen Sun, Jin-Kao Hao, Yuhao Zang, and Xiangjing Lai. A solution-driven multilevel approach for graph coloring. *Applied Soft Computing*, 104:107174, 2021.

27 Olawale Titiloye and Alan Crispin. Graph coloring with a distributed hybrid quantum annealing algorithm. In *KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*, pages 553–562. Springer, 2011.

28 Olawale Titiloye and Alan Crispin. Quantum annealing of the graph coloring problem. *Discrete Optimization*, 8(2):376–384, 2011.

29 Olawale Titiloye and Alan Crispin. Parameter tuning patterns for random graph coloring with quantum annealing. *PloS one*, 7(11):e50060, 2012.

30 Dave A. D. Tompkins and Holger H. Hoos. UBCSAT: an implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2004.

31 R. P. van der Hulst. A branch-price-and-cut algorithm for graph coloring. Master's thesis, University of Twente, 2021.

32 Allen Van Gelder. Another look at graph coloring via propositional satisfiability. *Discrete Applied Mathematics*, 156(2):230–243, 2008.

33 Willem-Jan van Hoeve. Graph coloring with decision diagrams. *Mathematical Programming*, pages 1–44, 2021.

34 Miroslav N Velev. Exploiting hierarchy and structure to efficiently solve graph coloring as sat. In *2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 135–142. IEEE, 2007.

35 Qinghua Wu and Jin-Kao Hao. Coloring large graphs based on independent set extraction. *Computers & Operations Research*, 39(2):283–290, 2012.

36 Qinghua Wu and Jin-Kao Hao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693–709, 2015.

37 Ke Xu. Bhoslib: Benchmarks with hidden optimum solutions for graph problems, 2004.

38 Zhaoyang Zhou, Chu-Min Li, Chong Huang, and Ruchu Xu. An exact algorithm with learning for the graph coloring problem. *Computers & operations research*, 51:282–301, 2014.

On the Enumeration of Frequent High Utility Itemsets: A Symbolic AI Approach

Amel Hidouri [1]
CRIL – CNRS UMR 8188, University of Artois, France
LARODEC, University of Tunis, Tunisia

Said Jabbour
CRIL – CNRS UMR 8188, University of Artois, France

Badran Raddaoui
SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, France

Abstract

Mining interesting patterns from data is a core part of the data mining world. High utility mining, an active research topic in data mining, aims to discover valuable itemsets with high profit (e.g., cost, risk). However, the measure of interest of an itemset must primarily reflect not only the importance of items in terms of profit, but also their occurrence in data in order to make more crucial decisions. Some proposals are then introduced to deal with the problem of computing high utility itemsets that meet a minimum support threshold. However, in these existing proposals, all transactions in which the itemset appears are taken into account, including those in which the itemset has a low profit. So, no additional information about the overall utility of the itemset is taken into account. This paper addresses this issue by introducing a SAT-based model to efficiently find the set of all frequent high utility itemsets with the use of a minimum utility threshold applied to each transaction in which the itemset appears. More specifically, we reduce the problem of mining frequent high utility itemsets to the one of enumerating the models of a formula in propositional logic, and then we use state-of-the-art SAT solvers to solve it. Afterwards, to make our approach more efficient, we provide a decomposition technique that is particularly suitable for deriving smaller and independent sub-problems easy to resolve. Finally, an extensive experimental evaluation on various popular datasets shows that our method is fast and scale well compared to the state-of-the art algorithms.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence; Information systems → Data mining

Keywords and phrases Data Mining, High Utility Itemsets, Propositional Satisfiability

Digital Object Identifier 10.4230/LIPIcs.CP.2022.27

1 Introduction

The broad topic of data mining research aims to discover a set of relevant patterns that together represent the properties of the data. The successful use of data mining in e-commerce and e-marketing became a core practice in the retail industry. Pattern discovery, one of the most important sub-fields of data mining, involves computing interesting patterns in databases. Retailers are using such patterns to find customer habits in order to provide better services and increase sales. Traditional itemset mining models can be characterized into two lines of work. The first one, known as *Frequent Itemsets Mining* (in short, FIM), is based on the popular metric of support (i.e., the number of transactions involving the itemset) to determining how interesting a motif is. Specifically, FIM seeks to identify patterns whose frequency exceeds a predefined threshold. Nevertheless, frequency alone is often considered as a poor measure of interestingness [29]. In fact, frequent itemsets have a significant bottleneck

[1] Corresponding author

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 27; pp. 27:1–27:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in that they only reflect the occurrence of items in a database and miss their importance, e.g., items that can be rare but generate more profit. Typically, the significance of an item in each transaction can be different. The second line of research consists of *High Utility Itemset Mining* (HUIM, for short), which is an extension of FIM. Basically, HUIM is a research area designed to address the shortcomings of frequency-based algorithms by taking in addition to the item frequency the significance/interestingness of items into account, such as the price, the quantity, etc., when mining patterns. A high utility itemset is then an itemset whose utility value is greater than a user-specified threshold. Generally speaking, the utility can be quantified in terms of cost, risk, profit or any other user preference relations among items. The HUIM task has emerged as a key data mining primitive in many practical applications, including market analysis, customer trend analysis and financial analysis [13]. The two aforementioned lines of research, however, are generally considered separately. To be precise, the two frameworks were designed with different goals in mind, either for mining the set of items that occur frequently while ignoring their profit, or for computing the set of items that yield the highest gain values as a sole criterion while avoiding the frequency measure. In the last case, non-frequent itemsets could be considered as high utility itemsets. In recent years, with ever-advancing technology, one may be more interested in finding at the same time frequently purchased items with high gain values. In fact, computing such itemsets can help sales managers understand customer research behavior and what she/he needs, as well as provide appropriate product combinations. For example, a retail store manager can use this knowledge to make decisions to keep products neighboring or also to promote products. These itemsets can also be used to assess the risk of selling a product by removing for instance very seldom interesting products from the market.

In the literature, a number of frequent utility-driven mining methods have been studied, each with its own advantages. Specifically, Wei et al. [30] proposed a novel algorithm, called `FCHUIM`, that combines the frequency and utility constraints to find frequent high utility itemsets. This approach considers a candidate itemset as frequent if and only if it is covered by a minimum number of transactions and his utility in these transactions is greater than a user-specified minimum utility value. `FCHUIM` is based on a closed high utility representation and it employs a nested list to eliminate non-frequent itemsets. Furthermore, `HU-FIMi` [28] is another single phase algorithm to compute efficiently high utility frequent itemsets in transactional databases. It exploits different orderings of items and introduces two new pruning measures (cutoff and suffix utility) in order to reduce the search space exploring cost. Another specialized form of frequent utility-based data mining field is to extract only the itemsets with the highest utility values (i.e., above a fixed threshold value). As a result, transactions involving the itemset with the lowest utility value are not considered as covers for this itemset because no valuable information is added to the itemset's overall utility value in the database. `2P-UF` algorithm [32] was the first to address the problem of computing such patterns. It considers utility frequent motifs to be a subset of the high utility itemset problem. This approach is a two-phases algorithm based on a quasi-support measure that addresses the issue of the support-utility measure's non-monotonicity. Consequently, this algorithm is not suitable to handle large-scale databases. In contrast to `2P-UF`, `FUFM` algorithm [27] was introduced to handle the frequent-utility task as a subset of the frequent itemset mining problem. To find such patterns among frequent itemsets, it employs a metric known as extended support. Since it is based primarily on the frequent itemset mining approach, this algorithm is very simple and fast. A parallel version of `FUFM`, called `P-FUFM` [26], is a two-phases based algorithm that aims to reduce the running time of the `FUFM` algorithm. In the first step, this algorithm generates candidates, and in the second phase, it computes utilities. The scheme is to implement a parallel generation of candidates as well as the corresponding utilities. Unfortunately, all these designed algorithms suffer from a degradation in scalability for large scale databases.

In this paper, we are mainly interested in mining more efficiently frequent high utility itemsets from transaction databases in the case when the frequency metric is applied to transactions in which the itemset appears, and also when such metric is restricted to the transactions with highest gain value. To address these two problems, we propose a novel symbolic framework based on propositional logic to efficiently extract a concise set of itemsets that are frequently encountered while yielding the highest profit margins in a transaction database. In fact, symbolic Artificial Intelligence (AI) approaches, such as Boolean Satisfiability (SAT) and Constraint Programming (CP), are applied in data mining by translating the problem of mining patterns in terms of constraints and then delegate the enumeration of solutions to the appropriate solver (e.g., [2, 9, 14, 16, 20]). The application of symbolic AI to data mining is supported by its theoretical and algorithmic foundations and the flexibility it affords, i.e., the ability to add new user-specified constraints to control interesting patterns without the need to modify, from scratch, the underlying algorithms. Furthermore, the close relationship between constraint-based languages and pattern discovery enables data mining problems to benefit from a variety of powerful propositional satisfiability-based solving techniques in order to improve the efficiency of such approaches. In this paper, we propose two SAT-based approaches: the first aims to compute the set of frequent high utility itemsets, while the second is for identifying all patterns that are local high utility frequent itemsets.

2 Formal Preliminaries

2.1 High Utility Itemset Mining

Let Ω denote a universe of items (or symbols), called alphabet. The elements of Ω are denoted by the letters a, b, c, etc. A subset of Ω ($I \subseteq \Omega$) is called an *itemset*. The set of all itemsets over Ω are denoted as 2^{Ω}, and the capital letters I, J, K, etc. are used to represent the elements of 2^{Ω}. Typically, a *transaction* is an ordered pair (i, I) where $1 \leq i \leq m$, called the *transaction identifier* (TID, for short), and I an itemset, i.e., $(i, I) \in \mathbb{N} \times 2^{|\Omega|} \setminus \emptyset$. When there is no confusion, a transaction will be simply denoted as T_i. A *transaction database* D is defined as a finite non-empty set of transactions where each transaction identifier refers to a unique itemset. Given a transaction database D and an itemset I, the *cover* of I in D is defined as follows: $\texttt{Cover}(I, D) = \{i \in \mathbb{N} \mid (i, J) \in D \textit{ and } I \subseteq J\}$. The *support* of I in D is then defined as the cardinality of $\texttt{Cover}(I, D)$, i.e., $\texttt{Supp}(I, D) = |\texttt{Cover}(I, D)|$. A high utility itemset $I \subseteq \Omega$ s.t. $\texttt{Supp}(I, D) \geq 1$ is *closed* if and only if for any itemset J with $I \subset J$, $\texttt{Supp}(J, D) < \texttt{Supp}(I, D)$.

In the high utility setting, each item $a \in \Omega$ is associated with a positive number that indicates its *external utility* (e.g., unit profit). We write $w_{ext}(a)$ for the external utility of a. In addition, each item a in a transaction T_i is associated with a positive value $w_{int}(a, T_i)$, called its *internal utility*. Based on these two kinds of utility, the *utility of an item* a in a transaction T_i, written $u(a, T_i)$, is computed as follows: $u(a, T_i) = w_{int}(a, T_i) \times w_{ext}(a)$. Now, the utility of an itemset I in a transaction T_i, denoted by $u(I, T_i)$, is defined as $u(I, T_i) = \sum_{a \in I \subseteq T_i} u(a, T_i)$. Then, the *utility of an itemset* I in the entire database D is defined as $u(I, D) = \sum_{T_i \in D \ \mid \ I \subseteq T_i} u(I, T_i)$.

Given a transaction database D and a user-specified utility threshold θ, the classical high utility itemset mining problem aims at finding the set of all itemsets in D whose utility value is no less than θ. More formally, the aim is to compute the set $\{I : u(I, D) \mid I \subseteq \Omega, u(I, D) \geq \theta\}$. An itemset with a utility greater than the minimum utility threshold θ is called a *high utility itemset* (HUI, for short).

In order to prune the search space, existing proposals of HUIM use the so-called *Transaction Weighted Utilization* (TWU, for short), which is an upper bound of the utility measure, together with the property of anti-monotonicity in order to filter out the candidate itemsets that are not high utility [22]. More formally, the transaction utility of a transaction T_i in D, denoted by $TU(T_i)$, is the sum of the utility of all items in T_i, i.e., $TU(T_i) = \sum_{a \in T_i} u(a, T_i)$. Then, the transaction weighted utilization of an itemset X in a transaction database D, denoted by $TWU(X, D)$, is defined as: $TWU(X, D) = \sum_{(i,T_i) \in D \ \mid \ X \subseteq T_i} TU(T_i)$.

Another task in data mining related to HUIM problem consists in enumerating the set of HUIs by taking into account the frequency. Such itemsets are called *frequent high utility itemsets* (FHUIs, for short). To be precise, given a support and utility minimum thresholds δ and θ respectively, an itemset I is a FHUI in D iff. $\texttt{Supp}(I, D) \geq \delta$ and $u(I, D) \geq \theta$. We denote by FHUIM the task of computing the set of FHUIs in D. Clearly, the HUIM task is a particular case of FHUIM where δ is set to 1.

▶ **Example 1.** Consider the transaction database shown in Table 1 (which will be used throughout the paper). For the sake of simplicity, we set the external utility of each item to 1. In fact, every transaction database can be represented as a single table by multiplying the internal and external utilities of items. In that sense, each item has a single number that represents its utility in the transaction. Let $\theta = 20$ be a minimum utility threshold, and $\delta = 2$ be a minimum support threshold. Then, the set of FHUIs in D are $\{a\}$, $\{a, b\}$, and $\{a, b, g\}$.

Table 1 Sample Transaction Database.

TID	Items					
T_1	$(a, 8)$	$(b, 2)$				$(g, 1)$
T_2		$(b, 6)$	$(c, 3)$		$(e, 2)$	
T_3			$(c, 4)$	$(d, 3)$		
T_4	$(a, 6)$			$(d, 4)$	$(e, 1)$	
T_5	$(a, 8)$	$(b, 7)$			$(f, 2)$	$(g, 1)$

Now, we wish to emphasize that the utility of itemsets is over-estimated when mining FHUIs from transaction databases. In other words, all transactions containing the candidate itemset are considered without regard for their utility value in these transactions: even if the utility of an itemset I in a transaction T is low, T is chosen if it contains I. To be precise, the utility measure of an itemset I (i.e., $u(I, D)$) takes into account the utility of I in all the transactions where I appears. To alleviate such over-estimation in the problem of mining FHUIs, another specialized form of HUIM is to restrict the cover of the candidate itemset I to only the transactions in which the utility of I (i.e., $u(I, T_i)$) is greater than a local minimum utility threshold. Such HUIs will be called *frequent local high utility itemsets* (FLHUIs, for short). To define such sets, we need the following additional terminology.

▶ **Definition 2.** *Assume D is a transaction database, and θ' a local minimum utility threshold. Then, the utility-based cover of an itemset I in D is defined as:* $\texttt{Cover}_u(I, D) = \{(i, T_i) \in D,\ I \subseteq T_i,\ \text{and } u(I, T_i) \geq \theta'\}$. *Then, the utility-based support of I is the cardinality of its utility-based cover, i.e.,* $\texttt{Supp}_u(I, D) = |\texttt{Cover}_u(I, D)|$.

▶ **Property 3.** *Let D be a transaction database, I an itemset, and δ a minimum support threshold. Then, if* $\texttt{Supp}_u(I, D) \geq \delta$*, then* $\texttt{Supp}(I, D) \geq \delta$*.*

Given a support and local minimum utility thresholds δ and θ' respectively, an itemset I is a FLHUI in D iff. $u(I, D) \geq \theta'$.

▶ **Example 4.** Let us consider again Example 1. For the minimum support threshold $\delta = 2$ and the local minimum utility threshold $\theta' = 10$, the sets $\{a, b\}$ and $\{a, b, g\}$ are FLHUIs.

To avoid ambiguity, we will refer to the second problem of enumerating all FLHUIs from transaction databases as FLHUIM. This paper deals with a suitable reduction of both the problems of FHUIM and FLHUIM to the propositional satisfiability model enumeration task, and then the use of state-of-the-art SAT solvers to solve these two problems.

2.2 Propositional logic

Let $\mathscr{L}$ be a propositional language built up inductively from a countable set $\texttt{PS}$ of propositional variables, the boolean constants $\top$ (*true* or 1) and $\bot$ (*false* or 0) and the classical logical connectives $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ in the usual way. We use the letters x, y, z, etc. to range over the elements of $\texttt{PS}$. Propositional formulas of $\mathscr{L}$ are denoted by Φ, Ψ, etc. A literal is a propositional variable (x) of $\texttt{PS}$ or its negation $(\neg x)$. A clause is a (finite) disjunction of literals. For any formula Φ from $\mathscr{L}$, $\mathscr{P}(\Phi)$ denotes the symbols of $\texttt{PS}$ occurring in Φ. A formula in conjunctive normal form (CNF, for short) is a finite conjunction of clauses. A Boolean interpretation Δ of a CNF formula Φ is defined as a function from $\mathscr{P}(\Phi)$ to $\{0, 1\}$. A model of a formula Φ is an interpretation Δ that satisfies Φ, i.e., if there exists an interpretation $\Delta : \mathscr{P}(\Phi) \rightarrow \{0, 1\}$ that satisfies all clauses in Φ. The formula Φ is satisfiable if it has at least one model. In the sequel, we write $\models$ for the logical consequence relation and $\models_{\texttt{UP}}$ for the consequence relation restricted to the application of unit propagation[2].

The propositional satisfiability problem (SAT, for short) is a decision problem used to solve constraint satisfaction problems. Specifically, given a CNF formula Φ, SAT determines whether exists a model for each clauses in Φ. The application of SAT solvers in a range of real-world scenarios, e.g., electronic design automation, software and hardware verification [25], data mining [4], overlapping community detection in networks [17–19], has resulted from the progress of this NP-Complete problem over the previous decade. SAT technology has widely been used mainly in decision problems and its extensions such as SAT Modulo Theory (SMT), Maximum Satisfiability (Max-SAT), Quantified Boolean Formulas (QBF) but also recently in model enumeration problems built on top of modern SAT solvers such as Conflict Driven Clause Learning (CDCL) solver.

3 Computing High Utility Itemsets with Propositional Satisfiability

3.1 A SAT Approach to Frequent High Utility Itemset Mining

In this subsection, we deal with the translation of the FHUIM problem into propositional logic, so that SAT solvers can be used to enumerate FHUIs from transaction databases. We recall first that the traditional HUIM task has been recently reduced to SAT [15]. Specifically, in order to have a one-to-one mapping between the set of HUIs and the models of the

[2] Unit propagation is a kind of inference technique based on resolution with unit clauses (i.e., clauses containing exactly a single literal), e.g., $\Phi \wedge x \wedge (\neg x \vee \alpha) \models_{\texttt{UP}} \alpha$.

underlying propositional formula, a set of propositional variables and logical constraints have been introduced. More formally, given a transaction database D, the proposed encoding associates to each item a (resp. transaction identifier i) of D a propositional variable referred to as p_a (resp. q_i). Figure 1 depicts the different constraints that accomplish the SAT-based encoding scheme of the HUIM problem. To be precise, Constraint 1 encodes the candidate itemset's cover. This constraint expresses the presence of the itemset in the i^{th} transaction, i.e., $q_i = 1$. More specifically, the candidate itemset is not supported by the i^{th} transaction (i.e., q_i is *false*), when there exists an item a (i.e., p_a is *true*) that does not belong to the transaction ($a \in \Omega \backslash T_i$); when q_i is *false*. This means that at least one item not appearing in the transaction i is set to *true*. Constraint (3) captures the closedness requirement of HUIs. It ensures that if the candidate itemset is involved in all transactions containing the item $a \in \Omega$, then a must be added to the itemset (i.e., a must be propagated to *true*). The constraint over the utility of the candidate itemset in the database D is expressed using the linear inequality (4) w.r.t. the user threshold θ. Notice that Constraint (4) takes into account the TWU property to prune the search space.

In contrast to the work of [15], we tackle in this paper the problem of mining the set of all FHUIs from transaction databases. To do this, the previous encoding (Constraints (1), (3), and (4)) should be extended with Constraint 2. In fact, Constraint (2) requires that at least δ transactions involves the candidate itemset to be considered as a frequent HUI. Consequently, the problem of FHUIM is encoded in propositional logic with the propositional formula $\Phi^{\texttt{fhuim}} = (1) \wedge (2) \wedge (4)$. Moreover, a user can be interested in a more concise representation of FHUIs, called *closed* FHUIs (in short CFHUIs). Recall that the closedness constraint is encoded as Constraint 3. We shall note the formula encoding the computation of CFHUIs as $\Phi^{\texttt{cfhuim}} = \Phi^{\texttt{fhuim}} \wedge (3)$.

$$\bigwedge_{i=1}^{m}(\neg q_i \leftrightarrow \bigvee_{a\in\Omega\backslash T_i} p_a) \qquad (1) \qquad \sum_{i=1}^{m} q_i \geq \delta \qquad (2) \qquad \bigwedge_{a\in\Omega}(p_a \vee \bigvee_{a\notin T_i} q_i) \qquad (3)$$

$$\sum_{i=1}^{m}\sum_{a\in T_i} u(a,T_i)\times(p_a \wedge q_i) \geqslant \theta \qquad (4)$$

Figure 1 SAT-based Encoding Scheme for the FHUIM Problem.

▶ **Proposition 5.** *Let D be a transaction database, θ a minimum high utility threshold, and δ a minimum support threshold. Let $\Phi^{\texttt{cfhuim}} = \Phi^{\texttt{fhuim}} \wedge (3)$ be a propositional formula. Then, there exists a one-to-one mapping between the models of $\Phi^{\texttt{cfhuim}}$ and the set of CFHUIs in D.*

3.2 A SAT Approach to Frequent Local High Utility Itemset Mining

This subsection presents our formulation of the problem of FLHUIM into propositional satisfiability. First, recall that a FLHUI in a transaction database D is a local high utility itemset and it meets a minimum support threshold. In contrast to the previous SAT-based encoding (see Figure 1), three subsets of propositional variables are introduced, namely $\{p_a,\ a \in \Omega\}$, $\{q_i,\ i \in [1..m]\}$, and $\{r_i, i \in [1..m]\}$. Afterwards, to restrict the frequency metric to the transactions with highest gain value, we consider the new Constraint (5). Specifically, this constraint states that an itemset I is covered by a transaction T_i if it is contained in T_i and the utility of I in T_i meets the required utility threshold. Alternatively, Constraint (5) can be rewritten as the following formula: $\bigwedge_{i=1}^{m} r_i \leftrightarrow \sum_{a\in T_i} u(a,T_i)(q_i \wedge p_a) \geq \theta'$. Now, to constrain the candidate itemset to be frequent, i.e., to be covered by at least δ transactions, we add the cardinality constraint (Constraint (6)). If that is the case, we call such candidate itemset as a FLHUI.

$$\bigwedge_{i=1}^{m}(r_i \leftrightarrow q_i \wedge (\sum_{a\in T_i} u(a,T_i)\ p_a \geq \theta')) \quad (5) \qquad\qquad \sum_{i=1}^{m} r_i \geq \delta \quad (6)$$

Figure 2 SAT Encoding Scheme for Frequent Local High Utility Itemset Mining Problem.

▶ **Proposition 6.** *Let D be a transaction database, θ' a local minimum utility threshold, and δ a minimum support threshold. Let $\Phi^{\texttt{cflhuim}} = (1) \wedge (5) \wedge (6) \wedge (3)$ be a propositional formula. Then, there exists a one-to-one mapping between the models of $\Phi^{\texttt{cflhuim}}$ and the set of closed FLHUIs in D.*

▶ **Example 7.** Let us consider the transaction database depicted by Table 1. Then, the formula that encodes the problem of enumerating all closed FLHUIs in D with $\theta' = 10$ and $\delta = 2$ is written as follows:

$$\begin{array}{lclr}
\neg q_1 & \leftrightarrow & (p_c \vee p_d \vee p_e \vee p_f) & p_a \vee q_2 \vee q_3 \\
\neg q_2 & \leftrightarrow & (p_a \vee p_d \vee p_f \vee p_g) & p_b \vee q_3 \vee q_4 \\
\neg q_3 & \leftrightarrow & (p_a \vee p_d \vee p_f \vee p_g) & p_c \vee q_1 \vee q_4 \vee q_5 \\
\neg q_4 & \leftrightarrow & (p_b \vee p_c \vee p_f \vee p_g) & p_d \vee q_1 \vee q_2 \vee q_5 \\
\neg q_5 & \leftrightarrow & (p_c \vee p_d \vee p_e) & p_e \vee q_1 \vee q_3 \vee q_5 \\
r_1 & \leftrightarrow & q_1 \wedge (8p_a + 2p_b + p_g \geq 10) & p_f \vee q_1 \vee q_2 \vee q_3 \vee q_4 \\
r_2 & \leftrightarrow & q_2 \wedge (6p_b + 3p_c + 2p_e \geq 10) & p_g \vee q_2 \vee q_3 \vee q_4 \\
r_3 & \leftrightarrow & q_3 \wedge (4p_c + 3p_d \geq 10) & \\
r_4 & \leftrightarrow & q_4 \wedge (6p_a + 4p_d + p_e \geq 10) & \\
r_5 & \leftrightarrow & q_5 \wedge (8p_a + 7p_b + 2p_f + p_g \geq 10) & \\
\multicolumn{3}{l}{r_1 + r_2 + r_3 + r_4 + r_5 \geq 2} &
\end{array}$$

The SAT encodings of FHUIM and FLHUIM tasks involve the so-called Pseudo-Boolean constraints[3] (e.g. Constraints (2), (4) and (6)). Solving such kind of constraints has received an important attention by the SAT community since Pseudo-Boolean constraints naturally arise in many propositional encodings of real-world problems, including classical pattern mining, product configuration and community discovery in networks. A common way to solve Pseudo-Boolean constraints is by transformation into a SAT equivalent propositional formula and then use SAT solvers in order to verify satisfiability using various state-of-the-art encoding techniques [11]. Another way is to handle Pseudo-Boolean-constraints directly in the SAT solver [1]. Pseudo-Boolean problems can also be modeled as an integer program, in which the nonlinear constraints are linearized like [6] which used cutting resolution. The solver bsolo [23] also combines integer programming techniques with SAT-solving. On the other hand, cutting resolution has also been used to solve Pseudo-Boolean constraints [6]. Conflict analysis, introduced by Marques-Silva and Sakallah [24], is an important component of modern SAT-solvers. It allows SAT solvers to learn conflict clauses from intractable sub-problems. These clauses allow the solver to prune other branches of the search tree and use non-chronological backtracking. However, its extension to Pseudo-Boolean constraints is not obvious [5, 8].

In our case, each transaction gives rise to a Pseudo-Boolean constraint. Moreover, we deal with an enumeration based problem. Consequently, due to some scalability concerns, we choose to manage these constraints lazily as in [21]. As described in Constraint (6), the PB

[3] A Pseudo-Boolean constraint is an expression of the form $\sum_{i=1}^{n} a_i x_i \ op\ b$, where a_i and b are real coefficients, x_i $(1 \leq i \leq n)$ are propositional variables, and the operator op belongs to $\{=, \leq, <, >, \geq\}$.

constraint allows to catch the conditions under which r_i is propagated particularly to *false*: either when q_i is *false*, or when the utility of the candidate itemset in T_i is less than the fixed utility threshold θ'. The latter is managed thanks to the use of counters that allow to check if the sum of the utilities of items in the transaction T_i not assigned to false is less than θ'.

3.3 SAT-based Enumeration for High Utility Mining

In this subsection, we present our method based on the classical DPLL procedure [7] to enumerating the set of FHUIs and FLHUIs in transaction databases. We use a DPLL procedure to avoid adding blocking clauses and then to circumvent the growing up of the sub-formulas size. The enumeration is performed by a simple backtrack at each found model. This is motivated by the fact that the number of models can be large particularly in pattern mining. Algorithm 2 (See appendix) illustrates the pseudo-code of the enumeration process. As mentioned previously, since our encoding includes both clauses and Pseudo-Boolean constraints, the latter are managed lazily following [21]. As will be shown in the empirical evaluation, the encoding of all Pseudo-Boolean constraints into CNF can make the resolution inefficient for large databases, since each variable r_i implies a Pseudo-Boolean constraint, which can be huge depending on the number of transactions in the database. This can make the SAT-based encoding intractable in practice. To address this issue, we propose a slight modification of the DPLL-based algorithm by employing *propagators*, a key component of CSP and SMT (Satisfiability Modulo Theory) solvers. The propagator is based on counters. In fact, it proceeds in the same manner as the well-known TWU measure used in the HUIM literature. When the propositional variable p_a becomes *false*, the utility of the item a is subtracted from the sum of the utility of the sub-tables in which the item a appears. Propagators are used in Constraint (2). We use counters to detect when the total weight of a transaction is less than the fixed threshold. If it is the case, then the variable q_i is propagated to *false*. As a result, our algorithm is divided into two parts: unit propagation for the propositional part and propagators for dealing with Pseudo-Boolean constraints to handle frequency and utility based constraints. Furthermore, propagators' primary function is to check the satisfiability of the Pseudo-Boolean constraint. The idea behind using propagators within our DPLL based approach is mainly to check the consistency of the Pseudo-Boolean constraints or to infer useful propagation. More precisely, for a Pseudo-Boolean constraint of the form $\sum_{i=1}^{n} w_i x_i \geq k$, a counter is initialized with the value $w = \sum_{i=1}^{n} w_i$. Each time an x_i is assigned to *false*, its corresponding weight w_i is subtracted from w leading to $w - w_i$. If such value is less than k, then a backtrack is performed. Note that the FHUIM task involves a cardinality and a Pseudo-Boolean constraints, while in the FLHUIM problem we can rewrite the encoding using conditional Pseudo-Boolean Constraints of the form $y \rightarrow \sum_{i=1}^{n} w_i x_i \geq k$ [3]. Such constraints allows to capture conditions under which $\neg y$ can be propagated. It is also worth noting that assigning the variables representing items (p_a, $a \in \Omega$) allows to fix the remaining variables via propagation, i.e., p_a, $a \in \Omega$ is a *strong backdoor* [31]. Then, from an heuristic point of view, it is better to assign such variables first.

3.4 A Decomposition-based Approach for FHUIM & FLHUIM

In practice, the DPLL based enumeration algorithm suffers from scalability issues, particularly when the size of the propositional encoding is very large. This can have a significant impact on the approach's efficiency, as stated by [15]. In fact, without decomposition the number of clauses of the encoding is equal to $|\Omega| \times |D| - (\sum_{T_i \in D} |T_i|)$. This is equivalent to the number of missing items in the database. This value can be very huge for large datasets (.e.g., for

Kosarak for $\delta = 1000$, the number of non missing items is 34009483, then the number of clauses that represent Constraint (5) is $41270 \times 990002 - 34009483$, which exceeds 40 billion clauses). To address this issue, we apply a decomposition scheme to split the transaction databases into numerous bases of reasonable size. Using decomposition, the size of each sub-problem is significantly reduced. For instance, Constraint (5) leads to 31205214 clauses for *Kosarak*, which is the total number of clauses of the different sub-problems instead of 40 billion clauses. The main idea of decomposition is to avoid encoding the entire database in favor of solving many independent sub-problems of small size rather than a single large problem. Basically, given a propositional formula Φ and a variable $x_1 \in \mathtt{PS}(\Phi)$, the models of Φ are those of $\Phi \wedge x_1$ and $\Phi \wedge \neg x_1$. By generalizing such principle for a subset of variables $\{x_1, \ldots, x_n\}$, the models of Φ are those of $\Psi_1, \ldots, \Psi_n$ where $\Psi_i = \Phi \wedge x_i \wedge \bigwedge_{1 \leq j < i} \neg x_j$. In our case, we have $\{x_1, \ldots, x_n\} = \{p_{a_1}, \ldots, p_{a_n}\}$ and Φ corresponds to $\Phi^{\mathtt{fhuim}}$ (or $\Phi^{\mathtt{cfhuim}}$ for closed patterns) or $\Phi^{\mathtt{flhuim}}$ (or $\Phi^{\mathtt{cflhuim}}$ for closed patterns). Hence, solving $\Psi_i = \Phi \wedge p_{a_i} \wedge \bigwedge_{1 \leq j < i} \neg p_{a_j}$ can be obtained by considering only transactions containing the item a_i. In fact, since the variable p_{a_i} is *true*, the models of Φ_i is restricted to those containing p_{a_i}, which means the itemset including a_i. Clearly, splitting the CNF formula generates a set of independent sub-formulas that encode subsets of a specific subset of transactions of the original database. Consequently, this allows to avoid modeling the entire database and without causing too large number of clauses as well as the associated computational problems. It is important to note here that the order in which we solve the generated sub-problems has further a tremendous impact on the effectiveness of our approach. In particular, we believe that starting from the last sub-problem $\Phi \wedge \Psi_n$ is the best choice since it is the simplest one. In fact, all the variables representing items are assigned to *false* except one. This results to a smaller encoding size for the current sub-problem compared to the previous sub-problems. Interestingly, the declarativity of our SAT approach is preserved when applying the decomposition technique. Thus, one just need to add the user-specific constraints to each sub-formula encoding the sub-table obtained by decomposition.

▶ **Example 8.** Let us reconsider the transaction database in Table 1. Figure 3 depicts the sub-tables obtained by applying the decomposition principle.

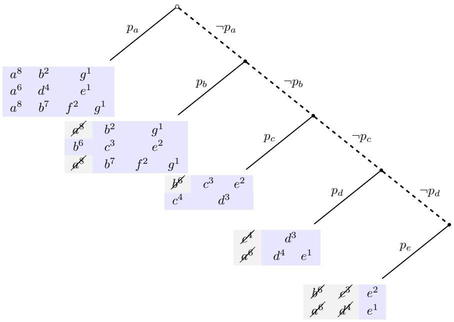

p_a
$\neg p_a$
a^8 b^2 g^1
a^6 d^4 e^1
a^8 b^7 f^2 g^1
p_b
$\neg p_b$
$\cancel{a^8}$ b^2 g^1
b^6 c^3 e^2
$\cancel{a^8}$ b^7 f^2 g^1
p_c
$\neg p_c$
$\cancel{b^6}$ c^3 e^2
c^4 d^3
p_d
$\neg p_d$
$\cancel{c^4}$ d^3
$\cancel{a^6}$ d^4 e^1
p_e
$\cancel{b^6}$ $\cancel{c^3}$ e^2
$\cancel{a^6}$ $\cancel{d^4}$ e^1

Figure 3 Item-based decomposition tree of the database in Table 1.

Algorithm 1 depicts our decomposition-based algorithm for mining frequent (local) high utility itemsets from transaction databases. This algorithm takes a transaction database, a (local) minimum utility threshold and a minimum support threshold as input, and returns all (closed) frequent (local) high utility itemsets. Based on the previously stated decomposition principle, our algorithm splits the transaction table into multiple independent sub-problems and restricts the encoding to a sub-table each time in order to enumerate all models corresponding to motifs of interest using the enumeration procedure described in Algorithm 2. To reduce the search space and, probably the encoding size, a pre-processing process is used to prune all itemsets that cannot be included in the final output. More specifically, in both FHUIM and FLHUIM, infrequent items are ignored. For the FHUIM method, if $TWU(a, D) < \theta$, then the item a is discarded, whereas for FLHUIM task, if $\sum_{a \in T} u(a, T) < \theta'$ with $T \in D$, then the transaction is not considered to be part of the search space.

Algorithm 1 **SAT** based **F**requent (Local) **H**igh **U**tility **I**temset **M**ining Approach.

```
Input: $D$: a transaction database, $\theta$: utility threshold, $\delta$: support threshold
Output: $S$: the set of frequent (local) high utility itemsets
$S \leftarrow \emptyset$;
for $i \in [1..n]$ do
    if $\mathtt{Supp}_u(a_i, D) \geq \delta$ then
        $D_i \leftarrow \{(k, T_k) \in D \mid a_i \in T_k\}$,    $\Omega = \langle a_1, \ldots, a_n \rangle \leftarrow items(D_i)$,    $\Gamma \leftarrow \emptyset$;
        for $T_j \in D_i$ do
            if $\sum_{c \in T_j} u(c, T_j) < \theta$ then
                $\Gamma \leftarrow \Gamma \wedge \neg r_i$,    $D_i \leftarrow D_i \setminus \{T_j\}$;
            end
        end
        for $b \in items(D_i)$ do
            if $\mathtt{Supp}_u(b, D_i) < \delta$ then
                $\Gamma \leftarrow \Gamma \wedge \neg p_b$;
            end
        end
        $\Psi \leftarrow p_{a_i} \wedge \bigwedge_{1 \leq j < i} \neg p_{a_j}$;
        $\Phi \leftarrow \Phi^{fhuim}(D_i, \theta, \delta) \wedge \Psi \wedge \Gamma$ ;                /* $\Phi^{flhuim}$ for FLHUIM */
        $S \leftarrow S \cup \mathtt{DPLL\_Enum}(\Phi)$
    end
end
return $S$;
```

4 Empirical Investigation

4.1 Experimental setup

Algorithm 1 is implemented in C++ language top-on the SAT solver MiniSAT [10], which is adapted to compute all models of a propositional formula by performing a DPLL procedure [7] as explained above. Our motivation here relies on the fact that we face on the problem of enumerating a huge number of models. For this, we adapt MiniSAT solver by keeping watched literals for unit propagation. Obviously, the restart and clause learning components can be disabled for better scalability and also to avoid growing the sub-formulas size. Note that the decomposition is performed by considering the frequency of items in ascending order, and the resulting sub-problems are addressed in a sequential manner. We have compared our proposed approaches against two baselines, namely `HU-FIMi` [28] and `FUFM` [27][4]. In our empirical evaluation, we conduct experiments over different commonly used benchmark datasets in the HUIM setting. These datasets are downloaded from the open-source data mining library SPMF [12]. All characteristics of both real and synthetic datasets are summarised in Table 4: the number of transaction (#Trans), the number of items (#Items), the average length of transactions (AvgTransLen), and the density (Density(%) for each dataset (see appendix).

Our experiments were performed on a machine with Intel Xeon quad-core processors with 32GB of RAM running at 2.66 GHz on Linux CentOS. Timeout was set to two hours for each run of an algorithm on a dataset. All experiments were conducted by varying the minimum support (δ) and the minimum high utility (θ) thresholds. It should also be mentioned that for our proposed algorithms, the computation time includes both the time needed for generating the CNF formulas and that for computing all models (i.e., itemsets of interest) of these formulas. We also note that the reported runtime in all the experiments is in seconds.

4.2 Results on Mining FHUIs

To evaluate the performance of our approach for mining frequent high utility itemsets, we consider a representative sample of real-world datasets (*Chess*, *Retail*, *Kosarak*, and *Chainstore*). We compared our `SATFHUIM` algorithm to the existing method named `HU-FIMi` [28][5]. Our approach is compared to this baseline according to running time and memory consumption for different minimum support and utility thresholds. Table 2 summarizes the empirical performance of our method against `HU-FIMi` on each dataset for each θ and δ values. Notice that the symbol (–) means that the algorithm is not able to complete the mining process under the fixed time out (i.e., TO). The size of FHUIM encoding in terms of the number of variables (#Var) and clauses (#Clauses) is given in Table 2.

According to our experimental results, our method outperforms the baseline across all datasets. In fact, `SATFHUIM` achieves interesting results on all databases when δ and θ are varied. For *Retail*, *Chainstore* and *Kosarak* datasets, `SATFHUIM` was respectively up to 39, 30 and 70 times faster than `HU-FIMi`. It can also be observed that on *Chess* dataset, `HU-FIMi` was enable to mine the target itemsets under the timeout except for $\delta = 50\%$ and $\theta = 400$k where the baseline took more than 4000 seconds to mine all FHUIs. However, for the same dataset `SATFHUIM` is able to scale for all minimum support and utility threshold values with a maximum running time of 1300 seconds. In terms of memory usage, `SATFHUIM` performs very well on both dense and sparse datasets, except for *Retail* and *Kosarak* datasets. This can be explained by the fact that even if the size of the generated sub-bases is small, the sub-problems could be numerous for these two datasets.

In our experiments, we also investigate the behavior of our SAT-based proposal `SATFHUIM` w.r.t. the running time and the number of FHUIs while varying both the values of θ and δ thresholds. The empirical results are depicted in Figures 4 and 5. The results show that `SATFHUIM` is able to solve all datasets even for small support and utility thresholds values where the number of obtained itemsets is huge. As illustrated in Figure 4, it is clear that the performance of our algorithm depends on the overall dataset characteristics. In addition, the minimum support δ as well as the minimum utility θ thresholds have a strong impact on the performance of the mining process. Specifically, for low values of θ and δ, `SATFHUIM` needs more time to discover all itemsets. The density value also has an impact on the execution

[4] We have used the C++ implementation for `HU-FIMi`, and the Python implementation for `FUFM`.

[5] We did not provide a comparison with `FCHUIM` because the source code is not public.

Table 2 Experimental results using different values of θ and δ.

			SATFHUIM					HU-FIMi		
Dataset	**δ(%)**	**θ**	**Time(s)**	**Memory (MB)**	**#conf**	**#Var**	**#Clauses**	**Time(s)**	**Memory (MB)**	**#cand**
Chess	30	200k	**1341.21**	**118**	5417007	75	4003366	–	–	–
		300k	**380.89**	**117**	3722143		4003184	–	–	–
		400k	**250.98**	**118**	539285		4002718	–	–	–
	40	200k	**363.171**	**117**	673551	75	3240897	–	–	–
		300k	**192.04**	**117**	1071967		3240897	–	–	–
		400k	**47.32**	**117**	426604		3240870	4022.47	149.16	1064837
	50	200k	**102.33**	**87**	103010	75	2930270	–	–	–
		300k	**70.14**	**87**	212158		2930270	3650.06	169.83	2485694
		400k	**28.15**	**87**	170565		2930243	1786.03	96.87	634997
Retail	0.2	1k	**1.83**	290	3167	16470	5835070	195.87	**145.44**	95912
		8k	**1.6**	207	1529		5500626	100.68	**146.92**	5877
		30k	**3.7**	156	1049		5084829	46.52	**131.91**	958
	0.4	1k	**0.96**	125	163	16470	2696209	61.48	**123.78**	30167
		8k	**0.92**	125	325		2650864	36.74	**123.22**	2652
		30k	**0.81**	124	402		2520038	18.95	**122.71**	482
	0.6	1k	**1.99**	125	163	16470	1785564	30.02	**118.44**	14591
		8k	**0.71**	**105**	121		1764079	20.64	117.87	1624
		30k	**0.64**	**105**	217		1710400	12.99	118.4	318
Kosarak	0.2	200k	**38.91**	**3011**	21977	41270	117031877	–	–	–
		400k	**32.20**	3004	4887		115627789	5348.66	**876.24**	191819
		600k	**28.71**	2986	1425		111434918	4002.07	**870.56**	75344
	0.4	200k	**18.23**	1514	1174	41270	49791548	1231.01	**860.74**	56174
		400k	**17.10**	1516	1306		49636024	849.15	**858.98**	18521
		600k	**16.03**	1509	816		49166188	652.24	**856.3**	7198
	0.6	200k	**12.79**	1154	446	41270	31646467	540.8	**839.82**	16552
		400k	**12.37**	1153	594		31594030	385.69	**826.5**	5531
		600k	**11.78**	1156	514		31498639	294.35	**826.89**	2358
Chainstore	0.2	1M	**7.8**	**754**	234	46086	15404456	245.79	939.4	1095
		1.4M	**7.5**	**752**	288		15184184	208.97	933.56	670
		1.8M	**7.19**	**747**	313		14899579	462.76	932.04	500
	0.4	1M	**5.44**	**523**	42	46086	7915721	123.13	896.05	503
		1.4M	**5.34**	**523**	69		7915721	117.42	877.91	346
		1.8M	**5.17**	**523**	89		7915721	269.54	887.14	240
	0.6	1M	**4.24**	**465**	10	46086	4481837	81.33	841.47	292
		1.4M	**4.16**	**465**	15		4481837	77.11	839.75	181
		1.8M	**4.08**	**465**	22		4481837	77.15	838.29	141

time. To be precise, `SATFHUIM` becomes slow on dense datasets, for instance it takes more than 1000 seconds to find all FHUIs for *Chess* dataset. In contrast, on sparse datasets, it is become easier to compute all FHUIs even for low thresholds values. This is the case for *Chainstore* dataset where the time needed to enumerate all patterns is only 80 seconds for low threshold values.

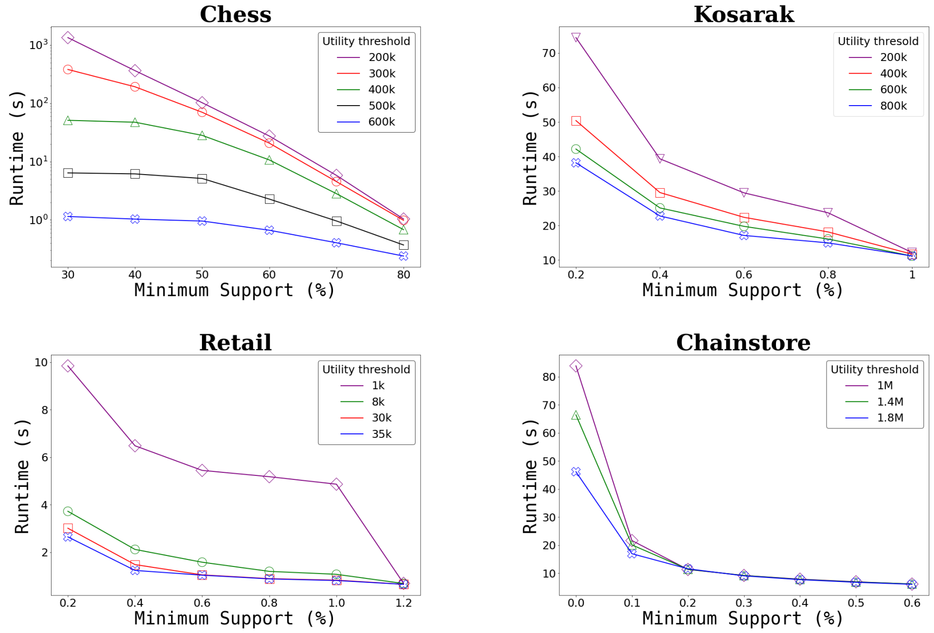


Figure 4 Running time of `SATFHUIM` w.r.t. minimum support threshold on real-world datasets.

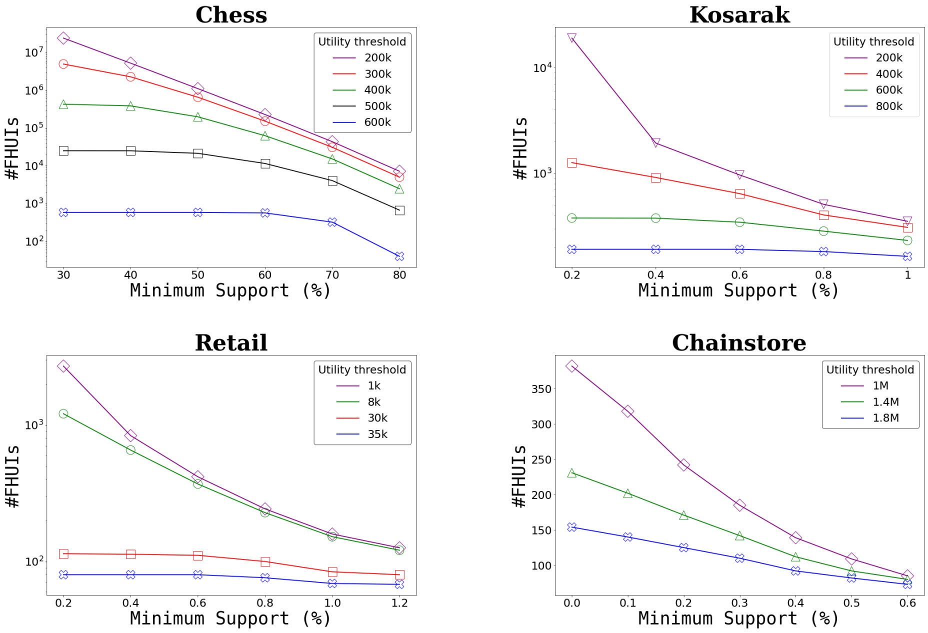


Figure 5 Number of `FHUIs` w.r.t. minimum support threshold on real datasets.

According to Figure 5, it is clear that the number of FHUIs always depends on the chosen values of θ and δ. In fact, for lowest threshold values this number becomes huge even for small datasets. For instance, on *Chess* and *Retail* the number can be more than 10^7 and 10^4, respectively, but still clearly small compared with the number of itemsets generated by HUIM algorithms[6] as the support constraint allows to discard an important number of patterns. Overall, we note that the number of FHUIs is always less to the number of HUIs.

4.3 Results on Mining FLHUIs

The second experiment was conducted to evaluate the performance of our `SATFLHUIM` algorithm and compare it with the state-of-the-art method `FUFM` [27]. This experiment was carried on the real datasets (i.e., *Chess*, *Retail*, *Mushroom*, *Accidents*, and *Chainstore*) and also on a synthetic one called *T60D10kI1k*. Note that *T60D10kI1k* was constructed using the transaction database generator in SPMF [12] (see Table 4 for the characteristics of this dataset). For this dataset, the internal (resp. external) utility values was generated using a uniform distribution in range [1,10] (resp. [1,6]). The parameters θ' and δ values were varied for the different datasets. Similarly to the previous experiment, we compare our approach against the baseline `FUFM` on both running time and memory usage for mining FLHUIs, followed by the number of generated FLHUIs. As our `SATFLHUIM` algorithm allows us also to mine closed FLHUIs, we add in the last column of Table 3 the number of closed FLHUIs (#CFLHUIs). Table 3 shows in addition the encoding size in terms of the number of variables (#var) and clauses (#Clauses). All experimental results are shown in Table 3. According to this latter, both algorithms produced the same output. On running time, `SATFLHUIM` performs well on all of datasets. Interestingly enough, our method outperforms the baseline `FUFM` for low values of δ and θ' where the number of generated FLHUIs is large. For instance, on *Chess* dataset and $\theta' = 100$ and $\delta = 70\%$, `FUFM` takes more than 1 hour to

[6] If the minimum support threshold $\delta = 1$, the set of FHUIs correspond exactly to the set of HUIs.

find all FLHUIs whereas SATFLHUIM does not exceed 2 seconds for the same task. This is primarily due to the fact that FUFM is a candidate generation-based method with a large number of candidates. Furthermore, as the support and minimum utility thresholds decrease, FUFM's runtime increases dramatically. From a memory usage point of view, there is a gap in terms of memory between SATFLHUIM and FUFM. For instance, FUFM consumes up to 18 times more memory than SATFLHUIM for *Chess* dataset. Roughly speaking, when the parameters δ and θ' decrease, the overall performance of the two algorithms begin to decrease, and vice versa. When compared to the number of FLHUIs, it can be seen that the number of closed FLHUIs does not decrease significantly for almost datasets, except for *Mushroom* and *Chess* where the number of closed FLHUIs decreases to half w.r.t. the number of FLHUIs.

Table 3 Experimental results using different values of θ' and δ.

			SATFLHUIM				FUFM			
Dataset	**δ(%)**	**θ'**	**Time(s)**	**Memory (MB)**	**#Var**	**#Clauses**	**Time(s)**	**Memory (MB)**	**#FLHUIs**	**#CFLHUIs**
Chess	70	100	**2.51**	**68**	75	1594175	3784.971	1469	17201	9167
		130	**1.36**	**68**		1590608	3741.68	1214	6004	3469
		150	**0.83**	**68**		1587442	3752.159	1071	2197	1378
	75	100	**1.04**	**68**	75	1476874	1434.251	577	5347	3343
		130	**0.57**	**68**		1472986	1420.615	530	1366	959
		150	**0.37**	**68**		1469384	1431.31	420	350	277
	80	100	**0.42**	**54**	75	1165792	541.532	259	1176	880
		130	**0.25**	**54**		1161235	540.855	271	147	129
		150	**0.19**	**54**		1157227	539.979	279	9	9
Retail	2	10	**0.4**	**104**	16470	814001	3108.718	186	39	39
		15	**0.39**	**104**		785221	3139.677	186	31	31
		20	**0.38**	**104**		763051	3106.071	186	28	28
	4	10	**0.32**	**96**	16470	554031	372.061	155	16	16
		15	**0.3**	96		531227	375.671	155	14	14
		20	**0.3**	**96**		514102	376.369	155	13	13
	6	10	**0.29**	**96**	16470	494337	256.184	148	14	14
		15	**0.29**	**95**		473078	254.18	148	12	12
		20	**0.27**	**95**		456605	255.564	148	7	7
	8	10	**0.3**	**96**	16470	494337	232.043	145	12	12
		15	**0.28**	**95**		473078	230.931	145	8	8
		20	**0.28**	**95**		456605	230.95	145	1	1
Mushroom	30	30	**0.67**	91	119	2205546	88.592	**59**	2286	383
		40	**0.64**	91		2200604	88.75	**65**	1869	343
		50	**0.61**	91		2193490	88.957	**65**	1390	282
	40	30	**0.32**	72	119	1554782	46.279	55	268	84
		40	**0.31**	72		1548637	45.725	**55**	146	53
		50	**0.28**	72		1539823	46.191	**59**	59	30
	50	30	**0.15**	58	119	806424	17.389	**49**	45	19
		40	**0.15**	58		799408	17.575	**46**	18	11
		50	**0.16**	58		790047	17.316	**47**	2	2
Accidents	70	40	**21.46**	**2544**	468	69675386	6798.121	3178	133	133
		45	**19.92**	**2555**		69542270	6950.867	3177	86	86
		48	**19.23**	**2553**		69466628	6841.843	3176	67	67
	75	40	**14.97**	**1977**	468	55728869	4190	3893	42	42
		45	**14.44**	**1977**		55599338	4216.008	3892	24	24
		48	**14.16**	**1976**		55523819	4259.47	2335	17	17
	80	40	**10.15**	1851	468	36447084	2238.2	**1051**	13	18
		45	**9.96**	1844		36271268	2170.082	**1488**	4	4
		48	**9.96**	1843		36177935	2133.187	**1488**	2	2
Chainstore	0.2	100	**10.49**	**929**	46086	18210234	777.52	1257	407	407
		500	**9.84**	**909**		16437597	775.13	1257	51	51
		800	**9.61**	**903**		16199306	768.64	1259	10	10
	0.4	100	**6.22**	**769**	46086	9475966	530.86	1263	142	142
		500	**5.94**	**672**		8255824	531.46	1265	18	18
		800	**5.91**	**668**		8101301	515.35	1266	3	3
	0.6	100	**4.26**	**676**	46086	5617273	206.81	1256	79	79
		500	**4.08**	**611**		4711977	212.15	1256	9	9
		800	**4.02**	**608**		4596272	198.97	1256	1	1

Figure 6 provides the execution times on the different datasets in the first line followed by the variation of the number of patterns in the second line. Due to space limitation, we did not provide the results for all θ' and δ values. Instead, each bar corresponds to the average number of FLHUIs for each δ threshold in terms of the average of all fixed θ' values.

According to these experimental results, the performance of our proposal is highly dependent on the dataset characteristics, and also on the thresholds values chosen. In fact, when θ' and δ are set to large values the runtime is quite similar for almost all datasets. This is understandable as the number of discovered patterns is small. However, for small threshold values there is a gap between the runtimes. It is also worth noting that the output size (i.e., the set of FLHUIs) is significantly decreased when compared to our SATFHUIM algorithm w.r.t. θ' threshold values. For instance, on *Chess*, the average number of FLHUIs is about 21744362 for all fixed θ' values and $\delta = 30\%$, whereas the number of FHUIs is 24081372 for the same δ value and $\theta' = 200$k.

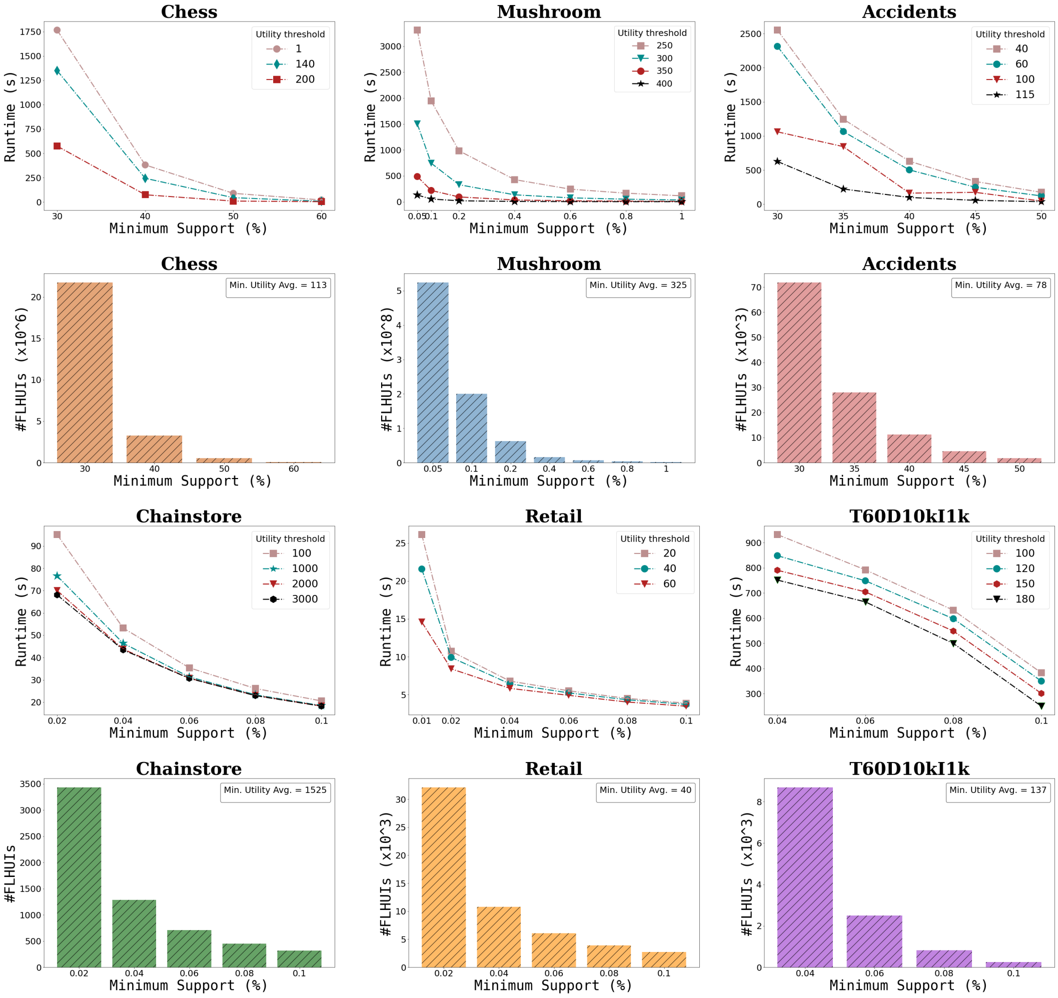


Figure 6 Experimental results of `SATFLHUIM` on several datasets.

5 Conclusion

In this paper we investigated how to solve the problem of mining (closed) FHUIs and FLHUIs from transaction databases using propositional logic. For the FHUIM task, we extended the existing approach of [15] with the frequency constraint, while for the FLHUIM problem we provided a new encoding using the well-known Pseudo-Boolean constraints. We extended the DPLL procedure to deal with both clauses and Pseudo-Boolean constraints in order to compute all models of CNF formulas. To scale up, a decomposition approach was presented, which allows the problem to be divided into several sub-problems of reasonable size. Empirical evaluation have shown how our approaches are very promising w.r.t. state-of-the-art.

In the future, we plan to investigate how to use propositional satisfiability to implement a limited but efficient clause learning in the context of patterns mining. In addition, by extending our approach for multi-objective optimization, we plan to investigate the problem of computing skyline HUIs from transaction databases using the two measures of interest (i.e., utility and frequency).

References

1 Fadi A Aloul, Arathi Ramani, Igor Markov, and Karem Sakallah. Pbs: a backtrack-search pseudo-boolean solver and optimizer. In *International Symposium on Theory and Applications of Satisfiability*, pages 346–353, 2002.

2 Mohamed-Bachir Belaid, Christian Bessiere, and Nadjib Lazaar. Constraint programming for mining borders of frequent itemsets. In *IJCAI*, pages 1064–1070, 2019.

3 Abdelhamid Boudane, Saïd Jabbour, Badran Raddaoui, and Lakhdar Sais. Efficient sat-based encodings of conditional cardinality constraints. In *LPAR*, pages 181–195, 2018.

4 Abdelhamid Boudane, Saïd Jabbour, Lakhdar Sais, and Yakoub Salhi. SAT-based data mining. *Int. J. Artif. Intell. Tools*, pages 1840002:1–1840002:24, 2018.

5 D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 305–317, 2005.

6 W. Cook, C.R. Coullard, and Gy. Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, pages 25–38, 1987.

7 Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, pages 394–397, 1962.

8 Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *AAAI*, pages 635–640, 2002.

9 Imen Ouled Dlala, Saïd Jabbour, Badran Raddaoui, and Lakhdar Sais. A parallel SAT-based framework for closed frequent itemsets mining. In *CP*, pages 570–587, 2018.

10 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2004.

11 Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *J. Satisf. Boolean Model. Comput.*, pages 1–26, 2006.

12 Philippe Fournier-Viger, Jerry Chun-Wei Lin, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Zhihong Deng, and Hoang Thanh Lam. The spmf open-source data mining library version 2. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 36–40, 2016.

13 Wensheng Gan, Jerry Chun-Wei Lin, Philippe Fournier-Viger, Han-Chieh Chao, Vincent S. Tseng, and Philip S. Yu. A survey of utility-oriented pattern mining. *IEEE Transactions on Knowledge and Data Engineering*, pages 1306–1327, 2021.

14 Tias Guns, Anton Dries, Siegfried Nijssen, Guido Tack, and Luc De Raedt. Miningzinc: A declarative framework for constraint-based mining. *Artif. Intell.*, pages 6–29, 2017.

15 Amel Hidouri, Said Jabbour, Badran Raddaoui, and Boutheina Ben Yaghlane. Mining closed high utility itemsets based on propositional satisfiability. *DKE*, page 101927, 2021.

16 Saïd Jabbour, Fatima Ezzahra Mana, Imen Ouled Dlala, Badran Raddaoui, and Lakhdar Sais. On maximal frequent itemsets mining with constraints. In *CP*, pages 554–569, 2018.

17 Saïd Jabbour, Nizar Mhadhbi, Badran Raddaoui, and Lakhdar Sais. Triangle-driven community detection in large graphs using propositional satisfiability. In *AINA*, pages 437–444, 2018.

18 Saïd Jabbour, Nizar Mhadhbi, Badran Raddaoui, and Lakhdar Sais. Sat-based models for overlapping community detection in networks. *Computing*, 102(5):1275–1299, 2020.

19 Saïd Jabbour, Nizar Mhadhbi, Badran Raddaoui, and Lakhdar Sais. A declarative framework for maximal k-plex enumeration problems. In *AAMAS*, pages 660–668, 2022.

20 Saïd Jabbour, Lakhdar Sais, and Yakoub Salhi. Mining Top-k motifs with a SAT-based framework. *Artif. Intell.*, pages 30–47, 2017.

21 Daniel Le Berre and Anne Parrain. The SAT4J library, Release 2.2, System Description. *Journal on Satisfiability, Boolean Modeling and Computation*, pages 59–64, 2010.

22 Ying Liu, Wei-keng Liao, and Alok Choudhary. A two-phase algorithm for fast discovery of high utility itemsets. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 689–695, 2005.

23 Vasco Manquinho and J. Marques-Silva. On using cutting planes in pseudo-boolean optimization. *Journal on Satisfiability, Boolean Modeling and Computation*, 2006.

24 João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.*, pages 506–521, 1999.

25 A Morgado and J Marques-Silva. Algorithms for propositional model enumeration and counting. Technical report, Citeseer, 2005.

26 A Sakthi Nathiarasan and M Manikandan. Performance oriented mining of utility frequent itemsets. In *International Conference on Circuits, Communication, Control and Computing*, pages 317–321, 2014.

27 Vid Podpecan, Nada Lavrac, and Igor Kononenko. A fast algorithm for mining utility-frequent itemsets. *Constraint-Based Mining and Learning*, page 9, 2007.

28 R Uday Kiran, T Yashwanth Reddy, Philippe Fournier-Viger, Masashi Toyoda, P Krishna Reddy, and Masaru Kitsuregawa. Efficiently finding high utility-frequent itemsets using cutoff and suffix utility. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 191–203. Springer, 2019.

29 Jilles Vreeken and Nikolaj Tatti. Interesting patterns. In *Frequent Pattern Mining*, pages 105–134. Springer, 2014.

30 Tianyou Wei, Bin Wang, Yuntian Zhang, Keyong Hu, Yinfeng Yao, and Hao Liu. FCHUIM: Efficient frequent and closed high-utility itemsets mining. *IEEE Access*, pages 109928–109939, 2020.

31 Ryan Williams, Carla Gomes, and Bart Selman. On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. *structure*, 23(4), 2003.

32 Jieh-Shan Yeh, Yu-Chiang Li, and Chin-Chen Chang. Two-phase algorithms for a novel utility-frequent mining model. In *Emerging Technologies in Knowledge Discovery and Data Mining*, pages 433–444, 2007.

A Appendix

Algorithm 2 DPLL_Enum: A DPLL backtrack search for Model Enumeration.

```
Input: $\Phi$: a CNF formula
Output: $S$: the set of all models of $\Phi$
$\Delta = \emptyset$; $S = \emptyset$;
if $(\Phi \models_{\text{UP}} p)$ then
    return DPLL_Enum$(\Phi \wedge p)$ ;                          /* unit clause */
end
if $(\Phi \models_{\text{UP}} \bot)$ then
    return $\emptyset$ ;                                          /* conflict */
end
if check_Pseudo_Boolean_constraint() == False then
    return $\emptyset$;
end
if $(\Delta \models \Phi)$ then
    $S \leftarrow S \cup \{\Delta\}$ ;                            /* new found model */
    return $\emptyset$
end
$p$ = select_variable($Var(\Phi)$);
$\Delta \leftarrow \Delta \cup \{p\}$; $S \leftarrow S \cup$ DPLL_Enum$(\Phi \wedge \Delta)$;
$\Delta \leftarrow \Delta \cup \{\neg p\}$; $S \leftarrow S \cup$ DPLL_Enum$(\Phi \wedge \Delta)$;
return $S$;
```

Table 4 Datasets Characteristics.

Instance	#Trans	#Items	AvgTransLen	Density(%)
Chess	3196	75	37	49.33
Mushroom	8124	119	23	19.33
Retail	88162	16470	10.3	0.06
Accidents	340183	468	33.8	7.22
Kosarak	990002	41270	8.1	0.02
Chainstore	1112949	46086	7.23	0.02
T60D10kI1k	10000	1000	30.4	3.04

Understanding How People Approach Constraint Modelling and Solving

Ruth Hoffmann
School of Computer Science, University of St Andrews, UK

Xu Zhu
School of Computer Science, University of St Andrews, UK

Özgür Akgün
School of Computer Science, University of St Andrews, UK

Miguel A. Nacenta
Department of Computer Science, University of Victoria, Canada

Abstract

Research in constraint programming typically focuses on problem solving efficiency. However, the way users conceptualise problems and communicate with constraint programming tools is often sidelined. How humans think about constraint problems can be important for the development of efficient tools that are useful to a broader audience. For example, a system incorporating knowledge on how people think about constraint problems can provide explanations to users and improve the communication between the human and the solver.

We present an initial step towards a better understanding of the human side of the constraint solving process. To our knowledge, this is the first human-centred study addressing how people approach constraint modelling and solving. We observed three sets of ten users each (constraint programmers, computer scientists and non-computer scientists) and analysed how they find solutions for well-known constraint problems. We found regularities offering clues about how to design systems that are more intelligible to humans.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Human-centered computing → Empirical studies in interaction design

Keywords and phrases Constraint Modelling, HCI, User Study, Grounded Theory

Digital Object Identifier 10.4230/LIPIcs.CP.2022.28

Supplementary Material *Dataset (Artefacts, Raw Observation Data, Intercoder Data, Code Explanations)*: `https://doi.org/10.5281/zenodo.6519841`

Funding This work is partially funded by NSERC Discovery Grant 2020-04401 (Canada).
Xu Zhu: University of St Andrews and EPSRC grant DTG1796157.

1 Introduction

Research in constraint programming (CP) techniques and algorithms during the last few decades have resulted in significant advances in how a large number of problems of practical significance can be addressed. For example, companies routinely use CP to schedule delivery routes [34], educational authorities leverage CP to match medical students to training positions [5] and administrators of high-performance computing clusters use CP for scheduling jobs [15]. CP has been used in commercial settings for several decades [13].

Despite the demonstrated value of CP in these and many other applications, CP remains the domain of a relatively small set of specialists and, arguably, an underappreciated area of computer science. In his seminal paper from 1996 [11], Freuder identifies the potential of CP technology for widespread adoption. In his recent paper from 2018 [12] he recognises the significant progress made by the field and identifies the next challenge as *ease of use*.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 28; pp. 28:1–28:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We believe that an increase in awareness of the hard-won knowledge in CP and increased accessibility of its techniques and tools (e.g., solvers, constraint modelling languages and modelling tools) can deliver positive improvements for individuals and communities.

For example, consider how a programmer might be able to generate a more efficient solution to a problem if they recognize a subproblem as a constraint satisfaction problem (CSP) for which they can include a CP solver library. Similarly, a person without specialist programming knowledge who identifies a problem as a CSP is more likely to learn how to access existing knowledge and software to find a solution.

One of the barriers to this wider awareness and better understanding of CP by a wider population is that CP tends to be approached from a mathematical, logical or computational point of view and this can be conceptually challenging. To make progress towards this goal we instead approach this problem from a human-centered point of view. In this work we study how people with different levels of expertise with respect to computer programming and CP think about CP and the process of solving constraint problems. Our assumption is that we can leverage a better understanding of how people think about constraint problems and their solutions to create tools such as CP languages or CP applications that are easier to access and use by non-specialists. For example, a programming language that uses core concepts closer to people's default conceptual approach to their problem will, presumably, require less effort to learn and co-opt as a personal tool [16]. Although we do not explicitly focus on improving the teaching of CP technology, this same information can help improve the teaching of constraint programming to novices because it offers a model of how people are likely to think about constraint problems.

We present a qualitative study of 30 participants with different degrees of familiarity with computer science and CP attempting to solve constraint problems. Our analysis shows that non-experts are not aware of the implications of the problem representation and will gravitate to simpler visualisations. Similarly, the visualisation/representation of solving strategies is something that has an impact on the understanding of a solving step. Finally, we found that there is a general belief that constraint problems are solved strategically (almost mathematical), rather than through search. The results from our analysis represent an initial step towards a better understanding of people's conceptual models of constraint problems and solvers. We also discuss ways in which this analysis is relevant for the design of CP languages and tools.

2 Overarching Goal

The overarching goal of this research is to gain knowledge into how people think about constraint problems and their solution. We work under the assumption that gaining this knowledge will help researchers and practitioners in CP in several meaningful ways: 1) A better understanding of how people think about CP can support designers of the languages and interfaces that users of CP come in contact with. For example, an end-user CP language could use terms or constructs that are more likely to be correctly understood by the user. In turn, this could result in more usable, easier to learn or faster to write languages. 2) Communication between the solver software and the end user is likely to benefit from a better understanding of the end-user's expectations. For example, an end-user could benefit from explanations provided by the solver software when performance is likely to mismatch the end-user's expectations (e.g., if an additional constraint results in computation times orders of magnitude larger). In the future, solvers might be able to provide explanations of performance that support the user's modeling activity, but to do this it is important to also understand how the user thinks about the problem. 3) Learners and future practitioners of CP, including future researchers in the area, can benefit from courses and instructors that are aware of which concepts, techniques and procedures come more naturally to learners of different backgrounds.

3 Methodology

To make progress towards the goal stated in the previous section, we designed a controlled observation experiment in which people attempt to visually model, solve, and program or constraint model constraint problems as described in a previous paper [40].

3.1 Participants

We recruited 30 participants from local universities, 10 belonging to each of the three expertise groups: non-computer scientists (identified as non-CS), computer scientists (CS), and constraint programmers (CP). Non-CS participants (7 female & 3 male, between 19 and 28 years in age), were non-computer scientists with negligible or no programming experience. Of the non-CS participants, 1 was studying towards a degree in a science faculty, 2 in a medicine faculty, and 7 in an art faculty. CS participants (4 female & 6 male, between 19 and 42 years in age) were students in a computer science degree with little or no experience in constraint programming but with experience of computer programming. CP participants (1 female & 9 male, between 21 and 64 years in age) were a mixture of students and staff who have either taken a constraint programming module, taught one or conduct research in that area. Participants received gift vouchers as compensation for their time. The three distinct groups were chosen because the way in which people solve problems is likely to be influenced by their experience and formal education, and they represent a range of levels of familiarity with formal problem specifications.

3.2 Procedure, Tasks and Problem Selection

In the experiment, participants were asked to visually model, solve, and program or model specific constraint problem instances by hand. Each participant provided written consent and was then assigned two problems. Problems were selected from a pool of constraint problems collected from CSPLib [21]. The selected problems are: Word Crypto, Subset Sum, Sudoku, Scheduling, Magic Square, and Knapsack. The exact formulations are in the supplementary materials. For the exact problem selection criteria, see the previous paper [40].

Out of all the participants, Sudoku and Subset Sum were attempted by two participants from each of the expertise groups while the rest were each attempted by 4 participants in each of the groups. For each of the two problems assigned to them, participants had to carry out a visual modelling task as well as a problem solving task, in sequence. Programmers in the CS and CP groups had to perform an additional programming/constraint programming task after. Participants always completed the tasks in order, which precludes bias due to the additional tasks performed by the CS and CP groups. Participants had 14 minutes to complete each task.

The visual modelling task was analysed in the previous paper [40]. For the solving task, participants were given the same problem they had during the modelling task and asked to try to solve the problem by hand, without computer aid. In this paper we focus exclusively on the solving task, although parts of the analysis is influenced by the visual modelling task from the previous paper. In this task, participants were encouraged to talk aloud their

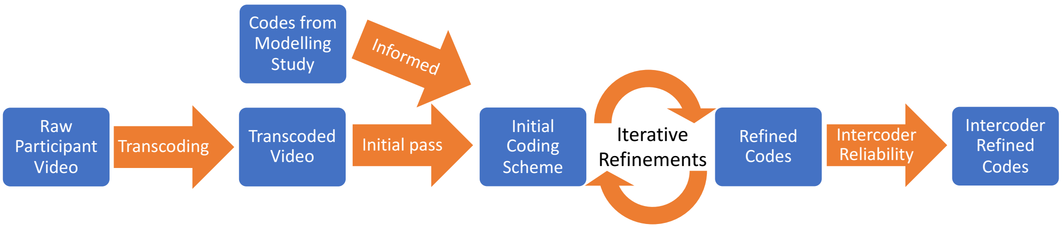


Figure 1 Flowchart of the different steps of the analysis methodology.

thought processes and decisions to allow the experimenter to more accurately understand what they were doing [39]. However, participants did not always do this and often needed prompting to describe their thoughts as they were trying to solve the problems. Occasionally, the experimenter would ask for clarifications or offered short reminders of the task. As part of the analysis, we try to infer thought processes from their actions.

We had trouble persuading one of the participants to solve the problems by hand. We have included their data as the participant was persuaded to describe how they would attempt to solve the problem.

3.3 Analysis Methodology

The analysed data consists of two streams of video for each of the participants (resulting in a total of around 32 hours of video per stream), and the paper output from their specifications (scans available in the Supplementary Materials). The two video streams captured two different views, one directly overhead and one pointing diagonally onto the workspace. Snippets from these materials in the remainder of this paper appear marked with the expertise group (non-CS, CS, CP), the number of the participant within that group (from 1 to 10) and whether this was their first or second problem (e.g., CS 7.2).

We followed a bottom-up open coding approach inspired by grounded theory [17]. We analysed both the artefacts from each participant (written notes and solutions) and the video recording of the solving process. An overview of the analysis workflow can be seen in Figure 1. As a preliminary step, we transcoded the two video streams to allow simultaneous viewing of the different camera angles. In a first analysis step, we analysed the artefacts and video produced by creating an affinity diagram of common occurrences and general themes. We then iteratively coded the features that appear within the artefacts and video using Microsoft Excel, refining the code list on each pass. A subset of the codes were initially adopted from the previous analysis from the visual modelling task [40], but were then refined for this analysis. However, most of the final codes (briefly summarised in Subsection 3.4) ended up being specific to this task. The authors met several times during this period to clarify any ambiguities in the codes and refine them.

3.4 A summary of the codes used in the analysis

The codes for annotating the participant behaviour are split into 3 categories. These categories are Visual Elements (VE), Memory related visual elements (MEM), and Solving Approaches (SA). In Table 1 we provide a small selection of codes to preserve space, the full list of codes can be found in the Supplementary Materials.

The VE codes describe what the participants wrote or drew on the paper, for example if they used the representation they were given in the problem statement (VE1.9) or if they crossed out/scribbled out any of the work they had written down at this point (VE1.11).

MEM codes indicate concepts written on the paper as well but they are used to keep track of information, such as keeping track of temporary solutions (MEM3) or having a representation of globally available values (MEM4.1).

Finally, the SA codes are a combination of behavioural observations and what is being written down or said by the participant in relation to their solving steps. The SA codes were split into problem specific strategies and universal strategies which can be found or used in any of the problems. Two examples in the universal strategy category are codes that identify whether a participant restarted solving the problem from scratch without remembering anything from their previous attempt (SA1.2), or if the participants followed a trial and error approach through random partial assignments of values to the variables without any rhyme or reason (SA1.3). For the problem specific strategies, we coded not only the strategies that made sense for the problem (such as SA4.2 where in the Subset Sum problem participants did brute force matching by creating all possible set sizes and then evaluated them), but also strategies that the participants used but which would not lead to a solution (such as SA2.5, where the participants packed the knapsack by the weight or value of the product, rather than the more traditional ratio).

3.5 Coding Validation Analysis

The bulk of the coding was performed by one of the authors. In order to ensure the robustness of the process, the remaining three authors performed an independent coding pass of a subset of 3 of the 60 videos. The inter-coder reliability ratio (the number of agreements divided by the total number of codes from the results of the independent coding pass) and the Cohen Kappa statistic [25] (calculated using the `scikit-learn` python library [32]) measured an inter-coder reliability ratio of 79% and a Cohen Kappa statistic of 0.50.

In an iterative second stage, all authors discussed any ambiguities and discrepancies between codings and resolved some of their differences. Some of these differences resulted from either lack of clarity regarding the scope of a particular code, or lack of familiarity with the video and missing a briefly appearing code. The authors then prepared another code pass based on this discussion. The post-discussion codes have a 92% agreement. The Cohen Kappa statistic post-discussion is 0.80, which is a significant improvement from pre-discussion result. The Supplementary Materials contain the CSV files and python code used to carry out the Kappa calculations.

4 Findings

We describe the regularities that we found in how people approach the constraint problem solving process roughly in chronological solving order. We split the process into three stages.

In the first stage, participants needed to decide how to represent the problem. When presenting the problem statement to them, we have a problem representation as part of the statement. Participants might reuse this representation or they may choose to re-represent the problem in a way that is more intuitive to them before they start operating on it. We discuss this process in the Representation section (Subsection 4.1).

Solving the problem usually involves proposing candidate (partial or complete) solutions and noting this solution on the chosen representation. Participants use several different approaches at this stage (Process and Strategies, Subsection 4.2).

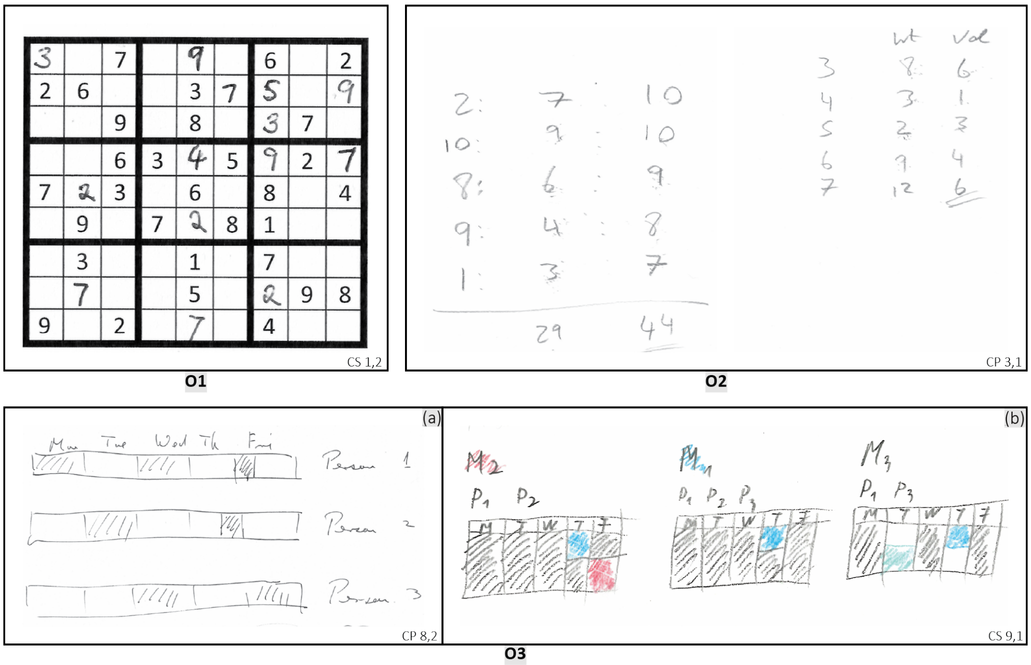
Finally, delivering a solution might involve verifying that the candidate solution is correct and, in the case of solution enumeration or optimisation problems, whether there are other valid (or better) solutions (Solution Verification and Multiple Solutions, Subsection 4.3).

Table 1 Representative sample codes for the four groups of codes used in the analysis. We have a total of 77 codes in the four groups listed above. The breakdown is 23 in the VE group, 6 in the MEM group, 11 in the SA1 group and 48 in the SA2 group. SA2 is the largest since it contains problem specific codes for the 6 problems. Full list of codes can be found in supplementary materials.

Group	Code	Name	Description
Visual elements	VE1.9	Recreating Instance	Use of the visual element given in the problem description or recreation of the representation used in the previous part of the experiment.
	VE1.11	Crossing out	Line(s) through a single or more characters, or scribbles over a larger area.
Memory elements	MEM3	Temporary solution	Noting and pointing out a temporary solution. This solution could be incorrect or partially valid. A temporary solution is one where there is a majority of variable and value assignments, and is written out as such.
	MEM4.1	Visual representation of globally possible values	Note (or a mark) of possible values that can be assigned to a variable. The assessment of the possibilities was made with all variables or values in consideration.
Solving approaches	SA1.2.1	Restart remembering (no)goods	The participant has decided to give up on their current partial (or full) solution. It is irrelevant whether that solution is correct. They then start the solving from the beginning and remember something they about the variables and values.
	SA1.3	Random Partial Assignment	A strategy of randomly assigning a few variable value pairs.
Problem specific SA	SA2.5	By weight value product	The set of object will be sorted into a highest to lowest weight by value product. A subset of the highest products will be chosen.
	SA4.2	Brute force by set size	The participant creates all sets of a given set size and checks which have the correct sum. This set enumeration can be done explicitly or implicitly.

At each step we found similarities and differences between the different groups of participants; we introduce these as appropriate. Note that, due to the design of the study, participants were exposed to different subsets of problems and some regularities belong to specific problems or problem types (see Section 3 for details). To facilitate the interpretation of the results in Section 6 we mark the notable observations of this Section with labels (e.g., **O1**, **O2**, etc.).

In some cases we provide aggregate measures about problems across different participant groups. Unless otherwise stated, aggregate numbers (e.g., 5 in the CS group and 21 in the CP group) refer to *problems*, not participants. Since participants did two problems each, the total counts are out of 20 for each of the groups or out of 60 for the whole data set.



3		7		9		6		2
2	6			3	7	5		9
		9		8		3	7	
		6	3	4	5	9	2	7
7	2	3		6		8		4
	9		7	2	8	1		
	3			1		7		
	7			5		2	9	8
9		2		7		4		

2:	7	10
10:	9	10
8:	6	9
9:	4	8
1:	3	7
	29	44

	wt	val
3	8	6
4	3	1
5	2	3
6	9	4
7	12	6

Figure 2 Artefacts from our user study that support the observations regarding representations. These are explained in Subsection 4.1. Each sub-figure is labelled with the name of the observation. Each image is labelled with the anonymised identifier of the relevant participant.

4.1 Representation

Participants were given an initial written problem statement (the exact problem statements are included the Supplementary Materials). In itself, the problem statement can be seen as a representation of the structure of the problem. Sometimes, these representations are adequate to directly operate on (Sudoku and Magic Square can be solved by filling the numbers on the grid provided and Word Crypto allows the reuse of the same shape used in the problem statement for the summation), whereas in other problems the participants need to come up with a representation that is not given in the problem statement itself (e.g., to solve the Knapsack problem one has to create some kind of list or sequence of objects).

(O1) When solving a problem, participants attempted to solve most problems (70%) using a representation they already had. That is, participants used a verbatim copy of the representation from the problem statement or, when this was not available, the representation that they themselves created in the modelling part of the study. See Figure 2.O1 for an example of reusing a given representation when solving a Sudoku instance. It has been long known that how a problem is first framed or presented to a human solver can have an effect on how it gets solved (e.g., [19, 35]), due the differing representation of the problem by people [37]. We refer to this issue as *framing* or *representational inertia*.

(O2) It is still key to highlight that framing has a sizeable effect on how the problem is understood, at least initially. Participants in the CS and CP groups were more likely to adopt alternative and more sophisticated representations of the problems such as using a multi-column table for the Knapsack problem (See Figure 2.O2), which is akin to using more sophisticated data structures to address the problem. This suggests that one of the key differences between experts and non-experts is the awareness of representational alternatives.

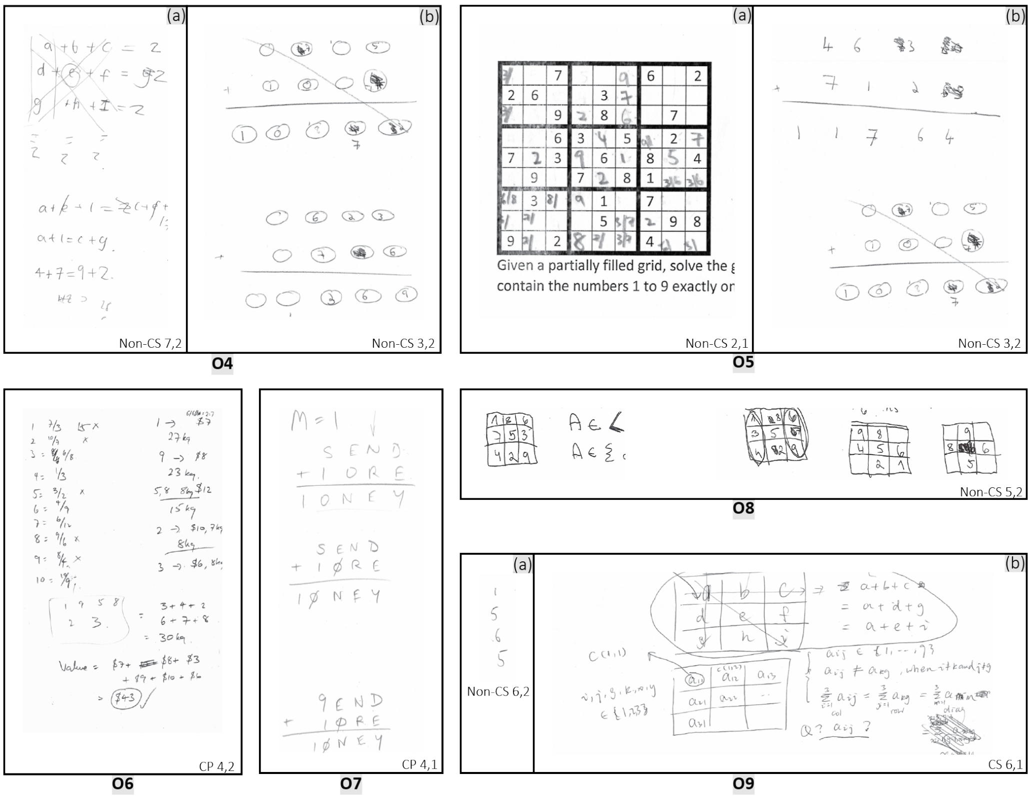


Figure 3 Artefacts from our user study that support the observations regarding process and strategies. These are explained in Subsection 4.2. Each sub-figure is labelled with the name of the observation. Each image is labelled with the anonymised identifier of the relevant participant.

(O3) Beyond the structure of the representation itself, some of the problems allowed for alternative choices in the focus of the representation. For example, the scheduling problem could be solved either by modelling the problem from the perspective of the people or focusing on a representation of a timetable. Figure 2.O3 gives an example of these two perspectives ((a) focusing on people, (b) using a timetable). This choice is likely to have an impact on the ability of the participant to solve the problem and correctly understand it. Additionally, the choice is likely to be affected by the semantics of the problem. That is, depending on the nature of the problem, participants might be inclined to model in a way that is not necessarily the most efficient. We suspect that in a problem which involves people a model that puts the person at the center might be preferred at least initially to a more "abstract" way to model the situation. This is consistent with our observation of the scheduling problem, where we saw all participants using a people-centric view of the problem.

4.2 Process and Strategies

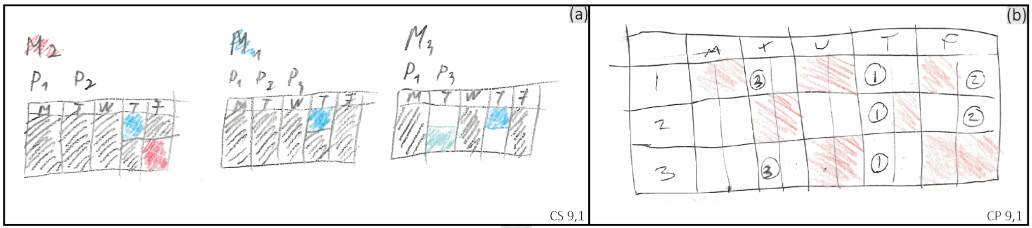
Once participants have settled on a representation (or re-representation) they need to get into the process of finding solutions. This is where we find that the assumptions and attitudes towards the problem solving process can have the most dramatic effects.

(O4) We observed, for example, that many participants in the non-CS and CS groups thought that there is an ideal way of finding the solution that does not include too much guesswork. Their assumption is that there is some kind of mathematical formula, algorithm, hidden insight, or an optimal sequence of steps through the solution space that will lead them to a solution without much effort and, presumably, with some level of satisfaction. We find evidence of this attitude in some of the approaches of the participants where they attempt to use a mathematical approach instead of trying to make progress with the puzzle with more intuitive steps of inference. In one case a participant says "there must be a mathematical approach to this". In another case, when solving the Magic Square problem, we observed a participant who turned the grid into a system of equations, which were then meant to be solved to find the common sum of the rows, columns and diagonals (see Figure 3.O4).

(O5) Some people simply apply a trial and error approach, in a similar fashion to a random partial (or full) assignment. This seems to be more prevalent among non-CS participants, 7 apply trial and error, in comparison to 5 and 3 in CS and CP categories respectively. For example, one of the non-CS participants, who has never solved a Sudoku puzzle before, started by filling the squares with seemingly random numbers (see Figure 3.O5.a). They then started noticing the implications of the filled in numbers (almost in the style of propagation) and picked up on more common solving strategies such as block/row/column elimination. Another participant from the non-CS group started by assigning a subset of the numbers of the Word Crypto problem randomly and trying to fill in the rest greedily (see Figure 3.O5.b).

(O6) As seen above trial and error is a natural first approach when one does not have a good understanding of the structure of the problem or lacks experience in how to address it. However, we found that participants can have quite different reactions to the almost inevitable discovery that the first trial does not comply with the constraints. In some cases participants use a backtracking process in which they undo a certain number of steps (one or more) that they have found to be incorrect or undesirable (e.g., because it might be, or be perceived as, a dead end), to then try a different alternative. This is a fairly sophisticated approach that generally requires keeping a more sophisticated tally of the options tried and the order they occurred in. In general we found that CP participants were less likely to backtrack as they might have a better grasp of the inference of a solution step, we observed 6 CP backtracking whereas 9 CS and 10 non-CS backtracked. But when the CP participants were to backtrack they were generally more likely to employ more sophisticated backtracking. Figure 3.O6 shows an example of this, in which the participant greedily populates the knapsack to find an intermediate solution. Then, they undo some of their decisions and replace the items with higher value items to improve the total value of the solution.

(O7) The alternative to backtracking is a restart of the solving problem process. This seemed to be the preferred approach for many, especially in the non-CS group where we observe restarts 12 times, in comparison to 7 and 2 in the CS and CP groups. In Figure 3.O7 we can see how the participant restarted the Word Crypto problem twice, while still remembering some of the information from their last approach. Similarly Figure 3.O4.b shows a participant who attempted the problem a few times from scratch with different strategies, even though they did not make any mistakes. Participants from the CP group seem to have a more realistic understanding of the solving process, and we observed many of them systematically exploring the solution space instead of prematurely restarting from scratch.



O13

Figure 4 Artefacts from our user study that support the observations regarding how participants check their candidate solutions. These are explained in Subsection 4.3. Each sub-figure is labelled with the name of the observation. Each image is labelled with the anonymised identifier of the relevant participant.

(O8) Amongst participants who applied restart approaches we observed two main subsets. A majority (17 vs 8 spanning all groups) restart the process without any evident sign of having learned anything from the previous approach. In Figure 3.O8 we have observed the participant attempting to solve the Magic Square problem, by doing four consecutive trials without making any substantial changes to their solving approach.

(O9) In contrast, other participants seem to learn about the structure of the problem after the first attempt and seamlessly transition into a more structured approach. For example, in Figure 3.O9.a we see a participant who started solving the Subset Sum problem by choosing a seemingly random set of numbers. They then realised that they can split the numbers into two groups (positive and negative numbers) and attempt to balance their selection across the two groups. Note that this is more likely to happen for certain kinds of problems due to their more obvious solution space structure. As seen above this is applicable in the Subset Sum problem, but also in the Magic Sum problem, where participants filled in the grid partially to see that in fact some numbers cannot be (or vice versa have to be) in certain cells. In some cases participants transitioned to approaches that are not necessarily useful; for example, in Figure 3.O9.b a participant attempted to create a formal system of equations for solving the Magic Square problem as opposed to trying to develop an intuition for the puzzle – this made the problem harder to solve for them.

4.3 Solution Verification and Multiple Solutions

Once the participants finish solving the problem, either by declaring that they have finished or by giving up, we found regularities in whether they attempted to verify their solutions and whether they attempted to find multiple solutions (where appropriate).

(O10) When participants arrive to a potential solution there is a strong tendency to declare the problem solved. We assumed that participants would naturally seek to validate the results (e.g., check the constraints), yet not everybody did this, only 3 non-CS, 8 CS and 6 CP validated their solutions, in total that is only 28% of problem instances.

(O11) Despite **O10**, only in 2 instances participants announced that what they have found is the solution, did not validate it and in fact the solution was not correct.

(O12) Our scheduling problem happened to have two correct solutions (while it only asked for one). Here 9 out of the 11 participants who finished solving this problem found both solutions. How the participants decided to arrange the results seemed to have an effect on their ability to find more solutions.

(O13) In Figure 4.O13.a we can see an example of how the participant's representation of the solution failed to make it possible for them to see the multiple solutions, as the week tables are separate for each meeting. Whereas, when the representation combines all information bundled into a single table, the compact overview allows a precise insight into the options, as shown in Figure 4.O13.b.

(O14) Amongst our problems there is an additional solution type category. The Knapsack problem is an optimisation problem, which requires a confirmation as to whether one has found the best solution. 11 participants claimed to have finished solving the problem, while only 8 of these actually found the correct solution, and 1 of them was lucky as they did not validate their solution yet they did find the correct one.

(O15) In the Subset Sum problem we asked the participants to find as many solutions as possible. This prompted a majority of the participants to find multiple solutions, but not everyone managed to find all solutions. In general it is difficult for a participant to know when they found all solutions. This is comparable to knowing whether a solution to an optimisation problem is indeed optimal. None of the participants validated their solutions, yet 3 announced that they were finished with the solving process. All participants who attempted to solve the problem found some of the correct solutions, while 4 of the 6 found all solutions.

5 Limitations

Before we move on to interpret our observations, it is important to highlight the limitations implied by the methodological and study design choices. First, the study uses methodologies appropriate for an initial exploratory assessment of the research questions (i.e., a grounded theory approach). This choice, justified in Section 3, prioritises identifying phenomena over reliably assessing their prevalence. Although we provide proportions and counts which serve as an initial estimation of how reliable or broadly represented an observation is likely to be in the general populations under study, the numbers of participants and problems that we sampled are not enough to make strong claims about the frequency of their occurrence, let alone run statistical analysis. The much higher participant numbers required for this would be prohibitive timewise, or would have led to a coarser analysis, which would not have exposed the most interesting phenomena. Instead, our findings should be more generally interpreted as a "prove of existence" of these effects. This is in the same spirit as Nielsen's observation that a relatively small number of participants in a user study (5 in their case) can surface a majority of usability issues in an interface design [29, 20].

Another limitation of our study is also inherent to the limits of observation methodologies. Although we can be reasonably sure of the actions that participants carry out on paper or express verbally, it is not possible to assess with certainty the internal motivations of participants. We endeavor to keep observations factual; nevertheless, readers must be aware that some interpretation and error are unavoidable.

Practitioners should also be aware that the problems that our participants faced, although representative of a reasonable variety of existing constraint programming types, only partially match the variety and complexity of problems that actual users will face in their real lives. We believe that it is useful and productive to start at the lower end of complexity for an initial study of the topic. Nevertheless, further study of the challenges and barriers encountered by users when facing more complex constraint problems is granted.

When interpreting or applying our findings, practitioners should be aware that, although we aimed at a representative sample of participants with different levels of computing expertise, the majority of the participants had a high level of educational achievement. This means that populations with average levels of formal education are likely underrepresented. Despite this, we do not believe that this invalidates our results or makes them less valuable; if constraint modelling and programming are to become more widely used, it is likely that the broadening will take place first for non-CP computer scientists and then for professionals with high levels of education, before it can be further democratised. This is akin to how spreadsheet tools spread in the 1980's (see e.g., [27]).

Similarly, we agree that the presence of people with mathematical or scientific modeling expertise in the non-CS cohort could be a potential source of bias for the results. However, in our experience we cannot appreciate a difference between people with science and non-science backgrounds in the non-CS group. In fact, we suspect that it is the computational background of participants in the CS and CP groups is what makes the biggest difference. Whether this kind of modelling is more natural for scientists than non-scientists is an interesting question in itself which is, to the best of our knowledge, still unresolved in the scientific literature.

Finally, the interpretations, suggestions and lessons for practitioners that we offer in the next section, although evidence based, will still need to be validated after being put into practice through real modelling languages and tools. We see this study as an initial step to inform the design of a new generation of CP tools that are more aware of the human factor, not as a full characterisation of human behavior in relationship with constraint problems.

6 Discussion

In this section we interpret the observations from Section 4 and indicate how these might affect the design of user interfaces that solve constraint problems or communicate with users about constraint problems. There are two main categories of design decisions in which we imagine the findings being applied. The first category involves design of human-facing parts of existing constraint solving technologies such as constraint modelling languages, editors or graphical interfaces. The second is the design of system-provided automatic explanations or guidance that exposes the working of solvers to make it more intelligible to their users, perhaps to address issues of human trust and accountability in artificial intelligence systems. This is in line with the broader current push for intelligible, explainable or interpretable artificial intelligence (e.g., [1, 9]).

We group our discussion into three main topics, one about the importance of representations, one about people's mental models of the operations involved in the solving of constraint problems, and one about the larger context in which constraint solving takes place.

6.1 Representation, Representational Competence and Visualisation

Solving constraint problems can be characterised, at least in part, as a representational problem. When people recognise a world state that could benefit from calculating a solution based on constraints, the challenge is often for them to achieve a sufficiently accurate representation of the structure of the problem. Observations **O1**, **O2** and **O3** directly highlight some of the challenges of an efficient human-machine interface for constraint solving. The initial representation in which a problem appears to a person might lock their thinking to representations that might be suboptimal or even pernicious to the overall objective. We use a fictional (but plausible) example here to illustrate the different challenges. Consider a university administrator trying to schedule a series of rooms for exams of a group of students; the same student should not be required to be in two rooms at the same time, but different students take different combination of courses, whose associated exams should take place simultaneously and preferably in the same room. Our observations suggest that people will tend to adopt the most salient representation of the problem. In this case, it is likely that the administrator would naturally use a room-centric representation of the problem, since booking of rooms is their most visible required action. But a room-based representation can make it difficult for the administrator to express some of the student-centric constraints, and might also not be the best way to express the problem in a modelling language to obtain a solution efficiently. The ability of the problem holder to consider different representations might be a key element in the success of the modelling activity, yet we observed that non-CS people seem to have very little awareness of the importance of representations. Representational awareness and the related concepts of representational and meta-representational competence [23, 31, 8, 35] (i.e., the idea that people's ability to create different representations of a problem and translate between different representations is a fundamental skill for its understanding and solution) have been identified in other areas such as chemistry and physics education as key elements of expertise and solving skills [22].

The issue of representation is also likely to have an influence in the other direction as well. Once people get used to specific ways of representing problems (e.g., because they get used to a particular language or interface), the representation could become their default way of expressing this type of problems. If the representation is powerful and efficient for computation, this can be useful, but if the representation is unduly constrained or inappropriate it can prevent people from even recognizing a problem as such.

Our observations also bring a reminder that how solutions are represented is also important (**O13**). Ideally, constraint solving software should be able to present results in ways that are readily accessible to the problem holder. In fact, how the solutions are represented might have an influence on how they understand the solution space (see also Subsection 6.3) and whether they will try to iterate their modelling and choose solutions when multiple are available (this is discussed further in Subsection 6.2).

Overall, we believe that the design of representation specification tools and solution visualization are both areas of inquiry that can leverage significant benefit towards the goals stated in Section 2.

6.2 The Larger Context of Constraint Solving

Current tools for modelling and solving constraint problems (such as Conjure [2], SavileRow [30] and MiniZinc [28]) assume a programming workflow where the focus is on the sequence of a) model writing; b) model running/execution; c) output. However, some of our observations above suggest that there is benefit in considering the other surrounding activities in which the problem holder is involved. We have already indirectly mentioned two: users might need to switch representations (based on **O2** and **O3**), and users will need to interpret the solutions to see if they fit their needs (**O13**).

Our findings indirectly point to other parts of the process that might be also important. One is the need for validation of solutions (**O10**, **O11**); even if we can safely assume that the software will only provide solutions that are correct, it is possible that a human error occurred elsewhere (e.g., in the modelling, or when specifying to the solver whether the problem should be treated as a satisfaction problem – any solution will do – vs. an optimisation problem). An effective user interface can facilitate user validation that the solutions actually address the intended problem (not only the problem that the user managed to model). The need for validation is also recognised in previous work on using animation and model checking to allow users to gain confidence in their specifications [24].

In general, our observations of the process lead us to an understanding of the process of constraint problem solving that is less compartmentalised than what current interfaces assume. The iterative process of trying to solve a problem often drives people to improve their solution finding strategies (**O9**), but also to a better general understanding of the structure of the problem itself. For people, it might not be until they try to solve a problem that they start understanding whether the problem can have a single, more than one or even no solutions. Similarly, insights acquired by trying to manually solve the problem might lead to re-modeling (re-representations) of the problem, which could lead to improved efficiency. This follows findings in the domain of data analysis that shows that manual (and often tedious) specification of visualizations can enhance understanding of data [26]. Additionally, we have very little doubt that the general process of constraint problem solving with human intervention most often requires multiple iterations of modelling, execution and verification (note, however, that our study did not offer sufficient context to expose this larger loop).

6.3 Human Thinking about Constraint Solving

At first sight it might seem gratuitous to study how people solve constraint problems manually. After all, modern computers running modern solvers are more accurate and orders of magnitude faster than humans at this task. However, we think that there are several reasons why our observations can be beneficial for the design of future systems.

One such reason is that the user's expectations matter. For example, we observed that many participants in the non-CS and CS groups expected problems to have a "happy", straightforward, or analytical solution (**O4**). Users which assume that every problem has this kind of solution might expect fewer solving steps, even when the state space is large and even an efficient solver can take seconds or even hours to find a solution, let alone all solutions or demonstrate that there are none. It is not clear where this assumption comes from, but it might be from how people often encounter constraint problems as puzzles or exam questions, which are usually designed precisely to avoid the trial-error-backtracking process that can be inevitable in many problems.

Through the study of how people attempt to solve these problems we have also observed that many non-CP experts have very little understanding of the size of the state space and the task of solving some of these problems, even when they have some background in CS. This presents a problem for systems integrating constraint solvers because interfaces will often not be instantaneous, which is the current expectation for most applications. We see two possible ways to ameliorate this problem. The system could explain and communicate with users why a solution can not be arrived at very fast. Beyond this, systems could educate the user to recognise the type of modeling structures that cause the delay in the first place, and perhaps suggest alternatives that preserve the meaning of the model but are faster to solve. A very sophisticated system that enables this kind of human-CP solver dialogue likely requires maintaining in the solver some kind of user model that incorporates knowledge about regularities of the kind that we found in our study (e.g., preference for modelling from a certain point of view – **O3**).

7 Related Work

Previous work considers the interaction of constraint programming and its users in the context of teaching, visualisation and explainable AI.

A typical way of teaching constraint programming is beginning with its theoretical foundations, which often takes a mathematical approach. For example the "Essentials of Constraint Programming" book [14] contains chapters on logic, Boolean algebra, linear polynomial equations and non-linear equations. Similarly, the "Principles of Constraint Programming" book [3] defines constraint programming as a linear equation solver and a unification algorithm. On the other hand, in his invited talk at CP 2014, Prosser explains his experiences with teaching constraint programming [33]. According to this talk[1], getting students to solve problems as soon as possible increases their engagement as opposed to starting with a more theoretical background. Chan et al. [7] presents a unique approach to teaching CP in a MOOC (Massive Online Open Course) setting. They use a Fable-Based learning approach and present the topics in the course using a coherent story plot and through problem solving. Both Prosser and Chan et al. describe benefits gained from listening to the learners and explain how they improved their teaching style upon feedback.

There are several examples of visualisation tools for the *solving* process, we describe a small selection here. Bauer et al. [4] presented an integrated development environment (IDE) for constraint programming. The IDE provides a visual debugger which displays the search tree that is explored by the constraint solver. The debugger is solver-independent, with minor modifications it can support any solver. However, their system only focuses on visualising the solving process and not modelling. Recently Goodwin et al. [18] described a user-centred design process for tools that visualise the solving process, building on earlier work by Shishmarev et al. [36]. From an Information Visualisation perspective, Goodwin et al. [18] looked at how different visualisations could be useful in the process of profiling constraint models. This allows users of constraint programming to refine their models or test different parameters.

Constraint programming lends itself to automatically creating explanations for unsatisfiable problems, as well as explanations for how certain inferences or decisions are made. This is mainly due to the model-based nature of CP: typically users write a declarative model when applying CP technology and this explicitly captures the requirements of the task at hand. Sqalli et al. [38] produce explanations for inference-based CSPs, Bogaerts et al. [6] extend this work to multiple steps in the context of logic puzzles, and Espasa et al. [10] show human-like solving steps in a variety of pen and paper puzzles.

8 Conclusion

This paper presents a user-centric qualitative study, aiming to understand how people approach constraint modelling and solving. The study is done on three groups of participants (30 in total), and hence we are able to explore regularities within and across distinct groups of people. To the best of our knowledge this paper presents the first user-study of its kind: recording participants model and solve constraint problems with maximum freedom in their approach, analysing 32+ hours of video recordings, coding each recording to characterise it, and using grounded theory to offer an analysis of the codes we produce.

Some directions for future work are related to the limitations in our study. Further experiments are necessary to statistically quantify the preponderance of the different phenomena that we observed (our study did not have sufficient numbers for it, since it was designed for identifying the phenomena instead), to ascertain possible sub-populations of interest in the non-CS cohort (e.g., see whether people with and without science backgrounds show differences in how they think about constraint problem solving – this could have biased our results, since three participants in our non-CS group had science backgrounds) and to generalize to people without formal tertiary education (including children). More advanced CP-specific aspects during modelling (like the use of global constraints, the effect of implied constraints and symmetry/dominance breaking constraints) and during the solving process

[1] Available online: `http://www.dcs.gla.ac.uk/~pat/presentations/CP2014.pptx`

(like search heuristics and clause learning) that affect the efficiency of solving need to be studied further as well. Another direction of future work is the development of prototype user interface systems that build on top of our findings and a thorough evaluation of such user interfaces.

References

1 Amina Adadi and Mohammed Berrada. Peeking inside the black-box: A survey on explainable artificial intelligence (XAI). *IEEE Access*, 6:52138–52160, 2018. doi:10.1109/ACCESS.2018.2870052.

2 Özgür Akgün, Ian Miguel, Christopher Jefferson, Alan M. Frisch, and Brahim Hnich. Extensible automated constraint modelling. In *AAAI 2011*. AAAI Press, 2011. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3687.

3 Krzysztof R. Apt. *Principles of constraint programming*. Cambridge University Press, 2003.

4 Andreas Bauer, Viorica Botea, Mark Brown, Matt Gray, Daniel Harabor, and John K. Slaney. An integrated modelling, debugging, and visualisation environment for G12. In *CP 2010*. Springer, 2010. doi:10.1007/978-3-642-15396-9_42.

5 Amine Benamrane, Imade Benelallam, and El-Houssine Bouyakhf. Constraint programming based techniques for medical resources optimization: medical internships planning. *J. Ambient Intell. Humaniz. Comput.*, 11(9):3801–3810, 2020. doi:10.1007/s12652-019-01587-6.

6 Bart Bogaerts, Emilio Gamba, Jens Claes, and Tias Guns. Step-wise explanations of constraint satisfaction problems. In *ECAI 2020*. IOS Press, 2020. doi:10.3233/FAIA200149.

7 Mavis Chan, Cecilia Chun, Holly Fung, Jimmy H. M. Lee, and Peter J. Stuckey. Teaching constraint programming using fable-based learning. In *AAAI*. AAAI Press, 2020. URL: https://aaai.org/ojs/index.php/AAAI/article/view/7059.

8 Andrea A. diSessa. Metarepresentation: Native Competence and Targets for Instruction. *Cognition and Instruction*, 22(3):293–331, 2004. doi:10.1207/s1532690xci2203_2.

9 Mengnan Du, Ninghao Liu, and Xia Hu. Techniques for interpretable machine learning. *Commun. ACM*, 63(1):68–77, 2020. doi:10.1145/3359786.

10 Joan Espasa, Ian P. Gent, Ruth Hoffmann, Christopher Jefferson, and Alice M. Lynch. Using small muses to explain how to solve pen and paper puzzles. *CoRR*, abs/2104.15040, 2021. arXiv:2104.15040.

11 Eugene C. Freuder. In pursuit of the holy grail. *Constraints An Int. J.*, 2(1):57–61, 1997. doi:10.1023/A:1009749006768.

12 Eugene C. Freuder. Progress towards the holy grail. *Constraints An Int. J.*, 23(2):158–171, 2018. doi:10.1007/s10601-017-9275-0.

13 Eugene C. Freuder and Mark Wallace. Constraint technology and the commercial world (interview). *IEEE Intell. Syst.*, 15(1):20–23, 2000. doi:10.1109/MIS.2000.820324.

14 Thom W. Frühwirth and Slim Abdennadher. *Essentials of constraint programming*. COGTECH. Springer, 2003. URL: http://www.springer.com/computer/swe/book/978-3-540-67623-2.

15 Cristian Galleguillos, Zeynep Kiziltan, and Ricardo Soto. A job dispatcher for large and heterogeneous HPC systems running modern applications. In *CP 2021*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CP.2021.26.

16 James H. Gerlach and Feng-Yang Kuo. Understanding human-computer interaction for information systems design. *MIS Q.*, 15(4):527–549, 1991. URL: http://misq.org/understanding-human-computer-interaction-for-information-systems-design.html.

17 Barney G Glaser and Anselm L Strauss. *The discovery of grounded theory: Strategies for qualitative research*. Routledge, 2017.

18 Sarah Goodwin, Christopher Mears, Tim Dwyer, Maria Garcia de la Banda, Guido Tack, and Mark Wallace. What do constraint programming users want to see? exploring the role of visualisation in profiling of models and search. *IEEE Trans. Vis. Comput. Graph.*, 23(1):281–290, 2017. doi:10.1109/TVCG.2016.2598545.

19 Francis Heylighen. Formulating the Problem of Problem-Formulation. In *Cybernetics and Systems '88*, pages 949–957. Kluwer Academic Publishers, Dordrecht, 1988.

20 Wonil Hwang and Gavriel Salvendy. Number of people required for usability evaluation: the 10+/-2 rule. *Commun. ACM*, 53(5):130–133, 2010. doi:10.1145/1735223.1735255.

21 Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P Gent. CSPLib: A problem library for constraints, 1999. URL: http://www.csplib.org.

22 Antje Kohnle and Gina Passante. Characterizing representational learning: A combined simulation and tutorial on perturbation theory. *Physical Review Physics Education Research*, 13(2):020131, 2017. doi:10.1103/PhysRevPhysEducRes.13.020131.

23 Robert Kozma and Joel Russell. Students Becoming Chemists: Developing Representational Competence. In *Visualization in Science Education*, Models and Modeling in Science Education, pages 121–145. Springer Netherlands, Dordrecht, 2005. doi:10.1007/1-4020-3613-2_8.

24 Michael Leuschel and Michael Butler. Prob: A model checker for b. In *International symposium of formal methods europe*, pages 855–874. Springer, 2003.

25 Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia Medica*, 22(3):276–282, 2012. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3900052/.

26 Gonzalo Gabriel Méndez, Uta Hinrichs, and Miguel A. Nacenta. Bottom-up vs. top-down: Trade-offs in efficiency, understanding, freedom and creativity with infovis tools. In *CHI 2017*. ACM, 2017. doi:10.1145/3025453.3025942.

27 Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. MIT press, 1993. URL: https://books.google.co.uk/books?hl=en&lr=&id=0drDRT370eoC&oi=fnd&pg=PR11&dq=nardi+small+matter+of+programming&ots=eFmV2hPujA&sig=ddWW4jgU1jebzDZpk7plDRCtcoU.

28 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *CP 2007*. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

29 Jakob Nielsen. Estimating the number of subjects needed for a thinking aloud test. *Int. J. Hum. Comput. Stud.*, 41(3):385–397, 1994. doi:10.1006/ijhc.1994.1065.

30 Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in savile row. *Artif. Intell.*, 251:35–61, 2017. doi:10.1016/j.artint.2017.07.001.

31 Orit Parnafes and Andrea A. diSessa. Relations between types of reasoning and computational representations. *Int. J. Comput. Math. Learn.*, 9(3):251–280, 2004. doi:10.1007/s10758-004-3794-7.

32 Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011. URL: http://dl.acm.org/citation.cfm?id=2078195.

33 Patrick Prosser. Teaching constraint programming. In *CP 2014*. Springer, 2014. doi:10.1007/978-3-319-10428-7_2.

34 Bochra Rabbouch, Foued Saâdaoui, and Rafaa Mraihi. Constraint programming based algorithm for solving large-scale vehicle routing problems. In *HAIS 2019*. Springer, 2019. doi:10.1007/978-3-030-29859-3_45.

35 S I Robertson. *Problem Solving*. Psychology Press, 2001.

36 Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. Visual search tree profiling. *Constraints An Int. J.*, 21(1):77–94, 2016. doi:10.1007/s10601-015-9202-1.

37 Herbert A Simon and John R Hayes. The understanding process: Problem isomorphs. *Cognitive psychology*, 8(2):165–190, 1976.

38 Mohammed H. Sqalli and Eugene C. Freuder. Inference-based constraint satisfaction supports explanation. In *AAAI/IAAI 1996*. AAAI Press / The MIT Press, 1996. URL: `http://www.aaai.org/Library/AAAI/1996/aaai96-048.php`.

39 Maarten Van Someren, Yvonne F. Barnard, and J. Sandberg. The think aloud method: A practical approach to modelling cognitive processes. *London: AcademicPress*, 11, 1994.

40 Xu Zhu, Miguel A. Nacenta, Özgür Akgün, and Peter Nightingale. How people visually represent discrete constraint problems. *IEEE Trans. Vis. Comput. Graph.*, 26(8):2603–2619, 2020. `doi:10.1109/TVCG.2019.2895085`.

Learning Constraint Programming Models from Data Using Generate-And-Aggregate

Mohit Kumar ✉
KU Leuven, Belgium

Samuel Kolb ✉
KU Leuven, Belgium

Tias Guns ✉
KU Leuven, Belgium

Abstract

Constraint programming (CP) is used widely for solving real-world problems. However, designing these models require substantial expertise. In this paper, we tackle this problem by synthesizing models automatically from past solutions. We introduce COUNT-CP, which uses simple grammars and a generate-and-aggregate approach to learn expressive first-order constraints typically used in CP as well as their parameters from data. The learned constraints generalize across instances over different sizes and can be used to solve unseen instances – e.g., learning constraints from a 4 × 4 Sudoku to solve a 9 × 9 Sudoku or learning nurse staffing requirements across hospitals. COUNT-CP is implemented using the CPMpy constraint programming and modelling environment to produce constraints with nested mathematical expressions. The method is empirically evaluated on a set of suitable benchmark problems and shows to learn accurate and compact models quickly.

2012 ACM Subject Classification Applied computing → Operations research

Keywords and phrases Constraint Learning, Constraint Programming, Model Synthesis

Digital Object Identifier 10.4230/LIPIcs.CP.2022.29

Supplementary Material *Software (Source Code)*: `https://github.com/ML-KULeuven/COUNT-CP`

Funding This project was partially funded by the Research Foundation – Flanders (FWO) project G0G3220N and FWO-S007318N, as well as the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant No. 101002802: CHAT-Opt, and 694980: SYNTH). Samuel Kolb is supported by the "Agentschap Innoveren & Ondernemen" (VLAIO) as part of the innovation mandate HBC.2021.0246.

1 Introduction

Constraints play an important role in modelling many real-world decision problems. They are used widely in fields like cryptography [13, 15], complexity theory [1] and automatic theorem proving [14]. However, identifying the constraints of a problem and encoding them into a mathematical model requires both domain knowledge and modelling expertise. This non-trivial task is often the major bottleneck for the widespread application of constraint-based methods and solvers.

Consider, for instance, the case of scheduling nurses in a hospital, where the aim is to create a schedule for nurses every week. Modeling this problem requires domain knowledge to identify relevant constraints, such as, *every shift requires at least three nurses* or *nurses may work at most five days a week*. Next, these constraints have to be encoded as a mathematical model, e.g., a Constraint Satisfaction Problem (CSP). The skills required to achieve both these steps makes powerful techniques for efficiently solving such problems inaccessible to people without a mathematical or computer science background.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 29; pp. 29:1–29:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Constraint learning approaches aim to overcome this issue by instead learning constraint models from past solutions [19]. In the example of nurse scheduling, this means learning the constraint model from manually created past schedules. By automating the modeling step, constraint learning makes constraint solving techniques more accessible and makes the modeling process faster and cheaper. In the CP community there are a number of existing approaches to learn constraints from solutions [10, 2, 4] and – in some cases – non-solutions [17, 18]. A popular approach to learning constraint is the so called generate-and-test approach [21]. The idea behind generate-and-test is to generate candidate constraints and apply them to various subsets of the decision variables and test whether the constraint holds in the training data.

Most of these approaches learn constraints at the level that a constraint solver accepts: individual constraints, such as predicates with a fixed number of arguments. By listing all possible predicates and their signature, different predicate/variable combinations can be generated and tested. Learning more expressive constraints, however, often requires generating prohibitively large combinations of predicates and makes constraint learning very time-consuming. As a result several approaches design their constraint space instead as a flat catalog of more expressive candidate constraints, learning global constraints [2] or relational spreadsheet formulas [10]. By modeling constraints using an expressive, richer language, rather than acquiring individual lower-level constraints, these approaches are able to synthesize high quality models quickly. The key limitations are that only constraints from these catalogs can be learned and that parameters can only be inferred using constraint-specific parameter inference methods.

A recent approach (COUNT-OR [11]) to learning constraints for OR models, such as nurse scheduling problems, offered an alternative: Using a simple grammar of aggregation operators, different aggregation expressions are generated and applied to various slices of matrices and in general tensors of decision variables. By simply computing the lower and upper bounds of these expressions across all training examples, the method automatically identifies relevant parameters from data and learns constraints quickly.

We built on this approach to design a constraint learner (COUNT-CP) that applies the ideas of bounded expressions to learn CP constraints. First, we observe that constraint models over finite domain integers usually consist of Boolean expressions and numeric expressions with a comparison (e.g., `x = y` and `x <= y`). Because Boolean expressions in this context can be seen as a special case of numeric expressions that are equal to 1, we can use suitable bounded numeric expressions `lb <= expr <= ub` to express common types of constraints (e.g., `abs(x-y) <= 0` and `x-y <= 0`). Second, we observe that first-order constraints such as *each nurse works at most 5 days a week* or global constraints such as `alldifferent` can be decomposed into multiple bounded-expression constraints.

Based on these observations, we suggest to learn bounded-expressions using a COUNT-OR style approach which offers an obvious mechanism to infer the constants. COUNT-CP uses a simple grammar to generate suitable nested mathematical expressions and computes their lower and upper bounds. However, to produce expressive, first-order constraints, the learned bounded-expression constraints are then grouped together over structured sets of variables using simple grammars of `foreach` statements. This step also serves as an inductive bias in selecting which constraints should make up the final learned model. As a result, COUNT-CP is able to assemble first-order constraints, such as *each nurse works at most 5 days a week* and global constraints such as `alldifferent`. COUNT-CP allows users to provide background knowledge in the form of sets of variables that the user considers related – e.g., connected edges in a graph or shared skill levels of nurses – and adds these sets to the grammar used for grouping constraints.

Our developments have been inspired by the PTHG-21 Holy Grail Challenge, which contains variable-sized problem instances over the integer domain, and where a preliminary version of our approach was the winning (and only) entry. In principle, first-order constraints are independent of the instance size and can be used to solve different instances of unseen sizes. However, the numeric constants fitted by approaches such as COUNT-OR may be instance size dependent and would not apply to unseen instances. To resolve this issue, we propose to fit a symbolic bound expression across training instances, using generic problem features as well as semantic constants provided by a domain expert – e.g., minimal staffing requirements of a hospital.

To summarize, the key contributions of this paper are:

- Learning first-order bounded-expression constraints, which are expressive enough to capture many complicated constraints *and* can be learned using a simple and fast generate-and-aggregate procedure.
- Defining the language bias for constraints using simple and simple-to-extend grammars that are combined to learn intricate constraints.
- Replacing constants in the learned constraints by symbolic expressions, which allows learned model to generalise to different and unseen problem sizes.
- Allowing users to provide background knowledge using a simple interaction protocol: sets of related variables and semantic instance-level constants.
- Providing an effective strategy for removing redundant constraints to improve the interpretability and speed of learned models.

This paper is structured as follows: First, we review related work on constraint learning (Section 2). Second, we present our constraint learning approach (COUNT-CP, Section 3) by discussing its links to COUNT-OR, how it learns propositional constraints, first-order constraints and how we filter out constraints to produce compact models. Third, we empirically evaluate our approach on a set of suitable benchmark problems (Section 4). Fourth, and finally we summarize our conclusions (Section 5).

2 Related Work

Learning constraints from a given set of feasible examples has a long history. The first algorithm in this regard was given by Valiant [21], back in 1984. Given a set of feasible examples, this algorithm learns Boolean formulas consistent with the given examples. To do so, it enumerates all possible formulas upto a pre-defined complexity and keeps only those which are satisfied by all feasible examples. This is essentially a generate-and-test approach, where the algorithm generates all possible constraints and then tests whether they hold on the given dataset. This approach was later extended to first order logic under the banner of inductive logic programming [8]. Although important, these early works are limited to Boolean variables and logical formulas.

More recent works, like the series of work by Bessiere et al [4, 5, 6] extend these approaches to integer variables. For instance, Conacq [4] learns constraints, typically using the basic comparison relations $(=, <, \leq, \geq, >, \neq)$. The relations considered is called the *bias*. It basically searches for such constraints over every compatible subset of variables (called *scope*, e.g. all pairs) and defines a lattice structure of the comparison relations, based on the generalisation/specialisation relation between them. Feasible examples are used to remove relations from the lattice. Infeasible examples only say that there has to exist one constraint over all possible variable combinations that is violated; which is expressed as a meta constraint. The authors use the concept of *convergence* to denote if a lattice for a pair of variables only contains a single relation; if not, the default action is to take the most specific relation as constraints, e.g. in case of $\geq, \leq$ and $=$, $=$ is taken. The concept of the lattice comes from the version space algorithm where a lattice is defined over the whole program space. When considering a separate lattice for every variable pair, arguably its main purpose may be to identify which relations are redundant, e.g. subsumed by others. None of these works can learn the bounds in the data, let alone symbolic bounds. This work was later extended to learn generalized constraints [3], however, the assumption here was that the user knows which variables are supposed to be grouped together. It was further extended to detect the groups automatically in [7].

Another well known approach is ModelSeeker [2], which is also a generate-and-test approach; it does not just consider basic comparison relations, but a subset of all global constraints in the global constraint catalog. Furthermore, it does not search over all subsets of variables, but instead has candidate *generator* expressions that uses the structure of the decision variables (e.g. a list or matrix) to group the variables into meaningful subsets (e.g. per row, per column, all pairs). It then generates and tests which constraints are satisfied in each of the groups, over all positive instances. The use of "generators" matches the programming style of "foreach s in ...: constraint(s)" that CP modellers often use. If the constraint has additional parameters, then these need to be inferred separately using custom rules for every constraint. However, ModelSeeker requires the constraint catalog to be provided beforehand and does not learn symbolic parameters/bounds.

Finally the CPS [12] approach uses inductive logic programming (ILP) to find logic programming rules of the form $\textit{condition} \implies \textit{constraint}$. The condition can be seen as a *generator* too, for the learned logical rules have to be flattened into low level constraints for every possible substitution of the condition.

Our proposed approach does not go as far as ModelSeeker in considering a bias of a wide range of generic to very specialized global constraints. We do go beyond simple comparison relations between pairs of variables, by considering a comparison relation on a mathematical expression over variables. We use a grammar to generate the possible mathematical expressions. This captures the basic comparison relations, but also captures linear (unweighted) expressions and the use of constants such as in $x + y \geq 2$. This approach of mathematical expressions turned into constraints by identifying bounds on them, also generalizes well to the use of *generator* expressions.

A recent approach used for learning hard constraints given a set of feasible and infeasible examples is to encode the learning problem itself as a mathematical model [10, 17]. For instance, Pawlak et al. [17] learn constraints with linear, quadratic and trigonometric terms by encoding the learning as a MILP. The hard constraints of this MILP ensure that each example is correctly classified by the learnt model, while the objective tries to minimise the complexity of the learnt model. This ensures that the learnt model is concise and easily readable. This work was later extended to work with positive only training instances [16], where the idea is to fit a Gaussian mixture model on the given set of feasible points and use this model to sample infeasible points, the learning strategy then uses both feasible and the sampled infeasible instances to learn a model. The main drawback of these methods is that they do not learn symbolic expressions, and thus can not generalise to unseen problem sizes.

3 Learning constraints using COUNT-CP

3.1 Background on COUNT-OR

Our approach for learning constraints is inspired by COUNT-OR [11], a constraint learning approach for acquiring personnel rostering constraints from example schedules. Instead of generating and testing a set of possible constraints, COUNT-OR instead generates a set of *expressions* that capture useful quantities in the data by applying aggregates to various slices of matrices and tensors. We use the term tensor for a multi-dimensional matrix, basically a matrix is a 2D tensor and a tensor can have more than 2 dimensions.

▶ **Example 1.** Consider a Boolean matrix of nurses (rows) and days (columns) that encodes whether a given nurse works on a given day (1) or not (0). Summing over values in a row of such a weekly schedule expresses the number of working days for a nurse.

By comparing the values of these expressions across different example schedules, COUNT-OR finds upper and lower bounds for every expression. Together, these bounds and expressions can be translated to constraints. For example, if the number of working days of nurse i in the example schedules is always between 0 and 5, COUNT-OR produces a constraint

```
0 <= sum(X[i, :]) <= 5
```

COUNT-OR also compares values across tensor slices, e.g., comparing working days for different nurses, to find generalized constraints, such as

```
foreach i: 0 <= sum(X[i, :]) <= 6
```

These generalized constraints can then be applied to schedules with different numbers of nurses.

We will adopt a similar strategy for learning CP models, however, we use specialized grammars to generating different types of expressions and to find slices that can be used for generalized constraints. Given the conceptual similarity to COUNT-OR, we call our approach COUNT-CP. First, we will explain how COUNT-CP generates *propositional* expressions for fixed problem sizes, that is, individual constraints that involve specific subsets of variables (their scope) and a relation between those variables. Second, we describe how to group these propositional expressions to find *generalized* constraints that can also be carried over to unknown problem sizes.

3.2 Learning propositional constraints

In this work, we consider the case of learning constraints of the following form:

```
lb <= expr <= ub
```

where `expr` is a mathematical expression over variables, such as `X[i] + X[j]`, or an aggregate expression over a group of variables, such as `sum(X[:])`. To learn these constraints, we defined a grammar that captures expressions frequently occurring in CP problems, and a mechanism to find suitable lower and upper bounds.

Our approach is different from current constraint learning approach in that our *bias*, the set of possible constraints that can be learned, is not determined by a fixed set of constraints (whose parameters might have to be inferred later). Rather, our bias consists of mathematical expressions on the one hand, and bound-constraints on these expressions on the other hand.

Expression grammar

To construct our grammar, we look at unary, binary and aggregate expressions that can be expressed in CP modelling languages such as MiniZinc and CPMpy. We consider the unary *identity* expression, the binary expressions *addition*, *subtraction* and *absolute difference*, and the aggregate *sum* expression.

Observe how this grammar does not include the “traditional” constraint biases `x != y`, `x <= y`, `x < y`, etc. The reason is that we have constraints that subsume those, namely `abs(x - y) >= 1`, `x - y <= 0` and `x - y <= -1`, respectively. Hence, our constraint bias

– inequalities over the expression grammar – can learn those traditional constraints. But it can also learn other constraints, like `abs(x - y) > 2`, as the bounds are automatically determined and not sequentially tested based on a predetermined list.

One of the few unary/binary constraints it can not learn is `x != c`, for some constant `c` which lies between (exclusive) the lower and upperbound of `x`. We believe it will be very rare that a constraint model intentionally excludes one individual value, without that value being specified as the "input data" (more on this later). Hence, it is not part of our bias.

The bias also does not include n-ary global constraints such as `alldifferent()` or `increasing()`, however, these have decompositions into binary constraints, meaning that we can learn the decomposed versions. In Subsection 3.3 we will see how to group these decomposed constraints into generalized constraints that recreate such global constraints.

Also currently not included are tertiary constraints, such as `x + y = z` or, equivalently, bounds on `x + y - z` for arbitrary triples. We leave it open whether these constructs are commonly used, and how to best manage the large number of candidates.

This simple grammar already allowed us to learn a varied set of constraints. However, for more complicated problems, the grammar can trivially be extended with additional unary (e.g., `X[i]*X[i]`, `mod(X[i], 2)`), binary or n-ary operators. This increased expressiveness would, however, incur an additional computational cost during learning.

Learning algorithm

Our learning algorithm learns from a set of **positive** examples T, i.e., given true solutions. For the propositional learner, which learns individual constraints on specific subsets of variables, we expect all examples to have the same size and hence the same number of decision variables. Every positive example consists of a set of tensors which contain assignments to a given set of decision variables. For the sake of exposition we limit our discussion in this paper to at most two dimensional tensors (lists and matrices) and use lists for illustration whenever possible.

▶ **Example 2.** Consider the problem of graph coloring: Given a list of nodes, assign a color to every node such that nodes that share an edge are assigned different colors. This problem can be encoded using a list of n integer decision variables X – one per node – whose values corresponds to colors. Positive examples would be assignments to X that satisfy the graph coloring constraints. For an instance with 5 nodes and edges $1-2, 1-3, 2-4, 3-5$, examples could be assignments t_1 = `[1, 2, 3, 1, 1]` and t_2 `[1, 2, 2, 1, 1]`.

To turn an expression into a constraint, COUNT-CP first generates all expressions in its grammar and computes their result for different decision variables. Unary expressions are simply applied to every single decision variable `X[i]`. Binary expressions are applied to every possible pair of decision variables `(X[i], X[j])`. In general, for n-ary expressions, all tuples of n variables are generated. We use lexicographical ordering to ensure pairs are not enumerated multiple times to avoid redundant constraints. For asymmetric expressions this optimization cannot be used as the position of variables are important. However, substraction is a special case because it holds that `lb <= a - b <= ub` can be rewritten as `-ub <= b - a <= -lb` which simply results in different bounds being learned.

Aggregates are applied to logical groups of variables, such as individual rows and columns of matrices or user provided groupings (see partitions in Subsection 3.3). For every expression e and set of variables V, COUNT-CP then computes the minimum and maximum result across all training examples. We represent these local lower- and upper-bounds as tuples $\langle e, V, \texttt{lb}, \texttt{ub}\rangle$, where – denoting the values of variables V in example t as V^t:

$$\texttt{lb} = \texttt{min}\{e(V^t) \mid t \in T\} \qquad \texttt{ub} = \texttt{max}\{e(V^t) \mid t \in T\} \tag{1}$$

It follows that, by design, the constraints learned by COUNT-CP are always satisfied by all training examples. In a sense, our approach learns the convex hull of all mathematical expressions that can be expressed by the grammar.

▶ **Example 3.** Let us apply this approach to the graph coloring example. For the binary expression `abs(X[i] - X[j])`, COUNT-CP would compute the result of the expression for every pair in every example and then compute the bounds for every pair across the examples:

Pair	t_1	t_2	`lb`	`ub`
`X[1], X[2]`	1	1	1	1
`X[1], X[3]`	2	1	1	2
`X[1], X[4]`	0	0	0	0
`X[1], X[5]`	0	0	0	0
`X[2], X[3]`	1	0	0	1
`X[2], X[4]`	1	1	1	1
`X[2], X[5]`	1	1	1	1
`X[3], X[4]`	2	1	1	2
`X[3], X[5]`	2	1	1	2
`X[4], X[5]`	0	0	0	0

For all pairs of nodes with an edge between them, COUNT-CP will learn a constraint `abs(X[i] - X[j]) >= 1`, i.e., the nodes must have different colors.

3.3 Learning first-order constraints

Until now we have focused on learning propositional constraints for individual instances of a problem type. These local constraints can capture constraints over specific subsets of variables, however, these constraints cannot be used to find solutions for instances of different sizes (and hence different numbers of variables) and are prone to overfitting the training examples. Our goal is to address these shortcomings by learning *first-order* constraints that are independent of the instance size. That is, constraints of the form `foreach V in ...: lb <= expr(V) <= ub` This will allow us to learn constraints from, e.g., a 4 × 4 Sudoku and use these constraints to solve a 9 × 9 Sudoku. Additionally, we can find constraints, e.g., that the number of working days of nurses is at most 5, by generalizing across different nurses in a single example, even if some nurses always worked fewer days in the training examples.

Grouping constraints

The propositional constraint learning approach can learn constraints such as `alldifferent` by learning individual constraints `abs(X[i] - X[j]) >= 1` between each pair of decision variables. However, these local pairwise constraints are *hardcoded* for individual pairs of variables, and will not generalize to instances of different sizes that, for example, have more or less decision variables.

To overcome this limitation and learn constraints that are independent of the problem size, we find *index groups*, groups of decision variables or pairs of variables, that share a constraint. The concept is akin to the concept of generator expressions in ModelSeeker [2].

In this setting, for example, `alldifferent` can be encoded as:

```
foreach pairs (x[i], x[j]) in X:
    abs(x[i] - x[j]) >= 1
```

For a given expression e, e.g., absolute difference, and the set of learned local constraints C COUNT-CP uses a *sequence grammar* to generate *sequences*, i.e., sets $\mathbf{V}$ of decision variables to group over. For example, the sequence `all pairs` generates all pairs of decision variables. Next, COUNT-CP aggregates all the lower- and upper-bounds that had been found for local constraints to obtain a grouped or *first order* constraint $\langle e, \mathbf{V}, \texttt{lb}, \texttt{ub} \rangle$, where:

$$\texttt{lb} = \texttt{min}\{l \mid V \in \mathbf{V} \wedge \langle e, V, l, u\rangle \in C\} \tag{2}$$

$$\texttt{ub} = \texttt{max}\{u \mid V \in \mathbf{V} \wedge \langle e, V, l, u\rangle \in C\} \tag{3}$$

Our COUNT-CP implementation includes the following sequences: `all`) all individual unary variables; `all pairs`) all pairs of variables; and `full`) a singleton set with all variables. These sequences are used for unary, binary and aggregate expressions, respectively. The implementation can easily be extended with additional sequences, such as, variables with even indices, sequential pairs of variables, etc.

Partitioning groups

An `alldifferent` constraint will not usually be applied to *all* possible variables. A common pattern, instead, is that the variables are partitioned into groups with an `alldifferent` over each group. This pattern is used both by COUNT-OR, as well as the constraint learning system ModelSeeker [2]. For example, consider the example of Sudoku where the decision variables are arranged in a matrix. In this case the variables can be partitioned into rows, columns or blocks and within each partition the variables will be `alldifferent`.

COUNT-CP follows this pattern, too, and first considers different ways to partition the decision variables before searching for sequences and corresponding bounds within each partition. By default, COUNT-CP considers arbitrary slices of tensors as partitions. In the case of matrices this would be rows, columns as well as the entire matrix. Additionally, COUNT-CP allows users to provide custom partitions. Custom partitions are a powerful way for users to interact with the system and provide high-level background knowledge, such as blocks for Sudoku or edges of a graph coloring problem.

For an expression e, a partition P and a sequence s, COUNT-CP iterates over all partitions $p \in P$ and generates sets of indices by applying the sequence to obtain sets of indices $\mathbf{V}_p = s(p)$. Using the tuples $\langle e, \mathbf{V}, \texttt{lb}, \texttt{ub}\rangle$ found in the grouping step, it aggregates the bounds across partitions to obtain tuples $\langle e, P, s, \texttt{lb}, \texttt{ub}\rangle$, where:

$$\texttt{lb} = \texttt{min}\{l \mid p \in P \wedge \langle e, s(p), l, u\rangle \in C\} \tag{4}$$

$$\texttt{ub} = \texttt{max}\{u \mid p \in P \wedge \langle e, s(p), l, u\rangle \in C\} \tag{5}$$

The combination of partitions, sequences and bounded expressions allow us to learn complicated sets of first-order constraints, e.g., that the variables in columns are `alldifferent` or that each of the sums over rows never exceed an upper bound.

▶ **Example 4.** Building on the graph coloring problem introduced above, a user can provide a custom partition P_{edges} for each instance that corresponds to the edge E of the graph used in an instance: $P_{\text{edges}} = \{\{X[i], X[j]\} \mid (X[i], X[j]) \in E\}$.

```
foreach group in P_edges:
    foreach (X[i], X[j]) in pairs(group):
        abs(X[i] - X[j]) >= 1
```

3.4 Symbolic expressions for bounds

So far, we talked about grouping constraints using partitions and sequence generators. In some cases, e.g., the column-wise all different, this grouping step is enough to learn first-order constraints that can be applied to instances of any size. However, in some cases the lower- and upper-bounds depend on the particular instance.

▶ **Example 5.** Reconsider the nurse scheduling example. When learning across schedules from different hospitals, the minimal staffing requirement, i.e., how many nurses have to work each day might differ and are an instance (hospital) dependent constant. Simply learning the smallest minimal staffing requirement across all hospitals will produce poor results.

To address this issue, COUNT-CP attempts to express bounds using symbolic expressions. These symbolic expressions can use computed features, e.g., the number of rows and columns, or custom features that the user provides for every instance, e.g., the minimal staffing requirements for hospitals. To keep the discussion clear, we focus on finding a symbolic expression for the upper-bound of a single first-order constraint $\langle e, P, s, \texttt{lb}_i, \texttt{ub}_i \rangle$ across instances i. The steps are repeated for each constraint and are analogous for lower-bounds.

COUNT-CP aims to find a simple, univariate symbolic expression of the form $f + b$, where f is a computed feature, a custom feature or 0, and b is a fixed offset. Given m candidate features f_j the goal is to find the feature and offset that minimize the error across all instances. By denoting the value of feature f_j in instance i as f_j^i and using a binary indicator variable α_j to select a feature, we can express the error E_i of a single instance as:

$$E_i = \sum_{j=1}^{m} \alpha_j f_j^i + b - \texttt{ub}_i \tag{6}$$

Finding the best symbolic expression now corresponds to finding the assignment to the indicator variables α_j and offset b that minimizes the overall error: $\texttt{sum}(|E_i|)$. COUNT-CP imposes an additional constraint that the symbolic bound must be an upper bound of the learned bounds. In other words, E_i cannot be negative. This ensures that learned constraints will be satisfied by every training example. We can now write the optimization problem as:

$$\min_{\alpha_j} \sum_i E_i \quad s.t. \sum_j \alpha_j = 1 \ \wedge \ \alpha_j \in \{0, 1\} \ \wedge \ \forall i : E_i \geq 0 \tag{7}$$

In practice, this problem can be solved easily by computing the optimal offset b_j and resulting error E_{ij} for each feature f_j and picking the index j^* with the smallest error:

$$b_j = \texttt{max}\{\texttt{ub}_i - f_j^i \mid i\} \tag{8}$$

$$E_{ij} = f_j^i + b_j - \texttt{ub}_i \tag{9}$$

$$j^* = \texttt{argmin}_j E_{ij} \tag{10}$$

The resulting expression will then be: $f_{j^*} + b_{j^*}$. Since COUNT-CP also includes the constant 0 as a feature, it will still return a numeric bound for expressions that do not depend on a symbolic feature. In fact, in these cases the fitted expression will simply be $\texttt{max}_i\ \texttt{ub}_i$, the aggregation operation we have already applied for aggregating bounds across examples, sequences and partitions.

▶ **Example 6.** For the nurse scheduling example, given a custom feature *minimal-staffing-requirement* (`msr`), COUNT-CP can now learn that the sum of every column (=nurses working on a day) is lower bounded by the `msr` leading to `foreach column: sum(column) >= msr`.

This approach can be applied on any of the `lb` or `ub` of the tuples $\langle e, \cdot, \texttt{lb}, \texttt{ub} \rangle$ found.

3.5 Filtering constraints

Filtering out useless constraints

By computing bounds over expressions, COUNT-CP ensures that learned constraints are always satisfied by training examples. However, by computing bounds over every expression, partition and sequence, COUNT-CP will always find valid lower- and upper-bounds. This can cause COUNT-CP to return many constraints which are true by default or trivially entailed by another constraint.

First, let us have a look at trivial constraints. As an example of a constraint that is true by default, consider `x[i] + x[j] <= c`, where `c` is the sum of the maximal values of the domains of variables `x[i]` and `x[j]`. COUNT-CP filters out trivial constraints by detecting them during the propositional learning step. Whenever COUNT-CP learns a local constraint $\langle e, V, \texttt{lb}, \texttt{ub}\rangle$, it also computes the minimal and maximal values of the expression e for the variables V and their domains: $l = \texttt{min}\ e(V), u = \texttt{max}\ e(V)$. If $\texttt{lb} = l$ (or $\texttt{ub} = b$) the bound is marked as trivial and the corresponding constraint, as well as any first-order constraint that includes that bound, is removed.

Second, let us consider trivial entailment. Consider two constraints: 1) for each column, the absolute difference of every pair of variables in the column in at least 1 ($\langle \texttt{abs}, \texttt{columns}, \texttt{all pairs}, 1, \_\rangle$); and 2) for the entire matrix, the absolute difference of every pair of variables in the matrix in at least 1 ($\langle \texttt{abs}, \texttt{all}, \texttt{all pairs}, 1, \_\rangle$). Because the pairs of variables in the first constraint is a subset of the pairs of variables in the second constraint, the first constraint is entailed by the second one, *unless* it has a stricter bound.

Consider two first-order constraints $c_1 = \langle e, P_1, s_1, l, u\rangle$ and $c_2 = \langle e, P_2, s_2, l, u\rangle$ with shared bounds (in practice entailment is computed for upper and lower bounds separately). If both constraints share the same partition $P = P_1 = P_2$ but one of the sequences is a subset of the other sequence: $\forall p \in P : s_1(p) \subset s_2(p)$, then c_1 is entailed by c_2. Since the sequence grammar is fixed, entailment between sequences can easily be computed in an offline step before learning. More generally, c_1 is entailed by c_2 if the union of sets of indices from c_1 is a subset of the union of sets of indices from c_2 (as in the example above): $\bigcup_{p \in P_1} s_1(p) \subset \bigcup_{p \in P_2} s_2(p)$. Because we allow users to provide custom partitions, the more general entailment cannot be fully pre-computed. COUNT-CP uses these entailment checks to filter out entailed constraints.

Because first-order constraints are made up of many local constraints, filtering out first-order constraints can drastically reduce the number of local constraints. This decreases the time it takes to solve a learned model and find new solutions, without affecting the quality of the model.

Filtering out overly restrictive constraints

COUNT-CP learns constraints that are satisfied by all training examples. However, there is a risk that the learned constraints exclude valid *unseen* solutions. Ideally, unconstrained expressions are detected by the trivial constraint detection step. However, given few training instances, COUNT-CP might find spurious constraints and produce bounds for unconstrained expressions. This may lead to the incorrect rejection of valid solutions.

Unfortunately, this problem is much more pertinent when learning constraints across instances and extrapolating to unseen instances. An incorrect, loose bound for an unconstrained expression might reject a few unseen solutions of the same size, however, it may reject large amounts of solutions for larger, unseen instances. COUNT-CP attempts to alleviate this issue by monitoring the errors E_i computed during the symbolic bound computation. If

Table 1 List of problems used in the experiments along with the background knowledge provided as the user input to COUNT-CP.

Problem	**User Input**	
	Custom partitions	Semantic constants
Sudoku	Blocks of variables	–
Magic Square	–	–
N-Queens	Diagonals	–
Graph Coloring	Edges of the graph	–
Nurse Rostering	–	Staffing requirements

the errors exceed a given threshold, COUNT-CP opts to reject the bound and produce no constraint instead.

In theory, this type of filtering can occur at every step where different bounds are aggregated – over training examples, over sequences, across partitions – however, since bounds naturally vary, a lot of training data is required to avoid rejecting *valid* bounds.

4 Experiments

In this section, we empirically answer the following research questions:

Q1 How well does COUNT-CP perform on instances used during training?
Q2 Do models learned by COUNT-CP generalize to unseen instances?
Q3 How does the performance change with the size of the training set?
Q4 How fast is COUNT-CP and how does the run-time scale with the number of training examples?
Q5 How effective is the filtering step in COUNT-CP?

To answer these questions, we use COUNT-CP to learn models for a set of benchmark problems and evaluate its performance according to different metrics. The code is available online[1] and uses the CPMpy modeling library [9]. The benchmark problems (see Table 1) consist of problems selected from CSPLib[2], which is a library of test problems for constraint solvers, and an adapted nurse scheduling problem used to evaluate COUNT-OR [11]. The language bias used in COUNT-CP is not expressive enough to model all the CSBLib problems, therefore, we selected problems that COUNT-CP should be able to learn successfully. We hope these experiments will showcase the capabilities of our approach and the viability of our architecture across different problem domains. The language bias of our approach can be extended to cover more complicated constraints by adding building blocks to the various grammars at the cost of increasing the run-time. We leave the exercise of crafting biases to cover larger benchmarks to future work.

Performance measures

The performance of the learned models are measured in terms of *Precision* and *Recall*. Precision tells us what percentage of the learned feasible region is actually feasible in the target model, while recall tells us what percentage of the target feasible region is captured by the learned model. Ideally, having high precision is more desirable, as it ensures that the solutions generated using learned model have higher chances of being feasible, while high recall means we can generate many feasible solutions.

[1] https://github.com/ML-KULeuven/COUNT-CP
[2] https://www.csplib.org/

Table 2 Performance of COUNT-CP across different problems and different training sizes. The results are shown only for training instances.

Training Size	1		5		10	
	Precision	Recall	Precision	Recall	Precision	Recall
Sudoku	100%	100%	100%	100%	100%	100%
Magic Square	100%	100%	100%	100%	100%	100%
N-Queens	100%	100%	100%	100%	100%	100%
Graph Coloring	100%	57.5%	100%	100%	100%	100%
Nurse Rostering	100%	15.5%	100%	100%	100%	100%

Calculating these performance measures is not trivial. We sample 100 solutions from the learned model and compute how many satisfy the target model – this gives us precision. The recall is computed by instead sampling 100 solutions from the target model and computing how many satisfy the learned model. Sampling uniformly from the feasible region of a model is extremely hard [20], however, the CPMPy constraint modeling framework[3] allows us to instruct the constraint solver to find solutions close to a given starting point. By generating each of the 100 solutions using different random starting points, we try to obtain a representative sample, which provides better estimates of the true precision and recall.

Setup

For each problem, we include two different *training* instances to learn from and another unseen *test* instance to evaluate the performance on unseen instances. Problems instances are instantiations of problems to a specific size or setting. For example, for the Sudoku problem, the training instances are of size 4×4 and 9×9, while the test instance is of size 16×16. For nurse rostering, different instances correspond to different hospitals, which have different staffing requirements and numbers of nurses.

For every problem instance, e.g., a 9×9 Sudoku, we use a set of training examples – solutions of the problem instance – to learn from. Specifically, we learn from $1, 5$ or 10 examples per instance. The performance is then evaluated using the sampling procedure described above.

Q1. How well does COUNT-CP perform on instances used during training?

To answer Q1, we report the precision and recall on the training instances (see Table 2). Using just a single training example per instance, COUNT-CP already learns models that have 100% precision. For two of the problems, a single example is not enough to obtain 100% recall. However, when given 5 training examples, COUNT-CP achieves 100% recall for all benchmark problems.

Q2. Do models learned by COUNT-CP generalize to unseen instances?

By measuring the precision and recall of models learned by COUNT-CP for unseen test instances (see Table 3), we can observe that the performance is similar to the performance seen on training instances: Learning from just one example per training instance, our approach obtains 100% precision – even for unseen instances – and given 5 examples, COUNT-CP achieves 100% recall, as well.

[3] `https://github.com/CPMpy/cpmpy`

Table 3 Performance of COUNT-CP across different problems and different training sizes. The results are shown only for test instances.

Training Size	1		5		10	
	Precision	Recall	Precision	Recall	Precision	Recall
Sudoku	100%	100%	100%	100%	100%	100%
Magic Square	100%	100%	100%	100%	100%	100%
N-Queens	100%	100%	100%	100%	100%	100%
Graph Coloring	100%	61%	100%	100%	100%	100%
Nurse Rostering	100%	18%	100%	100%	100%	100%

In this paper, we argued for the need to introduce symbolic bounds in constraints in order for them to be able to generalize to unseen instances. We evaluate this claim qualitatively by comparing the scores obtained by COUNT-CP on the *Nurse Rostering* problem with a modified version that simply keeps numeric bounds. As expected, we see that the learned model cannot generalize well to unseen instances with different staffing requirements (see Table 4).

Table 4 Comparison of COUNT-CP against a naive version which learns numerical bounds instead of symbolic expressions.

Training Size	1		5		10	
	Precision	Recall	Precision	Recall	Precision	Recall
COUNT-CP	100%	18%	100%	100%	100%	100%
Naive version	0%	100%	0%	100%	0%	100%

Q3. How does the performance change with the size of the training set?

The change in performance across different training sizes is shown in Table 2 and Table 3. When we use more training examples, COUNT-CP learns less tight bounds, which in turn would lead to improved recall, and that is exactly what we observe in the results as well. In most cases we learn perfect model with just one example, and in cases where this is not the case (last two rows in both tables), the performance improves as the size of the training set increases.

Table 5 COUNT-CP learns all problems in less than a minute except nurse rostering where it takes close to 5 minutes.

Problem	**Time Taken (in seconds)**
Sudoku	2.5
Magic Square	58.3
N-Queens	8.5
Graph Coloring	22.6
Nurse Rostering	328.3

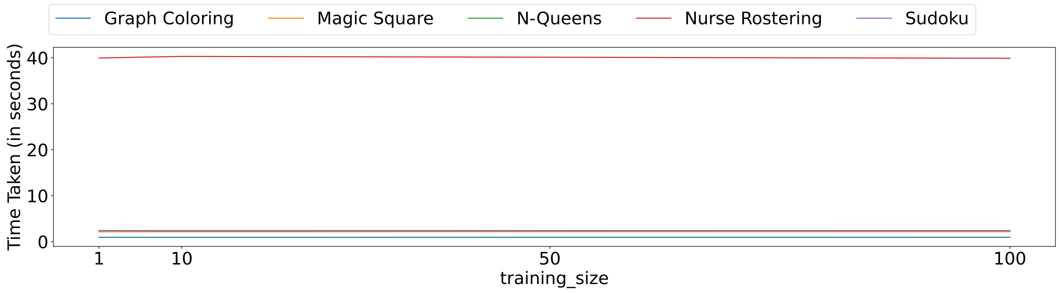


Figure 1 The learning time of COUNT-CP remains consistent when increasing the number of training examples.

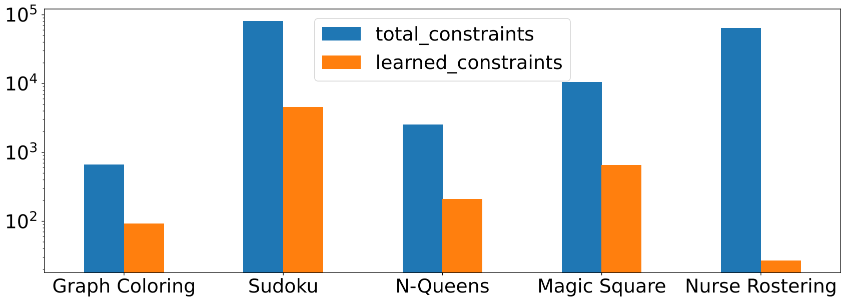


Figure 2 Filtering step in COUNT-CP leads to more than 96% reduction in the total number of learned constraints.

Q4. How fast is COUNT-CP and how does the run-time scale with the number of training examples?

COUNT-CP learns most problems in a less than a minute, except for the *Nurse Rostering* problem, for which it requires close to 5 minutes (see Table 5). Considering the time taken by experts to model a problem and the fact that once learnt, these models can be used to solve problems of different sizes, we can characterize our learning time as lightning fast in comparison. The run-time depends mainly on the number of decision variables and since COUNT-CP enumerates all pairs of variables, the run-time increases quadratically with the number of decision variables. Here, again, the ability to learn models from small instances and apply them to much larger instances makes COUNT-CP useful in practice.

COUNT-CP scales linearly with the number of training examples, however, by evaluating candidate expressions efficiently using vectorized operations, the impact of the number of training instances is negligible in most cases (see Figure 1). This, again, is good news, as it allows the user to provide large number of training examples to learn more accurate models, while avoiding long learning times.

Q5. How effective is the filtering step in COUNT-CP?

In Subsection 3.5, we discussed the importance of filtering out useless and overly restrictive constraints. Unnecessary constraints make the learned models less interpretable and slower to solve. Filtering out these constraints, however, has a cost: It significantly increases the learning time by adding overhead for every single local constraint learned.

To answer Q5 and evaluate the effectiveness of the filtering step, we compare the total number of possible constraints produced by COUNT-CP with the constraints included in the learned model after the filtering step. Our experiments show that COUNT-CP is able to drastically reduce the number of constraints it outputs (see Figure 2). On average, COUNT-CP filters out 96.7% of the constraints, significantly improving the interpretability and solving time of learned models.

5 Conclusion

In this paper, we presented the novel constraint learner COUNT-CP, which uses simple grammars and a generate-and-aggregate approach to generate mathematical expressions, compute their bounds across training examples and group the learned constraints to obtain first-order constraints that can generalize to unseen instances. A symbolic expression fitting step is used to obtain symbolic bounds for expressions, making them instance-independent. Additionally, COUNT-CP uses an effective filtering step to remove useless and spurious constraints. We empirically evaluated our approach on a set of suitable benchmark problems. This evaluation showed that, indeed, COUNT-CP is able to learn compact, high quality models quickly. The learned models achieve high precision and recall, even when only trained on a handful of examples. Because the learned models contain first-order constraints and support bound expressions, these results also hold true for unseen instances. Finally, our simple interaction protocol allows users to provide relevant background knowledge without requiring any specialized knowledge about the underlying constraint language. We believe that the COUNT-CP architecture is a promising approach to constraint learning that can be further tuned to learn a wide range of constraint problems.

References

1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.

2 Nicolas Beldiceanu and Helmut Simonis. A Model Seeker: Extracting global constraint models from positive examples. In *International Conference on Principles and Practice of Constraint Programming*, 2012.

3 Christian Bessiere, Remi Coletta, Abderrazak Daoudi, Nadjib Lazaar, Younes Mechqrane, and El-Houssine Bouyakhf. Boosting constraint acquisition via generalization queries. In *ECAI*, pages 99–104, 2014.

4 Christian Bessiere, Remi Coletta, Eugene C. Freuder, and Barry O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 123–137, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

5 Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A sat-based version space algorithm for acquiring constraint satisfaction problems. In *Machine Learning: ECML 2005*, pages 23–34, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

6 Christian Bessiere, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. Query-driven constraint acquisition. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 50–55, 2007. URL: `http://ijcai.org/Proceedings/07/Papers/006.pdf`.

7 Abderrazak Daoudi, Nadjib Lazaar, Younes Mechqrane, Christian Bessiere, and El Houssine Bouyakhf. Detecting types of variables for generalization in constraint acquisition. In *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 413–420. IEEE, 2015.

8 Luc De Raedt and Sašo Džeroski. First-order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70(1-2):375–392, 1994.

9 Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *The 18th workshop on Constraint Modelling and Reformulation at CP*, pages 1–8. ModRef, 2019.

10 Samuel Kolb, Sergey Paramonov, Tias Guns, and Luc De Raedt. Learning constraints in spreadsheets and tabular data. *Mach. Learn.*, 106(9-10):1441–1468, 2017. doi:10.1007/s10994-017-5640-x.

11 Mohit Kumar, Stefano Teso, Patrick De Causmaecker, and Luc De Raedt. Automating personnel rostering by learning constraints using tensors. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 697–704, 2019. doi:10.1109/ICTAI.2019.00102.

12 Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, pages 45–52. IEEE Computer Society, 2010. doi:10.1109/ICTAI.2010.16.

13 Fabio Massacci and Laura Marraro. Logical cryptanalysis as a sat problem. *Journal of Automated Reasoning*, 24(1):165–203, 2000. doi:10.1023/A:1006326723002.

14 Ralph Eric Mcgregor. *Automated Theorem Proving Using Sat.* PhD thesis, Clarkson University, USA, 2011. AAI3471671.

15 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*, pages 102–115, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

16 Tomasz Pawlak. Synthesis of mathematical programming models with one-class evolutionary strategies. *Swarm and Evolutionary Computation*, 44, May 2018. doi:10.1016/j.swevo.2018.04.007.

17 Tomasz Pawlak and Krzysztof Krawiec. Automatic synthesis of constraints from examples using mixed integer linear programming. *European Journal of Operational Research*, 261, February 2017. doi:10.1016/j.ejor.2017.02.034.

18 Tomasz Pawlak and Krzysztof Krawiec. Automatic synthesis of constraints from examples using mixed integer linear programming. *EJOR*, 2017.

19 Luc De Raedt, Andrea Passerini, and Stefano Teso. Learning constraints from examples. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence - AAAI 18*, pages 7965–7970. AAAI Press, 2018. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17229.

20 Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S. Meel. Knowledge compilation meets uniform sampling. In *Proceedings of International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, November 2018.

21 L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, November 1984. doi:10.1145/1968.1972.

Combining Reinforcement Learning and Constraint Programming for Sequence-Generation Tasks with Hard Constraints

Daphné Lafleur
Polytechnique Montréal, Canada
Quebec Artificial Intelligence Institute (Mila), Canada

Sarath Chandar
Polytechnique Montréal, Canada
Quebec Artificial Intelligence Institute (Mila), Canada
Canada CIFAR AI Chair, Toronto, Canada

Gilles Pesant
Polytechnique Montréal, Canada

Abstract

While Machine Learning (ML) techniques are good at generating data similar to a dataset, they lack the capacity to enforce constraints. On the other hand, any solution to a Constraint Programming (CP) model satisfies its constraints but has no obligation to imitate a dataset. Yet, we sometimes need both. In this paper we borrow RL-Tuner, a Reinforcement Learning (RL) algorithm introduced to tune neural networks, as our enabling architecture to exploit the respective strengths of ML and CP. RL-Tuner maximizes the sum of a pretrained network's learned probabilities and of manually-tuned penalties for each violated constraint. We replace the latter with outputs of a CP model representing the marginal probabilities of each value and the number of constraint violations. As was the case for the original RL-Tuner, we apply our algorithm to music generation since it is a highly-constrained domain for which CP is especially suited. We show that combining ML and CP, as opposed to using them individually, allows the agent to reflect the pretrained network while taking into account constraints, leading to melodic lines that respect both the corpus' style and the music theory constraints.

2012 ACM Subject Classification Software and its engineering → Constraint and logic languages; Theory of computation → Reinforcement learning; Applied computing → Sound and music computing

Keywords and phrases Constraint programming, reinforcement learning, RNN, music generation

Digital Object Identifier 10.4230/LIPIcs.CP.2022.30

Supplementary Material *Software (Source Code)*: `https://github.com/chandar-lab/RL-Tuner-CP`

Funding Financial support for this research was provided by an IVADO Fundamental Research grant.

Acknowledgements We would like to thank all the members of Chandar Lab, Laboratoire Quosséça, family and friends who gave very useful feedback and improvements before the submission. We would also like to thank the CP 2022 reviewers for their constructive criticism.

1 Introduction

Recurrent Neural Networks (RNNs) [14] are a class of Machine Learning (ML) algorithms renowned for their ability to extract structural information from a corpus in order to generate sequences that mimic said corpus' style. However, some sequence-generation tasks require the final output to respect a set of rules. Music generation, especially applied to Renaissance

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 30; pp. 30:1–30:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

music, is a perfect example of these kinds of tasks. Given a series of melodic lines, RNNs are able to produce sequences that resemble the style of a given composer. Nonetheless, RNNs are not easy to control and it is hard to make them enforce music rules without injecting domain knowledge. It is akin to asking someone with no musical training to extract rules just from analyzing a pile of scores.

RL-Tuner [7] is an algorithm that uses Reinforcement Learning (RL) to bridge the gap between RNNs and hard constraints, resulting in samples that both better respect the constraints and are representative of the data the algorithm was trained on. However, the constraints are enforced by manually tuning a reward for each rule. In this work we use Constraint Programming (CP) with Belief Propagation (BP) to learn better how to satisfy constraints. By using CP instead of checking each rule individually, we benefit from the interactions between all the constraints and may anticipate violations. Furthermore, the addition of BP is a recent advance in CP that provides marginal probabilities for every selected value [12]. These marginals take into account the entirety of the sequence and act as a metric to determine if choosing an action could potentially lead to a complete sequence with no added constraint violations. This information is crucial in increasing what we introduce as the constraint satisfaction of the generated sequences.

We illustrate our framework by applying it to contrapuntal music writing in the style of the Renaissance period. We design two closely-related CP models implementing the rules of counterpoint for melody writing according to a standard textbook [15]. We train our RNN model using a Bach chorale corpus, a widely available source that is compatible to some degree with the constraints we enforce.[1] We combine our CP and RNN models within RL-Tuner and analyze the evolution of the reward to show that our algorithm retains its acquired knowledge from the Bach corpus while more-closely following the rules of counterpoint we added. We also discuss why some constraints may be harder to learn than others.

Our main contribution with this paper is to show that RL can be used to combine RNNs and CP in order to fine-tune training so that it generates sequences that reflect the corpus while enforcing constraints. Even though we applied it here to music generation, such a combination can be useful for other sequence-generation tasks with hard constraints.

In the rest of the paper, Section 2 provides some necessary background on RNNs, RL, CP/BP, and music theory. Section 3 surveys related work. Section 4 describes the core of our contribution: each component, how they interact, their specialization for melody generation. Section 5 presents an empirical evaluation of our contribution. Section 6 discusses our present and future work. Finally we conclude with Section 7.

2 Background

2.1 Recurrent Neural Network

Recurrent Neural Networks (RNN) [5] are a family of ML algorithms able to generate sequences. At every iteration, a token is passed through the network and used to predict the next token. After each prediction, the history is updated to make sure that the next token takes into account the whole sequence.

$$h_t = f(W_h x_t + U_h h_{t-1} + b_h) \qquad \mathbf{\hat{y}_{t+1}} = g(W_y h_t + b_y)$$

[1] Even though Bach belongs to the Baroque period, he would have been (heavily) influenced by Renaissance music.

In the first equation, we obtain the current history h_t by updating the old history h_{t-1} with the current token x_t. Weights W_h and U_h and a bias b_h are applied to control the importance of x_t and h_{t-1} in the update. The activation function f makes the update process non-linear. In the second equation, we compute $\mathbf{\hat{y}_{t+1}}$, a vector indicating the probability distribution of each possible value for the next token using the current history h_t. Once again, weights, a bias and a non-linear activation function are used for the prediction. With this distribution, we can select the value with the highest probability as $\hat{x}_{t+1}$. All the weights and biases are adjusted and learned during the training of the RNN (see below) to produce the best tokens. To measure how good an RNN is, we use the cross-entropy loss function

$$L = -\sum_t \mathbf{y_t} log(\mathbf{\hat{y}_t}).$$

For each token x_t in the dataset, we feed the previous tokens to the RNN and obtain the *predicted* distribution $\mathbf{\hat{y}_t}$. $\mathbf{y_t}$ represents the *target* distribution. It is a one-hot vector where all the components are 0, except for the component representing x_t, which is equal to 1.

Training is done with gradient descent by modifying the weights and the biases to decrease the loss. Once training is over, we use the final weights and biases to predict new sequences.

2.2 Reinforcement Learning and DDQNs

The basic idea of Reinforcement Learning [8] is to learn through interaction with an environment, in order to maximize the sum of the *rewards*. The agent will sample from the *policy* (state-dependent probability distribution) to pick an *action*. Once an action has been picked, the environment will update its *state* based on said action and return a reward, indicating how good the action is. This reward will then be used to update the estimated value of that state-action pair. Once the value has been updated, the policy will be adjusted to encourage actions that have a higher value.

Different families of RL algorithms define ways to compute state-action values. For example, in Q-Learning, the following equation is used:

$$Q(s,a) = [1-\alpha]Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a')]$$

In this equation, state-action value $Q(s,a)$ is updated by summing the reward r and the maximum value of the next state s'. Two hyper-parameters α and γ control the magnitude of the update.

In Deep Q-Networks (DQN) [9], a *neural network* is used to compute the state-action value from the current state. To update the state-action values, we can compute the loss of the network by using the difference between the current value $Q(s,a)$ and the update part of the Q-learning equation (see above):

$$\delta = r + \gamma \max_{a'} Q(s',a') - Q(s,a)$$

We can then adjust the parameters by applying one iteration of gradient descent per update to decrease this loss. However, since Q-Learning uses the maximum value of the next state ($\max_{a'} Q(s',a')$), overestimated values will be encouraged. Double Deep Q-Networks (DDQN) [17] reduce this effect by adding a second network to compute the state-action values:

$$\delta^A = r + \gamma Q^B(s', \mathtt{argmax}_{a'} Q^A(s',a')) - Q^A(s,a)$$

Here we use network A to pick the best next action with $\mathtt{argmax}_{a'} Q^A(s',a')$. However, instead of using A to compute its value, we will use network B, mitigating A's possible overestimation. Once δ^A has been computed, only the parameters of network A will be updated. Note that these two roles (A and B) are assigned randomly every time.

2.3 CP-Based Belief Propagation

CP-based Belief Propagation, introduced in [12], offers a new way to propagate constraints. Instead of simply removing unsupported values from domains, the solver computes the probability that each value will respect a given constraint. These partial probabilities are then sent as messages to the other constraints through their shared variables, which makes them update and refine their own probabilities. Out of these interactions, marginal probabilities are derived for each variable-value pair. These marginals approximate the probability that, if a variable is assigned a given value, all the constraints will be satisfied. They also provide information as to how many possible values respect the constraints.

2.4 Music Basics

We provide a brief introduction to music concepts essential to the understanding of this paper. Generally, music is built from musical notes each having a *pitch* and a *duration*. Pitches an octave apart are grouped in a *pitch class*. A *scale* is an organized subset of pitch classes. There are many kinds of scales – we will consider *diatonic* scales and in particular the major scale featuring the seven natural pitch classes (C, D, E, F, G, A, B). We loosely use the term *melody* for a sequence of notes (pitch, duration) sounded consecutively. An *outline* in a melody is a maximal subsequence of notes between a temporary high point and the next temporary low point (in terms of pitch) or vice-versa. The distance between two pitches is called an *interval*, identified by the integral number of pitch steps from one to the other (e.g. a third between C and E) and by the quality of the interval based on the number of semitones between them (e.g. a major third between C and E; a minor third between D and F). The melodic *motion* between two consecutive notes can either be by *step*, if the notes are adjacent in the scale, or by *skip*.

3 Related Work

The abundance of previous work on music generation, both in ML and in CP, shows that the task we are tackling is especially well suited for both these approaches. Surveys from Briot et al. [1], Fernandez Rodriguez and Vico [13], and the book edited by Truchet and Assayag [16] provide an extensive view on ML and CP being applied to music. In this literature review, we focus on the most relevant ones to us.

RL-Tuner [7] has been the greatest source of inspiration for our work. In the paper, the authors present a Reinforcement Learning algorithm that combines probabilities from an RNN with a reward function measuring how much the generated sequence respects a set of rules. They apply it to music generation. The RL agent used is a Double Deep Q-Network (DDQN). The authors show that RL-Tuner enforces both the music theory laws and the similarity to the RNN. The main difference from our work is that they compute the music theory reward by checking each rule individually in an *ad hoc* fashion based only on the previous notes. We argue that we can improve this algorithm by using CP with belief propagation to compute marginals that take into account the whole sequence and not just the previous notes.

On the ML side, C-RNN-GAN [10] is a classical music generation system that uses a generative adversarial network (GAN). GANs work by having two networks: the generator that creates the piece and the discriminator that tries to distinguish between fake and real music. The goal of the generator is to fool the discriminator, so it learns to create music that is as similar as possible to the dataset.

DeepBach [4] combines two LSTMs (a type of RNN) and one neural network. The first LSTM picks a note based on the *previous* notes. The second one picks a note based on the *future* notes. The neural network picks a note based on the notes that will be played at the same time on other voices. Finally, these three note candidates are sent to another neural network, which will choose the note to be added to the piece.

Coconet [6] uses a Convolutional Neural Network (CNN) to generate Bach chorales. CNNs are usually used for image classification tasks because they are able to handle 2D patterns (whereas RNNs are limited to linear sequences). Combining all four voices of a chorale creates a 2D structure that is well suited for CNNs. Coconet's CNN is used to compute probabilities of a note with respect to its context. Once the probabilities have been computed, Gibb's sampling is applied to populate the music piece. To remove bad choices due to less context information, a window of notes is resampled, with the window's size decreasing over time.

Anticipation-RNN [3] combines two RNNs to enforce unary constraints. In this algorithm, the first RNN is used to predict a unary constraint based on the future notes. This constraint is then used to condition the output of the second RNN, which will pick the note based on the partial sequence and the future constraints.

Young et al. [19] introduce a generative model able to generate music with relational constraints. What is really interesting about this work is that the constraints are synthesized from the dataset instead of being hard-coded. These constraints include equality of different notes and transposition of a sequence of notes (repeating the same pattern with each note transposed by the same amount). They try three different techniques to incorporate the constraints: sampling from the model and rejecting if the token doesn't respect the constraints, representing the constraints in a graph convolutional network, and using MIP to maximize an objective function.

4 Sequence Generation with Hard Constraints

Figure 1 gives an overview of our architecture. We describe the training process:

(i) The partial sequence is sent to the DDQN.

(ii) The DDQN produces the next note.

(iii) The RNN computes $p(a|s)$ from both the partial sequence s and the next note a. The CP models do the same to obtain the marginals and the violations.

(iv) The output of the RNN and of the CP models are used to compute the reward.

(v) The DDQN's weights are updated based on that reward and the next note is added to the sequence.

The different components are detailed below. Our code is publicly available[2].

4.1 CP models

Among several textbooks that teach counterpoint, the relatively recent "Modal Counterpoint, Renaissance Style" by P. Schubert [15] is of particular interest to create our CP models. It gathers rules from several treatises, including the seminal Fux, draws from the works of many period composers, and pays much attention to the quality of melodic lines. But foremost it strongly appeals to the constraint programmer: rules are declarative and classified as

[2] `https://github.com/chandar-lab/RL-Tuner-CP`

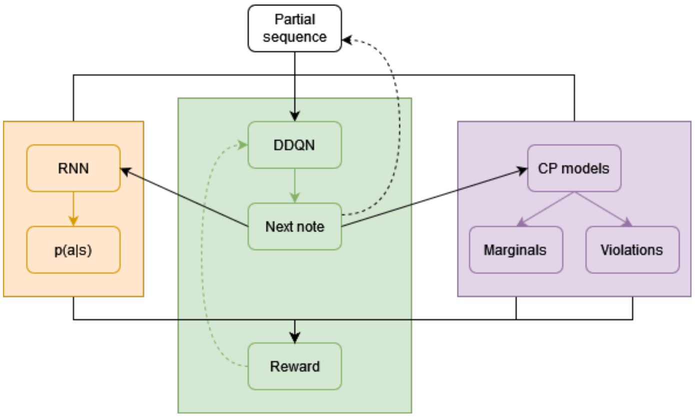


Figure 1 An overview of our training architecture.

hard and soft. The constraint models we built are based on the rules in this book. We implemented our models using the MiniCPBP solver[3] in order to have access to marginal probabilities.

4.1.1 Variables and domains

Even though rhythm is an important part of a melody, these rules (and those typical of the period, aside from the general recommendation that there should be rhythmic variety) do not take it into consideration. Accordingly, we represent a melody as a sequence of n pitches (notes). As is standard practice, we transpose the melodic lines from the corpus so that they are all in the same key (C major) and generate sequences in that same key.

Pitches should belong to the key, though we allow an accidental B♭ to avoid tritone intervals (i.e., augmented fourth or diminished fifth) from F. We also want a melody to stay within the octave range of its key, possibly extending it by one note below and above that range (resp. B♭ and D). The pitch values represent the number of semi-tones above the lowest pitch in the dataset (value 0 is G). Because of the range restriction explained above, the lowest value allowed in the CP domain is 3 (or B♭). We define our variables as:

$$\texttt{pitch}[i] \in \{3,4,5,7,9,10,12,14,15,16,17,19\} \quad 1 \leq i \leq n$$

Schubert's book adds restrictions about the permitted values of the intervals. An interval can span no more than a sixth (9 semi-tones), except that an octave is also allowed and that a tritone is not. That interval can be sung up or down. We also want to avoid intervals of a sixth, except for an ascending minor sixth. Finally, two consecutive notes should not have the same pitch:

$$\texttt{interval}[i] \in \{\pm1,\pm2,\pm3,\pm4,\pm5,\pm7,+8,\pm12\} \quad 1 \leq i \leq n-1$$

An interval is computed as the number of semi-tones between two consecutive notes. In other words:

$$\texttt{interval}[i] = \texttt{pitch}[i+1] - \texttt{pitch}[i] \quad 1 \leq i \leq n-1$$

[3] https://github.com/PesantGilles/MiniCPBP

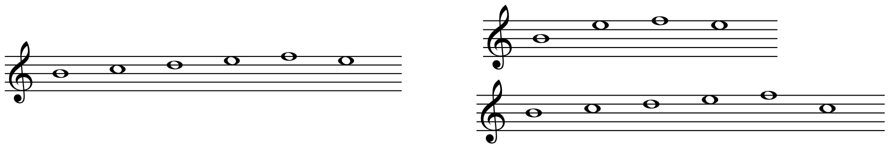


Figure 2 A legal augmented fifth outline (left) and two forbidden ones (right).

4.1.2 Constraints

Now that the main variables have been defined, we describe the melodic constraints we consider. Though some of them are considered soft by Schubert, here all are expressed as hard constraints for our system to learn.

(i) End on the tonic.

$$\texttt{pitch}[n] = \mathrm{C}$$

(ii) End by stepwise descent.

$$-2 \leq \texttt{interval}[n-1] < 0$$

(iii) Use more steps than skips.

$$\sum_{i=1}^{n-1} (\texttt{interval}[i] < -2 \vee \texttt{interval}[i] > 2) < \frac{n-1}{2}$$

(iv) An accidental B♭ should be followed by a descending interval.

$$\texttt{pitch}[i] \neq \mathrm{B}\flat \vee \texttt{interval}[i] < 0 \qquad 1 \leq i \leq n-1$$

(v) Tritone outlines. An outline of an augmented fourth is prohibited. An outline of a diminished fifth is allowed only if it is completely filled in by step (interval smaller than 2) and then followed by a step in the opposite direction. Fig. 2 gives examples of one legal and two forbidden tritone outlines. This rule is not as straightforward to express and requires that we consider several consecutive notes.

A useful observation is that the only allowed tritone outline spans four steps (which could correspond to more than five notes if we have repeated notes). We express this using a `regular` [11] constraint on `interval` variables with an automaton recording in its states the number of steps and semi-tones in a potential outline. Fig. 3 gives the automaton for ascending outlines – the case of a descending outline is similar. Though admittedly a little hard to decipher, it shows our ability to model complex rules.

(vi) A skip should be preceded or followed by a step in the opposite direction.

(vii) Avoid more than two successive skips.

(viii) Avoid skipping on both sides of a temporary high or low point.

(ix) Two successive skips in the same direction should be small. We consider a skip of a third or fourth to be small.

(x) "Pyramid" rule. An ascending outline should not have large skips following smaller skips or steps; a descending outline should not have large skips preceding smaller skips or steps. As when building a pyramid, larger blocks should be used at the bottom and smaller ones at the top. (Keep in mind that such rules are derived from practice at that time according to aesthetics and "singability".)

Because they are closely related, the previous five rules (**(vi)** to **(x)**) are handled together, through a single `cost-regular` [2] constraint on a characterization of individual intervals as ascending/descending steps, small skips (a third or a fourth), and skips (fifth, sixth,

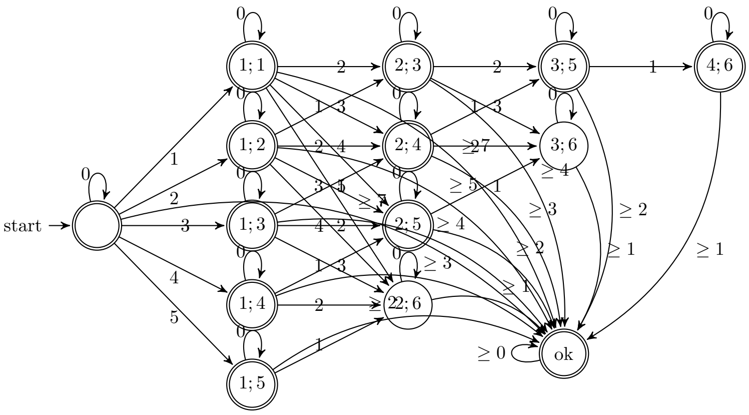


Figure 3 Automaton to check for ascending tritone outlines. Negative transitions to descending tritone states are not shown.

or octave). It is sufficient to create a state for each pair of characterizations of intervals. All states are accepting but each transition carries a cost corresponding to the sum of the penalties for every rule broken on that transition. An interval value of 0, corresponding to a repeated note, loops to the same state. In the interest of clarity, we do not show this automaton.

(xi) **Modal range.** A melody must cover (include notes from) the whole octave range. Longer melodies are encouraged to cover that range every so many notes. Through a single `gcc` constraint on `pitch` variables, we express the requirement that each pitch in that range should appear at least once (setting their minimum number of occurrences to 1).

(xii) **Minimum number of modal skips.** The final of the mode (C) and the pitch class four steps above it (G) are more important than the other pitch classes. Skips between these two pitches are characteristic of the mode and help establish it. We can enforce it by lower bounding to 3 the sum of reified constraints expressing modal skips.

$$\sum_{i=1}^{n-1}[(\texttt{pitch}[i] = \textsc{c} \wedge \texttt{pitch}[i+1] = \textsc{g}) \vee (\texttt{pitch}[i] = \textsc{g} \wedge \texttt{pitch}[i+1] = \textsc{c})] \geq 3$$

We thus have a total of 12 constraints − 4 (because 5 constraints are handled together) + 4 constraints (the ones used to restrict the domains of our variables) = 12.

4.1.3 Marginals model

Our goal is to have a model that, given a sequence of notes, will compute the marginal probabilities of the next note. Since that sequence is provided by the RL-Tuner, it is possible that there have already been constraint violations. To clearly measure the impact of the next note, regardless of previous violations, we apply the constraints and domain restrictions only on the future notes. However, this requires some adjustments in our constraints. For instance, since our automata can no longer start at the beginning of our sequence, we need to modify the starting state according to the previous notes. Furthermore, if we want to respect a certain lower bound (for example, the minimum number of modal skips in constraint **(xii)**), we need to adjust that bound based on the input to the CP model. After the first propagation of the constraints, we compute the marginals using belief propagation and return the corresponding value for the chosen next note, provided by the RL agent.

4.1.4 Violations model

This model is very similar to the one presented above. The main difference is that, instead of enforcing the constraints, we use reified constraints to count the number of violations for each constraint. This causes a little bit of a challenge for some constraints. The tritone outline constraint **(v)** is limited to 25 different interval values (the domain of the intervals is from -12 to +12). However, if we allow the possibility of any interval, which we must since no constraints are enforced, we have a total of 57 different interval values (from -28 to +28). To deal with values not supported by the automaton, we use a soft version of that constraint [18]. If the interval is illegal, the tritone outline constraint will return one constraint violation.

Something similar is done for the other automaton (constraints **(vi)** to **(x)**). However, since this is a `cost-regular` constraint, we cannot assign to it a single violation. To compute the number of violations for this constraint, in case of an invalid interval, we decompose the cost in two sections. First, we compute the cost associated to the current transition by averaging the costs associated to transitions from the current state. Afterwards, we compute the cost of the future transitions with the same `cost-regular` constraint, but applying it only on the future transitions and modifying the current state.

For the rest of the constraints, we use reified constraints. For example, instead of restricting the domain of the notes, we compute the number of notes that are outside of this domain. The number of violations for each constraint is then a variable, and we return the smallest value in its domain.

This model receives as input the note sequence and the value of the chosen note. It will then only return the minimum number of violations for each of the twelve constraints, for that chosen note.

4.2 Training the RNN

We extracted melodic lines from the Boulanger-Lewandowski corpus of Bach chorales[4]. We used publicly-available code[5] to preprocess our midi files. During this preprocessing, we check every repeated note (midi files having a really small granularity, every repeated note has to be a note that is held) and replace the subsequent repetitions with a special token, indicating which notes are held. We then extracted melodic lines using a quarter-note granularity, where we ignored the hold token since we do not take into account the note duration. We transposed all sequences in the key of C and kept only the chorales in C major, for a total of 200 sequences. That dataset was split into 150 examples for training and 50 examples for validation. In order to have notes in similar ranges, we used only the soprano voice which usually represents the main melody. Since the RNN used in the RL-Tuner was trained with the Magenta library, we created our own set of inputs and outputs, using one-hot encoding of length 29 – two and a half octaves, similar to the vocal range – for the input representing the last note and a single value for the output representing the predicted note. We trained the RNN on 1000 iterations using the basic-rnn configuration provided by Magenta[6] with a batch size of 64 and two layers of 64. These hyperparameters were tailored to our dataset. The loss function used is the softmax cross-entropy loss (softmax meaning that the activation function used to compute the probability distribution is the softmax function).

[4] `http://www-ens.iro.umontreal.ca/~boulanni/icml2012`

[5] `https://medium.com/analytics-vidhya/convert-midi-file-to-numpy-array-in-python-7d00531890c`

[6] `https://github.com/magenta/magenta/tree/main/magenta/models/melody_rnn`

Though we were able to reach a training accuracy of 90%, the final validation accuracy obtained was 50% (compared to 92% in the original paper). This is likely due to the size of our dataset (200 sequences vs 30 000), which is substantially smaller than the one used in the RL-Tuner (more on that in Section 7).

4.3 RL-Tuner

We copy the weights of the RNN to initialize the DDQN. Before training, the value of each action (or note) is equal to the probability returned by the RNN since it is exactly the same network. During training, the next note, sampled from this distribution by the DDQN, is used to compute the reward, by using the probability returned by the RNN and evaluating with respect to each constraint applied. With the final reward, the weights (and the policy) of the DDQN are updated, without changing the original RNN, thus making the DDQN able to learn independently from the RNN.

The architecture briefly described above can also be found in the original RL-Tuner paper [7]. The sections below explain the changes we made to the RL-Tuner in order to include our CP models.

4.3.1 Restricting sampling to the feasible domain

In the original paper, a note is chosen either randomly or sampled from the DDQN probabilities. This leads to a lot of bad choices with respect to the constraints. That is why we considered an additional option, where the output of the DDQN is restricted according to the number of violations, at training time only. In other words, the algorithm can only sample from notes that have 0 violations. We called this option “restrict domain”.

4.3.2 Reward functions

We implemented seven different reward functions. Here v represents the number of violations for each constraint, m is the marginal for the chosen note and $p(a|s)$ is the probability of the chosen note a according to the RNN, based on s, the partial note sequence. We added two constants, k and c to weigh the different parts of the reward and bring them to a similar range [7].

$$\begin{aligned}
\texttt{violations} &= -\sum v \\
\texttt{marginals} &= km \\
\texttt{marginals\_violations} &= km - \sum v \\
\texttt{rnn} &= log(p(a|s)) \\
\texttt{rnn\_violations} &= log(p(a|s)) - c\sum v \\
\texttt{rnn\_marginals} &= log(p(a|s)) + ckm \\
\texttt{rnn\_marginals\_violations} &= log(p(a|s)) + c(km - \sum v)
\end{aligned}$$

5 Experiments

We ran experiments to answer the following research questions:

1. Can CP reduce the number of constraint violations?
2. Can this be done without forgetting stylistic knowledge acquired by the RNN?
3. What is the impact of restricting the domain while sampling notes?
4. Are all constraints as easy to learn?

[7] Unlike for the RNN probabilities we do not use log-marginals because several marginals are null, which would generate $-\infty$ rewards.

Table 1 Hyper-parameters of our RL-Tuner.

Parameter	Value	Parameter	Value
random_action_probability	0.1	one_hot_length	29
store_every_nth	1	algorithm	q
train_every_nth	5	reward_scaler (c)	2
minibatch_size	32	cp_reward_scaler (k)	40
discount_rate	0.5	output_every_nth	5000
max_experience	100000	num_notes_in_melody	32
target_network_update_rate	0.01	num_steps	50000
rnn_layer_sizes	[64, 64]	exploration_period	25000

5.1 Experimental Setup

We performed experiments for each reward function twice, one where we restricted the domain and one where we did not. We chose the same constant c as in the original paper [7], since it allegedly produced better samples. The value for k was set to 40, converting our marginals to values between 0 and 40. Table 1 presents the hyper-parameters chosen to train our model as named in the RL-Tuner configuration.

Each experiment was averaged across 10 different seeds for a total of 50 000 iterations. Every 5000 iterations, we generated 10 sequences of 32 notes to evaluate our model and measure the evolution of our metrics. We focused mainly on two performance metrics. To answer Question 1, we measure the `constraint satisfaction`, a value between 0 and 1 indicating how good the algorithm is at following constraints (1 being perfection):

$$\texttt{constraint\_satisfaction} = \frac{\sum_{i=1}^{n}(1 - \frac{v_i}{v_i^{\max}})}{n}$$

where n is the number of constraints, v_i is the number of violations for constraint i, summed over all 10 sequences, and $v_i^{\max}$ is the maximum number of potential violations for that constraint during all the 10 sequences.

The second metric used to answer Question 2 is $log(p(a|s))$, the `rnn reward`. For each chosen note, we check the RNN probability and compute an average reward over every note and every sequence. The higher it is, the closer the generated sequence is to the trained model. This metric was also used in the RL-Tuner paper to show that the model does not "forget" the stylistic knowledge learnt from the corpus while enforcing the constraints.

Question 3 can be answered by comparing our two metrics, while restricting the domain or not. To answer Question 4, we analyze the number of violations for each constraint separately by plotting the values returned by the violations model.

Table 2 presents the final constraint satisfaction and average rnn reward for each of our reward functions.

5.2 Comparing metrics to the RL-Tuner

Since we have a different corpus (Bach instead of pop music) and different constraints, we cannot compare our results directly with those in the RL-Tuner paper. However, line 4 without domain restriction allows us to compare to the RL-Tuner approach because we only count the number of violations. We can see here that line 4 is better than line 7. This is similar as what was showed in the paper. However, we can also see that adding the marginals in the mix offers another improvement of 4%.

Table 2 Constraint satisfaction (sat) and average rnn reward ($\log(p(a|s))$) with different reward functions. *Takeaway:* rnn + marginals + violations yields the best constraint satisfaction and its rnn reward is still higher than without using CP.

	Restrict domain			
	No		Yes	
Reward function	sat (%)	$\log(p(a\|s))$	sat (%)	$\log(p(a\|s))$
1. violations	60.1	-4.9	62.1	-4.5
2. marginals	75.1	-2.1	75.8	-2.2
3. marginals + violations	76.9	-1.8	74.3	-2.3
4. rnn + violations (similar to RL-Tuner)	77.2	-1.8	74.7	-2.4
5. rnn + marginals	77.4	**-1.4**	76.6	**-2.0**
6. rnn + marginals + violations	**81.6**	-1.9	75.4	-2.3
7. rnn	74.5	-2.2	**76.9**	**-2.0**

5.3 Without domain restrictions

Without domain restrictions, as we expected, most of our CP models yield a better constraint satisfaction than just using the output from the RNN (lines 2 to 6 vs 7). This shows that, overall, CP helps the agent to respect the constraints. However, it is not the case for the violations reward function, which does a lot worse than the RNN alone (line 1 vs 7). Further experimentations would be required to understand if it simply takes more iterations to see an improvement in that case.

We wondered if CP and the RNN would counteract each other, the RNN wanting to stay close to the corpus and CP aiming to constrain the algorithm. On the contrary, combining them yields better constraint satisfaction than using them separately (lines 4 to 6 vs 1 to 3). This could be due to our choice of corpus: it is likely that Bach generally followed the music rules we imposed and so the RNN learnt them to some degree. However, adding the marginals and the violations gave the model a second push towards a better constraint satisfaction. The same can be said for the rnn reward. Yet again, it seems like the CP model generally helps the RNN to stylistically represent the corpus (lines 4 to 6 vs 1 to 3). This is once again due to concordance between our corpus and our constraints.

5.3.1 Combining violations and marginals

The reward function yielding the best constraint satisfaction is the `rnn + marginals + violations` function (line 6). Both CP models are necessary because the marginals provide information about how good the chosen note is to respect the constraints in the long run and the violations provide information about how big the constraint violation is. If a note does not break any constraint, the violations will be 0 but the marginals will give new information (it could return 100% if the chosen note is the only good choice). If a note breaks some constraints, the marginals will be 0 but the violations will provide new information on the number of constraints that were violated, in other words, how bad this choice is. That's why the best performance is obtained by combining marginals and violations, both providing information in different contexts.

The algorithm with the highest rnn reward is the `rnn + marginals` (line 5). It appears that adding the violations to this function increases the constraint satisfaction at the expense of the rnn reward. However, even if the rnn reward of line 6 (best constraint satisfaction) is lower than line 5, it is still an improvement over the initial `rnn` (line 7).

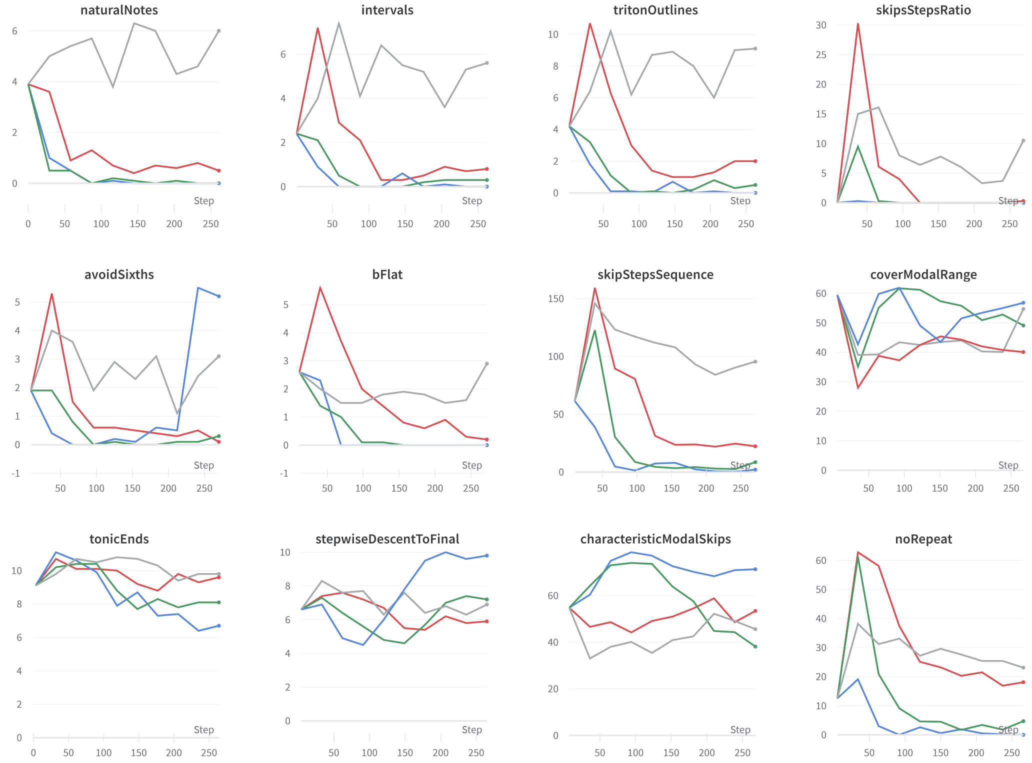


Figure 4 Number of violations for each constraint for different reward functions without domain restriction (red = rnn + violations, blue = rnn + marginals, green = rnn + marginals + violations, gray = rnn). *Takeaway:* For most of the constraints, the green curve quickly learns to produce close to no violations, better than the other curves.

5.4 With domain restriction

Now if we compare with the right part of Table 2, we see that adding domain restriction seems to have a mixed effect on both performance metrics. Only for lines 1, 2 and 7, we can see some improvement. What is interesting is that these three reward functions are the functions with only one of our three components (rnn, marginals and violations). It seems as if domain restriction has a good impact when the reward function is simple, but is harmful when the complexity increases.

Another interesting fact is that generally both metrics evolve in the same direction. Indeed, when we compare both halves of Table 2 on the same line, there is only one occurrence of the constraint satisfaction increasing and the rnn reward decreasing. This shows an alignment between the corpus and the constraints.

5.5 Comparing constraints

Figure 4 shows the number of violations during all 50 000 iterations, for each constraint enforced, averaged over 10 seeds. We are mostly interested in the green curve, representing the `rnn + marginals + violations` reward function. As we can see, the first seven constraints and the last constraint have a very similar behavior. Indeed, the number of violations drops close to 0 at around step 100 (20 000th iteration). However, constraints 8 to 11 seem to be a lot harder to learn. This can be explained by the fact that these constraints create violations only at the end of the sequence. For two of these four constraints, the best reward function

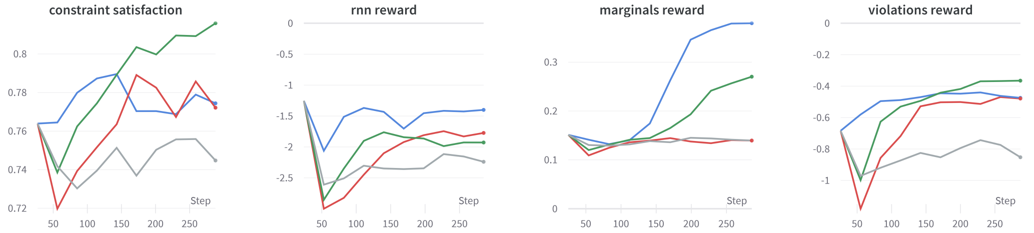


Figure 5 Evolution of our four metrics with different reward functions (red = rnn + violations, blue = rnn + marginals, green = rnn + marginals + violations, gray = rnn). The domains were not restricted. The origin of each curve shows the initial value before training the DDQN. *Takeaway:* All of the CP curves improve on the gray curve for all the metrics. The green curve achieves the best constraint satisfaction and the best violations reward.

is the `rnn` itself, which could mean that constraints only having an impact at the end are easier to learn from data than by using CP. We can also see that the green curve yields fewer violations for 9 out of 12 constraints.

The unexpected increase of the blue curve for constraint avoidSixths could be a sign that an agent keeping track only of the marginals doesn't hesitate to break more constraints when the marginals are already 0. That could explain why this behavior is not seen for reward functions that take into account the number of violations.

5.6 Comparing rewards

Figure 5 shows the evolution of the constraint satisfaction and the three rewards averaged over 10 seeds. By looking at these plots, it is obvious that adding CP outputs improves all four metrics. The great improvement for the constraint satisfaction, the marginals reward and the violations rewards is not surprising since the CP outputs are added to enforce constraints. However, even the rnn reward is increasing, showing that satisfying constraints also improves the capacity of the RL agent to reflect what the RNN learned from the corpus. We can also see that the `rnn + marginals + violations` function (green curve) is the winner for half of the metrics.

6 Discussion and future work

Even though our algorithm is an extension of the RL-Tuner, our results cannot be compared directly with those in the original paper. This is mainly due to our choice of corpus and constraints. However, the violations returned by the CP model are similar in spirit to evaluating each constraint one by one as is done in the original paper. We can then consider the `rnn + violations` reward function as reflecting what the original paper would have obtained, had the authors used the same RNN and constraints. If we use this reward function as a baseline for comparison, we see that the marginals provide an advantage and that the best performance is achieved by combining them with violations. Furthermore, we see that most of our constraints reach 0 violations in the first 20 000 iterations which decreases the number of steps required for convergence (more than 50 000 iterations for the red curve).

Experiments on the number of BP iterations required for convergence of the marginals could be useful to increase the speed of the CP model. As of now, it takes around 500 ms to compute either the marginals or the violations for each partial note sequence. The full training process for all 50 000 iterations takes a few hours. To increase efficiency, we store a list of marginals and violations to reuse them. With the marginals and violations already computed, the training process takes a few minutes. Unfortunately the length of the sequences prevents us from precomputing them for each possible note sequence.

It would also be relevant to evaluate our algorithm with a human study (e.g. Turing test or Likert scale). Given that our task was to generate melodic lines, we could not create samples without rhythmic information. Ongoing work adds rhythmic constraints to convert melodic lines to note sequences with pitches and durations. This will allow us to generate samples and conduct human studies.

We are aware that the performance of our RNN is not nearly as high as that of the one used in the RL-Tuner paper. However, our goal here was to show that the RL agent is able to combine the RNN's predictions with the constraints, and this goal is not affected by the initial RNN performance. To evaluate the aesthetic quality of generated sequences, we would need to have an RNN with a higher accuracy.

We also plan to investigate such a combination of ML and CP for sequence generation tasks in other application domains.

7 Conclusion

In this work, we presented an extension of the RL-Tuner algorithm: adding the output of CP models to the reward function in order to learn hard constraints. We applied our algorithm to the generation of melodic lines. We showed that combining the pretrained RNN's probability with the marginals and the number of constraint violations yields the best constraint satisfaction, while increasing the rnn reward obtained by the model without CP. This shows that combining ML and CP yields generated sequences that are better at both reflecting the corpus and respecting constraints than using only ML (or CP). We also studied each constraint individually and showed that for most of the constraints, the best reward function quickly converged to no violations. Our work allowed us to generate melodic lines that respect constraints without forgetting structural knowledge obtained from the Bach corpus.

References

1 Jean-Pierre Briot, Gaëtan Hadjeres, and François Pachet. Deep learning techniques for music generation - A survey. *CoRR*, abs/1709.01620, 2017. arXiv:1709.01620.

2 Sophie Demassey, Gilles Pesant, and Louis-Martin Rousseau. A Cost-Regular based Hybrid Column Generation Approach. *Constraints*, 11(4):315–333, December 2006. doi:10.1007/s10601-006-9003-7.

3 Gaëtan Hadjeres and Frank Nielsen. Anticipation-rnn: enforcing unary constraints in sequence generation, with application to interactive music generation. *Neural Comput. Appl.*, 32(4):995–1005, 2020. doi:10.1007/s00521-018-3868-4.

4 Gaëtan Hadjeres, François Pachet, and Frank Nielsen. Deepbach: a steerable model for bach chorales generation. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 1362–1371. PMLR, 2017. URL: http://proceedings.mlr.press/v70/hadjeres17a.html.

5 Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, December 1997. doi:10.1162/neco.1997.9.8.1735.

6 Cheng-Zhi Anna Huang, Tim Cooijmans, Adam Roberts, Aaron C. Courville, and Douglas Eck. Counterpoint by convolution. *CoRR*, abs/1903.07227, 2019. arXiv:1903.07227.

7 Natasha Jaques, Shixiang Gu, Richard E. Turner, and Douglas Eck. Tuning recurrent neural networks with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=Syyv2e-Kx.

8 Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *CoRR*, cs.AI/9605103, 1996. arXiv:9605103.

9 Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charlie Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

10 Olof Mogren. C-RNN-GAN: continuous recurrent neural networks with adversarial training. *CoRR*, abs/1611.09904, 2016. arXiv:1611.09904.

11 Gilles Pesant. A regular language membership constraint for finite sequences of variables. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 482–495, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

12 Gilles Pesant. From support propagation to belief propagation in constraint programming. *J. Artif. Intell. Res.*, 66:123–150, 2019. doi:10.1613/jair.1.11487.

13 Jose David Fernández Rodriguez and Francisco J. Vico. AI methods in algorithmic composition: A comprehensive survey. *CoRR*, abs/1402.0585, 2014. arXiv:1402.0585.

14 Robin M. Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview. *CoRR*, abs/1912.05911, 2019. arXiv:1912.05911.

15 Peter Schubert. *Modal Counterpoint, Renaissance Style.* Oxford University Press, 2nd edition, 2008.

16 Charlotte Truchet and Gérard Assayag, editors. *Constraint Programming in Music.* Wiley, 2011.

17 Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. arXiv:1509.06461.

18 Willem Jan van Hoeve, Gilles Pesant, and Louis-Martin Rousseau. On global warming: Flow-based soft global constraints. *J. Heuristics*, 12(4-5):347–373, 2006. doi:10.1007/s10732-006-6550-4.

19 Halley Young, Maxwell Du, and Osbert Bastani. Neurosymbolic deep generative models for sequence data with relational constraints, 2021. URL: https://openreview.net/forum?id=Y5TgO3J_Glc.

Exploiting Functional Constraints in Automatic Dominance Breaking for Constraint Optimization

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

Allen Z. Zhong
Department of Computer Science and Engineering,
The Chinese University of Hong Kong, Shatin, China

Abstract

Dominance breaking is an effective technique to reduce the time for solving constraint optimization problems. Lee and Zhong propose an automatic dominance breaking framework for a class of constraint optimization problems based on specific forms of objectives and constraints. In this paper, we propose to enhance the framework for problems with nested function calls which can be flattened to functional constraints. In particular, we focus on aggregation functions and exploit such properties as monotonicity, commutativity and associativity to give an efficient procedure for generating effective dominance breaking nogoods. Experimentation also shows orders-of-magnitude runtime speedup using the generated dominance breaking nogoods and demonstrates the ability of our proposal to reveal dominance relations in the literature and discover new dominance relations on problems with ineffective or no known dominance breaking constraints.

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

Keywords and phrases Constraint Optimization Problems, Dominance Breaking

Digital Object Identifier 10.4230/LIPIcs.CP.2022.31

Supplementary Material *Software (Source Code)*: `https://github.com/AllenZzw/auto-dom`

Funding We acknowledge the financial support of a General Research Fund (RGC Ref. No. CUHK 14206321) by the University Grants Committee, Hong Kong.

Acknowledgements We are grateful to the anonymous referees of CP-22 for their useful comments and suggestions.

1 Introduction

Dominance relations [7, 18] in Constraint Optimization Problems (COPs) describe relations between two full assignments where one is known to be subordinate compared with another concerning satisfiability and/or objective value. *Dominance breaking*, which adds additional constraints to remove dominated full assignments, is known to be effective in a range of problems [1, 16, 22, 31] but also demands sophisticated insights into the problem structures. Lee and Zhong [25, 24] give the method of automatic dominance breaking for a class of COPs, which can identify dominance relations and generate dominance breaking constraints automatically. Yet, the method is restricted to COPs with only objectives and constraints that are all provably *efficiently checkable*. For example, in order to apply automatic dominance breaking to a COP, the objective is required to be either a separable function or a submodular function. This prevents the use of automatic dominance breaking for COPs with varying objectives and constraints, especially the ones with nested function calls.

Functional expressions are ubiquitous in problem modelling, while the objective and constraints with functional expressions are usually not efficiently checkable. In practice, however, COPs are usually specified in a high-level modeling language [10, 28] and normalized/flattened into a form with only standard constraints.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 31; pp. 31:1–31:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

▶ **Example 1.** Consider a simple COP which minimizes the objective $\max(z_1, z_2) + 4z_3$ subject to the constraint $2z_1 - 3z_2 * z_3 \leq 5$, where $z_1, z_2, z_3 \in \{1, 2, 3\}$. The objective with the max function and the constraint with the multiplying function are not efficiently checkable. After normalization, the COP can become:

$$\begin{aligned} &\text{minimize } obj \\ &\text{subject to } y_2 \leq 5, obj = y_1 + 4z_3, y_1 = \max(z_1, z_2), \\ &\qquad\qquad y_2 = 2z_1 - 3y_3, y_3 = z_2 * z_3, \\ &\qquad\qquad z_1, z_2, z_3 \in \{1, 2, 3\}, y_1, y_2, y_3, obj \in \mathbb{Z} \end{aligned} \tag{1}$$

Note that y_1, y_2, y_3 and obj are newly introduced variables, and are functionally defined by $y_1 = \max(z_1, z_2)$, $y_2 = 2z_1 - 3y_3$, $y_3 = z_2 * z_3$ and $obj = y_1 + 4z_3$ respectively. We call these functional constraints, while $y_2 \leq 5$ is a non-functional constraint.

In this paper, we propose to exploit functional constraints to identify useful dominance relations in COPs with nested function calls. We first generalize the theory of dominance to normalized COPs which contain functionally defined variables and functional constraints. We present a method for automatic derivation of sufficient conditions for dominance relations in COPs based on functional constraints and common properties such as monotonicity, commutativity and associativity. The proposed method is implemented on top of the MiniZinc compiler [28]. Experimentation on various benchmarks confirms the superior efficiency of the generated nogoods to solve problems with ineffective or no known dominance breaking constraints in the literature. Even when nogoods are costly to generate, we give two case studies on the Steel Mill Slab Design Problem [11] and the Balanced Academic Curriculum Problem [5] to show how we can discover dominance relations and compact dominance (symmetry) breaking constraints by studying the nogood patterns of small instances.

2 Background

A *variable* x is an unknown. A *domain* D maps each variable x to the finite set $D(x)$ which contains the possible values for x. An *assignment* θ *on a set of variables* $S = \{x_1, \ldots, x_k\}$ is a tuple $(v_1, \ldots, v_k) \in \mathcal{D}^S = D(x_1) \times \cdots \times D(x_k)$, where $v_j = \theta[x_j]$ is the value assigned to x_j in θ, and $S = var(\theta)$ is the *scope* of θ. We abuse notations to use $\theta[S']$ to denote the tuple formed by projecting $\theta \in \mathcal{D}^S$ onto $S' \subset S$. A *constraint* c is a subset of the Cartesian product $\mathcal{D}^S$ where $S = var(c)$ is the *scope* of c. An assignment θ *satisfies* a constraint c if $\theta[var(c)] \in c$, where $var(\theta) \supseteq var(c)$. We define a *nogood* $\neg\theta$ for an assignment θ to be a constraint of the form $\vee_{x \in var(\theta)}(x \neq \theta[x])$, and its *length* is always equal to the scope size $|var(\theta)|$. A *functional constraint* is of the form $y = h(x_1, \ldots, x_k)$ where $h : \mathcal{D}^{\{x_1,\ldots,x_k\}} \mapsto \mathcal{D}^{\{y\}}$ is a function mapping any assignment on variables $\{x_1, \ldots, x_k\}$ to a unique assignment on y.

A *Constraint Satisfaction Problem (CSP)* is a tuple (X, D, C) where X is a set of variables, D is a domain for X and C is a set of constraints. A *Constraint Optimization Problem (COP)* (X, D, C, obj) extends a CSP with an objective variable obj which is to be minimized. Let $\bar{\theta} \in \mathcal{D}^X$ denote a *full assignment* whose scope is X. A *solution* of a COP/CSP P is a full assignment $\bar{\theta} \in \mathcal{D}^X$ such that $\bar{\theta}$ satisfies all constraints $c \in C$. We let $sol(P) \subseteq \mathcal{D}^X$ denote the set of all solutions of P. The goal of solving a COP is to find an *optimal solution* $\bar{\theta}^* \in sol(P)$ such that $\bar{\theta}^*[obj] \leq \bar{\theta}[obj]$ for all solutions $\bar{\theta} \in sol(P)$, and $\bar{\theta}^*[obj]$ is the *optimal value* of P.

A *dominance relation* $\prec$ *over* $\mathcal{D}^X$ [7] is a transitive and irreflexive relation such that $\forall\bar{\theta}, \bar{\theta}' \in \mathcal{D}^X$, if $\bar{\theta} \prec \bar{\theta}'$ with respect to P, then either: (1) $\bar{\theta} \in sol(P)$ and $\bar{\theta}' \notin sol(P)$, or (2) $\bar{\theta}, \bar{\theta}' \in sol(P)$ and $\bar{\theta}[obj] \leq \bar{\theta}'[obj]$, or (3) $\bar{\theta}, \bar{\theta}' \notin sol(P)$ and $\bar{\theta}[obj] \leq \bar{\theta}'[obj]$. Dominance relations can be generalized [25] to assignments over $\mathcal{D}^S$ where $S \subseteq X$. Let $\mathcal{D}^X_\theta = \{\bar{\theta} \in \mathcal{D}^X \mid \bar{\theta}[var(\theta)] = \theta\}$ be a subset of $\mathcal{D}^X$. We say that θ *dominates* θ' with respect to P iff $\forall \bar{\theta}' \in \mathcal{D}^X_{\theta'}, \exists \bar{\theta} \in \mathcal{D}^X_\theta$ such that $\bar{\theta} \prec \bar{\theta}'$ for some dominance relation $\prec$ with respect to P. When the context is clear, we abuse notations and let $\theta \prec \theta'$ denote θ dominates θ'.

▶ **Theorem 2** ([25]). *Suppose $\theta, \theta' \in \mathcal{D}^S$ are assignments of $P = (X, D, C, obj)$ where $S \subseteq X$. If $\theta \prec \theta'$ with respect to P, then removing all assignments in $\mathcal{D}^X_{\theta'}$ preserves the same satisfiability and optimal value of P.*

Note that removing all dominated full assignments in $\mathcal{D}^X_{\theta'}$ only requires to add a nogood $\neg\theta'$ to P. While generating all dominance breaking nogoods is impractical, Lee and Zhong [25] formulate it as constraint satisfaction to identify and exploit only a subset of such nogoods. The constraints in the *generation CSPs* are sufficient conditions over pairs (θ, θ') of assignments such that $\theta \prec \theta'$ with respect to P. The key step is to derive constraints in the generation CSP automatically as sufficient conditions for dominance relations. Lee and Zhong [25] give such constraints directly based on the objective and constraint types, but it is not easily extensible especially when there are nested function calls as shown in Example 1. To tackle this problem, we generalize the theory of dominance in Section 3 and present a method for automatic sufficient condition derivation in Section 4.

3 Functional Constraints and Dominance

In this paper, we assume that *a COP $P = (X, D, C, obj)$ is the result of applying some sort of flattening procedure*, such as the one used in the MiniZinc compiler [26] and similar to that shown in Example 1, to a problem model. Therefore, we have a set C_Y of functional constraints, each defining a variable $y \in Y$, and a set of non-functional constraints. Our proposed method utilizes the functional constraints and the properties of functions to derive sufficient conditions for dominance as shown in the following example.

▶ **Example 3.** Consider the COP in (1) and $\theta, \theta' \in \mathcal{D}^S$ where $S = \{z_1, z_2\}$. Our aim is to find sufficient conditions over θ and θ' that imply all full assignments in $\mathcal{D}^X_{\theta'}$ can be removed. Suppose we only consider full assignments that satisfy all functional constraints in (1). For each full assignment $\bar{\theta}' \in \mathcal{D}^X_{\theta'}$, we focus on a corresponding $\bar{\theta} \in \mathcal{D}^X_\theta$ where $\bar{\theta}[z_3] = \bar{\theta}'[z_3]$. By definition of dominance relations, if we have (a) betterment: $\bar{\theta}[obj] \leq \bar{\theta}'[obj]$, (b) implied satisfaction: $\bar{\theta}[y_2] \leq \bar{\theta}'[y_2]$, and (c) $\theta \neq \theta'$, then $\bar{\theta}'$ is dominated by $\bar{\theta}$ and hence can be removed. We can find sufficient conditions for betterment as follows:

- Variable obj is defined by $obj = y_1 + 4z_3$. If we have $\bar{\theta}[y_1] + 4\bar{\theta}[z_3] \leq \bar{\theta}'[y_1] + 4\bar{\theta}'[z_3]$, then $\bar{\theta}[obj] \leq \bar{\theta}'[obj]$ must hold since $\bar{\theta}$ and $\bar{\theta}'$ satisfy all functional constraints.
- Variable y_1 is defined by $y_1 = \max(z_1, z_2)$. It suffices to show that

$$\max(\bar{\theta}[z_1], \bar{\theta}[z_2]) + \bar{\theta}[z_3] \leq \max(\bar{\theta}'[z_1], \bar{\theta}'[z_2]) + \bar{\theta}'[z_3]. \tag{2}$$

- The summation function is monotonically increasing, (2) must be true if we have

$$\max(\bar{\theta}[z_1], \bar{\theta}[z_2]) \leq \max(\bar{\theta}'[z_1], \bar{\theta}'[z_2]) \wedge \bar{\theta}[z_3] \leq \bar{\theta}'[z_3] \tag{3}$$

- Since $\bar{\theta} \in \mathcal{D}^X_\theta$ and $\bar{\theta}' \in \mathcal{D}^X_{\theta'}$, we have $\bar{\theta}[z_1] = \theta[z_1]$, $\bar{\theta}[z_2] = \theta[z_2]$, $\bar{\theta}'[z_1] = \theta'[z_1]$, and $\bar{\theta}'[z_2] = \theta'[z_2]$. The condition (3) is equivalent to

$$\max(\theta[z_1], \theta[z_2]) \leq \max(\theta'[z_1], \theta'[z_2]) \tag{4}$$

  Inequality $(\bar{\theta}[z_3] \leq \bar{\theta}'[z_3])$ must hold since $\bar{\theta}[z_3] = \bar{\theta}'[z_3]$. Thus, if θ and θ' satisfy (4), the betterment condition must hold.

Similarly, we can find the sufficient condition for implied satisfaction as follows:

- Variable y_2 is defined by $y_2 = 2z_1 - 3y_3$, $\bar{\theta}[y_2] \leq \bar{\theta}'[y_2]$ must be true if
$$2\bar{\theta}[z_1] \leq 2\bar{\theta}'[z_1] \wedge 3\bar{\theta}[y_3] \geq 3\bar{\theta}'[y_3] \tag{5}$$
- Variable y_3 is defined by $y_3 = z_2 * z_3$. Since $z_2, z_3 \geq 0$, $3\bar{\theta}[y_3] \geq 3\bar{\theta}'[y_3]$ must hold if
$$\bar{\theta}[z_2] \geq \bar{\theta}'[z_2] \wedge \bar{\theta}[z_3] \geq \bar{\theta}'[z_3] \tag{6}$$
Since $\bar{\theta}[z_3] = \bar{\theta}'[z_3]$, the latter condition must hold.
- By definitions, we have $\bar{\theta}[z_1] = \theta[z_1]$, $\bar{\theta}[z_2] = \theta[z_2]$, $\bar{\theta}'[z_1] = \theta'[z_1]$, and $\bar{\theta}'[z_2] = \theta'[z_2]$, and therefore (5) and (6) must hold if
$$\theta[z_1] \leq \theta'[z_1] \wedge \theta[z_2] \geq \theta'[z_2] \tag{7}$$

The generation CSP for θ and θ' contains (4) and (7). To ensure the compatibility of generated nogoods, we can follow Lee and Zhong [25] to add the lexicographic ordering constraint $(\theta[z_1], \theta[z_2]) <_{lex} (\theta'[z_1], \theta'[z_2])$. One possible solution of the generation CSP is the pair (θ, θ') where $\theta = (1, 2)$ and $\theta' = (2, 1)$, and the constraint $\neg\theta' \equiv (z_1 \neq 2 \vee z_2 \neq 1)$ is a dominance breaking nogoods in (1). Similar derivation can also be applied to pairs of assignments over other scopes to obtain more dominance breaking nogoods.

As shown in Example 3, our method identify nogoods by the following process:

1. Choose a cardinality of a scope S
2. Enumerate all possible scope S with the chosen cardinality. For each S:
   a. Derive sufficient conditions and synthesize a generation CSP for S
   b. Solve all solutions of the generation CSP
   c. Collect the derived nogoods from the solutions (one nogood from each solution)
3. Add all the collected nogoods to the COP before solving

The key step is to synthesize a generation CSP considering the functional constraints. In the following, we give a theory of dominance for normalized COPs.

For ease of presentation, we associate each non-functional constraint $c \in (C \setminus C_Y)$ with a *reified variable* $b \in \{0, 1\}$, where a full assignment $\bar{\theta}$ satisfies c iff $\bar{\theta}[b] = 0$[1]. In other words, we treat each constraint $c \in (C \setminus C_Y)$ as a function returning 0/1 and define a (reified) functional constraint $c_b \equiv (b = c(x_{i_1}, \ldots, x_{i_k}))$. If $\bar{\theta}[b] \leq \bar{\theta}'[b]$ for two full assignments $\bar{\theta}$ and $\bar{\theta}'$, then $\bar{\theta}'$ satisfies c implies that $\bar{\theta}$ also satisfies c. We let C_B denote the set of (reified) functional constraints and B denote the set of reified variables.

Without loss of generality, let (Z, Y, B) and (C_B, C_Y) be a partition of variables X and constraints C respectively in a normalized COP, where $Z \cup Y \cup B = X$, $C_B \cup C_Y = C$ and $obj \in Y$. Note that Z, Y, B are pairwise disjoint and $C_B \cap C_Y = \emptyset$. In case a variable $y \in Y$ is introduced by the flattening procedure, we set the domain for y to be the largest possible set, and therefore, a constraint $c_y \in C_Y$ must be satisfied if the value of $y \in Y$ is computed from the assignments over variables $x_{i_1}, \ldots, x_{i_k}$. Note that when there is no flattening and reification, our definition of a COP degenerates to the classical definition [34].

We say that a full assignment $\bar{\theta}$ is *functionally valid* iff (a) $\bar{\theta}[b] = c(\bar{\theta}[x_{i_1}], \ldots, \bar{\theta}[x_{i_k}])$ for $(b = c(x_{i_1}, \ldots, x_{i_k})) \in C_B$ and (b) $\bar{\theta}[y] = h(\bar{\theta}[x_{i_1}], \ldots, \bar{\theta}[x_{i_k}])$ for $(y = h(x_{i_1}, \ldots, x_{i_k})) \in C_Y$. Note that $\bar{\theta}$ in a normalized COP will correspond to a full assignment in the original non-flattened problem model iff $\bar{\theta}$ is functionally valid. The value for $y \in Y$ (respectively $b \in B$)

[1] Note that this is different from the convention that 0 means false and 1 means true.

in a functionally valid full assignment are determined by $c_y \in C_Y$ (respectively $c_b \in C_B$) as well as values for variables in Z. We define the set of *determining variables* $A(x) \subseteq Z$ of a variable $x \in X$ is

- $A(z) = \{z\}$ for a variable $z \in Z$,
- $A(y) = \cup_{x \in var(c_y) \setminus y} A(x)$ for a variable $y \in Y$, and
- $A(b) = \cup_{x \in var(c_b) \setminus b} A(x)$ for a variable $b \in B$.

In the remainder of the paper, we assume that $P = (X, D, C, obj)$ is a normalized COP and consider only functionally valid full assignments in $\mathcal{D}^X$. Our aim is to find sufficient conditions for a pair of assignments θ and θ' over $S \subseteq Z$ such that $\theta \prec \theta'$ with respect to P. Recall that $\theta \prec \theta'$ requires $\forall \bar{\theta}' \in \mathcal{D}^X_{\theta'}, \exists \bar{\theta} \in \mathcal{D}^X_{\theta}$ such that $\bar{\theta} \prec \bar{\theta}'$ for some dominance relation over $\mathcal{D}^X$. It is expensive to check whether *there exists* $\bar{\theta}$ for each $\bar{\theta}'$ in $\mathcal{D}^X_{\theta'}$. Instead, we propose to check only if a specific $\bar{\theta}$ dominates $\bar{\theta}'$ by utilizing a *mutation mapping* for two assignments θ and θ' over the same scope.

▶ **Definition 4.** *The* mutation mapping $\mu^{\theta \to \theta'}$ *for two assignments θ, θ' over the scope S maps a full assignment $\bar{\theta} \in \mathcal{D}^X_{\theta}$ to another full assignment $\bar{\theta}' \in \mathcal{D}^X_{\theta'}$ such that:*

- $\bar{\theta}'[z] = \theta'[z]$ *for* $z \in var(\theta)$,
- $\bar{\theta}'[z] = \bar{\theta}[z]$ *for* $z \in Z \setminus var(\theta)$,
- $\bar{\theta}'[y] = h(\bar{\theta}'[x_{i_1}], \ldots, \bar{\theta}'[x_{i_k}])$ *where* $y \in Y$ *is defined by* $y = h(x_{i_1}, \ldots, x_{i_k}) \in C_Y$,
- $\bar{\theta}'[b] = c(\bar{\theta}'[x_{i_1}], \ldots, \bar{\theta}'[x_{i_k}])$ *where* $b \in B$ *is defined by* $b = c(x_{i_1}, \ldots, x_{i_k}) \in C_B$.

In other words, $\mu^{\theta \to \theta'}$ "mutates" the θ component of $\bar{\theta}$ to become θ' and assigns the values of variables in $Y \cup B$ accordingly. The mutation mapping $\mu^{\theta \to \theta'}$ is a bijection, and thus the inverse mapping $(\mu^{\theta \to \theta'})^{-1} = \mu^{\theta' \to \theta}$ always exists. The following proposition characterizes some useful properties of the mutation mapping.

▶ **Proposition 5.** *The followings are true for all full assignments $\bar{\theta} \in \mathcal{D}^X_{\theta}$ and $\bar{\theta}' = \mu^{\theta \to \theta'}(\bar{\theta})$:*

- *If $z \in S$, then $\bar{\theta}[z] = \theta[z]$ and $\bar{\theta}'[z] = \theta'[z]$.*
- *If $z \in Z \setminus S$, then $\bar{\theta}[z] = \bar{\theta}'[z]$.*

The following result gives a sufficient condition governing when $\theta \prec \theta'$ with respect to P.

▶ **Theorem 6.** *If a pair of assignments $\theta, \theta' \in \mathcal{D}^S$ satisfies:*

- *empty intersection: $\mathcal{D}^X_{\theta} \cap \mathcal{D}^X_{\theta'} = \emptyset$,*
- *betterment: $\forall \bar{\theta} \in \mathcal{D}^X_{\theta}, \bar{\theta}[obj] \leq \mu^{\theta \to \theta'}(\bar{\theta})[obj]$, and*
- *implied satisfaction: $\forall b \in B, \forall \bar{\theta} \in \mathcal{D}^X_{\theta}, \bar{\theta}[b] \leq \mu^{\theta \to \theta'}(\bar{\theta})[b]$,*

then θ dominates θ' with respect to P.

Theorems 2 and 6 imply that $\neg\theta'$ is a *dominance breaking nogood* to remove all dominated solution in $\mathcal{D}^X_{\theta'}$. To further ensure that all generated nogoods are *compatible* in the sense that not all optimal solutions of P are eliminated, a lexicographic ordering constraint $\theta <_{lex} \theta'$ is also added to the generation CSP [25]. What remains is to find constraints over θ and θ' that are sufficient conditions for empty intersection, betterment and implied satisfaction. Empty intersection is trivially satisfied if $\theta \neq \theta'$. In Section 4, we will focus on finding sufficient conditions for betterment and implied satisfaction. Note that the above definitions and results degenerate to those by Lee and Zhong [25] when Y and C_Y are empty.

4 Automatic Sufficient Condition Derivation

In this section, we describe formally how functional constraints and their properties are used for deriving effective sufficient conditions for betterment and implied satisfaction, which are predicates requiring an inequality to hold for all $\bar{\theta} \in \mathcal{D}^X_{\theta}$. When it is clear from the context,

Algorithm 1 Deriving sufficient conditions for betterment and implied satisfaction.

```
$Q \leftarrow \{(\forall\bar{\theta} \in \mathcal{D}_{\theta}^{X}, \bar{\theta}[obj] \leq \bar{\theta}'[obj])\} \cup \{(\forall\bar{\theta} \in \mathcal{D}_{\theta}^{X}, \bar{\theta}[b] \leq \bar{\theta}'[b]) \mid b \in B\}$, $F \leftarrow \emptyset$
while $Q \neq \emptyset$ do
    Remove a predicate $p \equiv (\forall\bar{\theta} \in \mathcal{D}_{\theta}^{X}, t\bar{\theta} \lhd t\bar{\theta}')$ from $Q$
    if $var(t) \cap (Y \cup B) \neq \emptyset$ then
        Let $x \in var(t) \cap (Y \cup B)$ be a variable defined by $x = f(x_{i_1}, \ldots, x_{i_k})$
        $\beta \leftarrow \{x/f(x_{i_1}, \ldots, x_{i_k})\}$
        $Q \leftarrow Q \cup \{(\forall\bar{\theta} \in \mathcal{D}_{\theta}^{X}, (t\beta)\bar{\theta} \lhd (t\beta)\bar{\theta}')\}$        // Replacement
    else if $var(t) \subseteq S \subseteq Z$ then
        $F \leftarrow F \cup \{(t\theta \lhd t\theta')\}$        // Binding
    else if $var(t) \subseteq (Z \setminus S)$ then
        Continue        // Deletion
    else
        Let $p$ be $(\forall\bar{\theta} \in \mathcal{D}_{\theta}^{X}, f(t_1\bar{\theta}, \ldots, t_k\bar{\theta}) \lhd f(t_1\bar{\theta}', \ldots, t_k\bar{\theta}'))$
        $Q \leftarrow Q \cup \{(\forall\bar{\theta} \in \mathcal{D}_{\theta}^{X}, t_i\bar{\theta} = t_i\bar{\theta}') \mid \forall i \in \{1, \ldots, k\}\}$        // General Decomposition
    end if
end while
return $F$
```

we let $\bar{\theta}' = \mu^{\theta \rightarrow \theta'}(\bar{\theta})$ denote the image by the mutation mapping of θ and θ'. We first present a general algorithm which only utilizes functional constraints, followed by rules that exploits the functional properties of monotonicity, associativity and commutativity.

4.1 General Decomposition

To formalize the derivation of sufficient conditions, we use the inductive definition of *terms* [2]:

- a variable x is a term, and
- if f is a k-ary function and $t_1, \ldots, t_k$ are terms, then $f(t_1, \ldots, t_k)$ is a term.

By abusing notations, we define $var(t) = \{x\}$ when $t \equiv x$, and $var(t) = \cup_{i=1}^{k} var(t_i)$ when $t \equiv f(t_1, \ldots, t_k)$. Note that f can either be the constraint c in a reified constraint $b = c(x_{i_1}, \ldots, x_{i_k})$ or the function h in a functional constraint $y = h(x_{i_1}, \ldots, x_{i_k})$. We say that a variable x is *fixed* in an assignment θ when the set of determining variables $A(x)$ of x is a subset of $var(\theta)$. A term t is *fixed in* θ iff all variables of t are fixed in θ, i.e., $\wedge_{x \in var(t)}(A(x) \subseteq var(\theta))$; otherwise t is a *free*.

A *substitution* is a finite mapping from variables to terms which assigns to each variable x a term t different from x. We write a substitution as $\beta = \{x_{i_1}/t_1, \ldots, x_{i_k}/t_k\}$ where $x_{i_1}, \ldots, x_{i_k}$ are different variables, and $t_1, \ldots, t_k$ are terms such that $\forall j \in \{1, \ldots, k\}, x_{i_j} \not\equiv t_j$. A substitution β can be *applied to a term* t to obtain $t\beta$ by replacing every occurrence of variable x_{i_j} in t by the term t_j for all $j \in \{1, \ldots, k\}$.

Let $\lhd$ be an operator in $\{\leq, \geq, =\}$. The betterment and implied satisfaction condition in Theorem 6 are predicates in the form of quantified inequalities, i.e., $(\forall\bar{\theta} \in \mathcal{D}_{\theta}^{X}, t\bar{\theta} \lhd t\bar{\theta}')$, where $t\bar{\theta}$ and $t\bar{\theta}'$ are obtained by substituting each variable in t with its values in $\bar{\theta}$ and $\bar{\theta}'$ respectively. Algorithm 1 shows an automatic process that derives predicates as sufficient conditions for the betterment and implied satisfaction, and all variables in the sufficient conditions are fixed in θ and θ'. The algorithm maintains two sets of predicates Q and F, where Q consists of predicates that have to be further processed, and F is the set of predicates whose variables are all in S. The while loop continues until Q is empty and all predicates has been processed. In each iteration of the while loop, a predicate is removed from Q and processed by *replacement* (lines 4 to 7), *binding* (lines 8 to 9), *deletion* (lines 10 to 11) and *general decomposition* (lines 12 to 14) rules respectively. Finally, F is returned for synthesizing the generation CSP. Note that t must be a function term of the form $f(t_1, \dots, t_k)$ in line 13, since $var(t)$ is a subset of Z and $var(t)$ has non-empty intersection with both S and $Z \setminus S$. The following theorem states an important property of Algorithm 1. For simplicity, let $Q \wedge F$ denote the conjunction of all predicates in Q and F.

▶ **Theorem 7.** *Algorithm 1 preserves the invariant that $Q \wedge F$ is always a sufficient condition for betterment and implied satisfaction of P.*

Proof. Since Q is initialized with the betterment and implied satisfaction, the statement holds at the beginning. By induction, it suffices to show that when $Q \wedge F$ is still a sufficient condition after an iteration in the while loop. There are four rules in the iteration:

- Replacement: the predicate $\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t\bar{\theta} \lhd t\bar{\theta}'$ is equivalent to $\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, (t\beta)\bar{\theta} \lhd (t\beta)\bar{\theta}'$, because we assume that full assignments are all functionally valid.
- Binding: the predicate $(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t\bar{\theta} \lhd t\bar{\theta}')$ is equivalent to $(t\theta \lhd t\theta')$ by Proposition 5, since a variable x in $var(t)$ also belongs to $S \subseteq Z$, and we have $\bar{\theta}[x] = \theta[x]$ and $\bar{\theta}'[x] = \theta'[x]$.
- Deletion: by Proposition 5 again, $\forall x \in var(t)$ where $x \in Z$ and $x \notin S$, we have $\bar{\theta}[x] = \bar{\theta}'[x]$. Therefore, the predicate $t\bar{\theta} = t\bar{\theta}'$ must hold and imply that $t\bar{\theta} \leq t\bar{\theta}'$ and $t\bar{\theta} \geq t\bar{\theta}'$.
- General decomposition: the conjunction $\wedge_{i=1}^{k} (\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t_i\bar{\theta} = t_i\bar{\theta}')$ implies $(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, f(t_1\bar{\theta}, \dots, t_k\bar{\theta}) \lhd f(t_1\bar{\theta}', \dots, t_k\bar{\theta}'))$ by definitions of functional and reified constraints.

Therefore, the invariant is preserved in Algorithm 1. ◀

Note that the replacement, binding and deletion rules are equivalent transformation of predicates, while general decomposition replaces $p \equiv (\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, f(t_1\bar{\theta}, \dots, t_k\bar{\theta}) \lhd f(t_1\bar{\theta}', \dots, t_k\bar{\theta}'))$ into a set of predicates that are sufficient conditions for p.

▶ **Theorem 8.** *Algorithm 1 always terminates.*

Proof. Without loss of generality, we assume that each variable $y \in Y$ appears only in at most one constraint other than the functional constraint $y = h(x_{i_1}, \dots, x_{i_k})$. By definition of a COP, $Y \cup B$ and $C_Y \cup C_B$ are finite sets. We maintain three natural numbers:

- v_1: the number of variables in $Y \cup B$ that have not been substituted in replacement,
- v_2: the number of occurrences of function symbols in Q, and
- v_3: the sum of $|var(t)|$ for all predicates $(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t\bar{\theta} \lhd t\bar{\theta}') \in Q$.

We claim that applying each rule reduces the triple (v_1, v_2, v_3) in a lexicographic sense. Indeed, each variable $x \in Y \cup B$ is only substituted when x is in the flattened constraint or the reified constraint, replacement must decrease v_1 by 1. General decomposition decreases v_2 while keeping v_1 unchanged. Further, binding and deletion remove one predicate $(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t\bar{\theta} \lhd t\bar{\theta}')$ from Q and therefore decrease v_3 by $var(t)$. The termination follows from the fact that there is no infinite descending sequence of triples of natural numbers. ◀

The set Q is empty upon termination, and the following corollary is a direct consequence of Theorems 7 and 8.

▶ **Corollary 9.** *When Algorithm 1 terminates, the conjunction of predicates in F is a sufficient condition for the betterment and implied satisfaction of P.*

In other words, Algorithm 1 is *sound* in the sense that F consists of predicates that are sufficient conditions for betterment and implied satisfaction in Theorem 6. The general decomposition considers that a function f is general without any properties, but this may result in too restrictive sufficient conditions. For example, if we use Algorithm 1 for the

COP in (1), the resulting sufficient conditions for betterment and implied satisfaction will be $\theta[z_1] = \theta'[z_1]$ and $\theta[z_2] = \theta'[z_2]$, which is in conflict with the empty intersection condition in Theorem 6. No solution can be found by solving the generation CSP, and no nogoods can be generated. Therefore, we want more relaxed sufficient conditions as far as possible.

We say that a predicate α is *stronger than* another predicate β iff $\alpha \Rightarrow \beta$, and β is *weaker than* α. The weaker the sufficient conditions in the generation CSP, the more pairs of assignments will satisfy all the conditions and the more nogoods can be found by Theorem 6. The idea is to apply different decomposition rules to derive weaker sufficient conditions based on properties of functions in functional and reified constraints. Aggregation functions [17], such as summation, maximum and minimum, combine multiple values into a single representative value. They are common in modeling COPs and enjoy useful properties such as *monotonicity*, *commutativity* and *associativity*.

4.2 Decomposition for Monotonic Functions

The first property of interest is *monotonicity*. A function $f : \mathbb{R}^k \mapsto \mathbb{R}$ is *monotonically increasing* if $(\forall i, a_i \leq b_i) \Rightarrow f(a_1, \ldots, a_k) \leq f(b_1, \ldots, b_k)$ and is *monotonically decreasing* if $(\forall i, a_i \geq b_i) \Rightarrow f(a_1, \ldots, a_k) \geq f(b_1, \ldots, b_k)$ where $a_i, b_i \in \mathbb{R}$. We also define the *reverse operators* of $\leq$, $\geq$ and $=$ to be $\geq$, $\leq$ and $=$ respectively. When the function f is monotonically increasing or monotonically decreasing, we have the following rules.

▶ **Definition 10.** *Let* $p \equiv (\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, f(t_1\bar{\theta}, \ldots, t_k\bar{\theta}) \lhd f(t_1\bar{\theta}', \ldots, t_k\bar{\theta}'))$ *be a predicate where* $\lhd \in \{\leq, \geq, =\}$ *is a comparison operator and* $\rhd$ *is the reverse operator of* $\lhd$*.*

- *Increasing decomposition: when* f *is monotonically increasing, the predicate* p *is replaced by* $\{(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t_i\bar{\theta} \lhd t_i\bar{\theta}') \mid \forall i \in \{1, \ldots, k\}\}$*.*
- *Decreasing decomposition: when* f *is monotonically decreasing, the predicate* p *is replaced by* $\{(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t_i\bar{\theta} \rhd t_i\bar{\theta}') \mid \forall i \in \{1, \ldots, k\}\}$*.*

Recall that all full assignments are functionally valid. The following theorem follows directly the definition of monotonically increasing and monotonically decreasing functions.

▶ **Theorem 11.** *The increasing decomposition and decreasing decomposition rules preserve that* $Q \wedge F$ *is a sufficient condition for betterment and implied satisfaction of* P*.*

Utilizing the property of monotonicity, increasing decomposition and decreasing decomposition can return weaker sufficient conditions than general decomposition.

▶ **Theorem 12.** *The conjunction of predicates in* $\{(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t_i\bar{\theta} \lhd t_i\bar{\theta}') \mid \forall i \in \{1, \ldots, k\}\}$ *(respectively* $\{(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t_i\bar{\theta} \rhd t_i\bar{\theta}') \mid \forall i \in \{1, \ldots, k\}\}$*) is weaker than the conjunction of predicates* $\{(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t_i\bar{\theta} = t_i\bar{\theta}') \mid \forall i \in \{1, \ldots, k\}\}$ *returned by general decomposition.*

Proof. A predicate $(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t_i\bar{\theta} = t_i\bar{\theta}')$ is always a sufficient condition for both $(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t_i\bar{\theta} \lhd t_i\bar{\theta}')$ and $(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t_i\bar{\theta} \rhd t_i\bar{\theta}')$. ◀

By Theorems 11 and 12, rules in Definition 10 can obtain a weaker sufficient condition for the betterment and implied satisfaction, and the general decomposition in Algorithm 1 should be replaced by them whenever possible.

4.3 Decomposition for Associative and Commutative Functions

In this section, we take advantage of associativity and commutativity so that the general, decreasing and increasing decomposition can obtain even weaker sufficient conditions. An aggregation function [17] f can take an arbitrary non-zero number of arguments, and we use a special notation to denote it. Let $\mathbf{t} = \langle t_1, \ldots, t_k \rangle$, $\mathbf{t}_1 = \langle t_1, \ldots, t_j \rangle$ and $\mathbf{t}_2 = \langle t_{j+1}, \ldots, t_k \rangle$ where $1 \leq j \leq k$, and the followings denote the same function call: $f(t_1, \ldots, t_k)$, $f(\mathbf{t})$ and $f(\mathbf{t}_1, \mathbf{t}_2)$. Two common properties of aggregation functions are:

- *Commutativity*: $f(t_1, \ldots, t_k) = f(t_{i_1}, \ldots, t_{i_k})$ where $\{1, \ldots, k\} = \{i_1, \ldots, i_k\}$.
- *Associativity*: $f(t) = t$ and $f(\mathbf{t}_1, \mathbf{t}_2) = f(f(\mathbf{t}_1), \mathbf{t}_2)$.

By commutativity, we can always permutate the arguments of $f(\mathbf{t})$ so that all fixed terms are clustered together.

▶ **Proposition 13.** *Let f be a commutative function and $\theta \in \mathcal{D}^S$ be an assignment where $S \subseteq Z$. If there are $l \geq 1$ fixed terms among $t_1, \ldots, t_k$, then we can always find a permutation $\{i_1, \ldots, i_k\} = \{1, \ldots, k\}$ and a partition $\mathbf{t}_1 = \langle t_{i_1}, \ldots, t_{i_l} \rangle$ and $\mathbf{t}_2 = \langle t_{i_{l+1}}, \ldots, t_{i_k} \rangle$ such that $f(\mathbf{t}) = f(\mathbf{t}_1, \mathbf{t}_2)$ and all fixed arguments in $\mathbf{t}$ are in $\mathbf{t}_1$.*

The proof directly follows the definition of commutativity. We have the following rule for a commutative and associative aggregation function.

▶ **Definition 14.** *Let $p \equiv (\forall \bar{\theta} \in \mathcal{D}^X_\theta, f(t_1\bar{\theta}, \ldots, t_k\bar{\theta}) \lhd f(t_1\bar{\theta}', \ldots, t_k\bar{\theta}'))$ be a predicate such that there are $l \geq 1$ fixed terms among $t_1, \ldots, t_k$.*

- *Aggregation: when f is commutative and associative, p is replaced by the predicate $(\forall \bar{\theta} \in \mathcal{D}^X_\theta, f(f(\mathbf{t}_1), \mathbf{t}_2)\bar{\theta} \lhd f(f(\mathbf{t}_1), \mathbf{t}_2)\bar{\theta}')$, where all fixed terms are in $\mathbf{t}_1$.*

The following theorem follows directly that aggregation is an equivalent transformation.

▶ **Theorem 15.** *The aggregation rule preserves the invariant that $Q \wedge F$ is a sufficient condition for the betterment and the implied satisfaction conditions.*

The aggregation rule allows decomposition to obtain weaker sufficient conditions.

▶ **Theorem 16.** *Let f be a commutative and associative function. Suppose $p \equiv (\forall \bar{\theta} \in \mathcal{D}^X_\theta, f(\mathbf{t}_1, \mathbf{t}_2)\bar{\theta} \lhd f(\mathbf{t}_1, \mathbf{t}_2)\bar{\theta}')$ and $p' \equiv (\forall \bar{\theta} \in \mathcal{D}^X_\theta, f(f(\mathbf{t}_1), \mathbf{t}_2)\bar{\theta} \lhd f(f(\mathbf{t}_1), \mathbf{t}_2)\bar{\theta}')$ such that all fixed terms are in $\mathbf{t}_1$. The conjunction of predicates resulting from applying general decomposition to p is weaker than that to p'.*

Proof. After general decomposition, p is replaced by $\{(\forall \bar{\theta} \in \mathcal{D}^X_\theta, t_{i_j}\bar{\theta} = t_{i_j}\bar{\theta}') \mid \forall j \in \{1, \ldots, k\}\}$, while p' is replaced by $\{(\forall \bar{\theta} \in \mathcal{D}^X_\theta, f(\mathbf{t}_1)\bar{\theta} = f(\mathbf{t}_1)\bar{\theta}')\} \cup \{(\forall \bar{\theta} \in \mathcal{D}^X_\theta, t_{i_j}\bar{\theta} = t_{i_j}\bar{\theta}') \mid \forall j \in \{l+1, \ldots, k\}\}$. The claim directly follows from the fact that the conjunction of $\{(\forall \bar{\theta} \in \mathcal{D}^X_\theta, t_{i_j}\bar{\theta} = t_{i_j}\bar{\theta}') \mid \forall j \in \{1, \ldots, l\}\}$ implies $(\forall \bar{\theta} \in \mathcal{D}^X_\theta, f(\mathbf{t}_1)\bar{\theta} = f(\mathbf{t}_1)\bar{\theta}')$ since all full assignments are functionally valid. ◀

Similar results can also be proved for the increasing decomposition and decreasing decomposition rules in Definition 10. Algorithm 2 gives decomposition rules considering the properties of monotonicity, associativity and commutativity.

▶ **Example 17.** Suppose we want to find sufficient conditions for θ and θ' where $var(\theta) = var(\theta') = \{z_1, z_3\}$. Let $p \equiv (\forall \bar{\theta} \in \mathcal{D}^X_\theta, \min(\bar{\theta}[z_1], \bar{\theta}[z_2], \bar{\theta}[z_3]) \leq \min(\bar{\theta}'[z_1], \bar{\theta}'[z_2], \bar{\theta}'[z_3]))$ be a predicate in Q. If we apply increasing decomposition to p directly, then we get

$$\{(\forall \bar{\theta} \in \mathcal{D}^X_\theta, \bar{\theta}[z_i] \leq \bar{\theta}'[z_i]) \mid i \in \{1, 2, 3\}\} \tag{8}$$

Since the function min is commutative and associative, we can obtain

$$\{\forall \bar{\theta} \in \mathcal{D}^X_\theta, \min(\min(\bar{\theta}[z_1], \bar{\theta}[z_3]), \bar{\theta}[z_2]) \leq \min(\min(\bar{\theta}'[z_1], \bar{\theta}'[z_3]), \bar{\theta}'[z_2])\} \tag{9}$$

by the aggregation rule. Then by increasing decomposition, we get

$$\{(\forall \bar{\theta} \in \mathcal{D}^X_\theta, \min(\bar{\theta}[z_1], \bar{\theta}[z_3]) \leq \min(\bar{\theta}'[z_1], \bar{\theta}'[z_3])), (\forall \bar{\theta} \in \mathcal{D}^X_\theta, \bar{\theta}[z_2] \leq \bar{\theta}'[z_2])\} \tag{10}$$

Algorithm 2 New decomposition rules.

```
 1: Let $\mathbf{t}_1, \mathbf{t}_2$ be a partition of arguments in $\mathbf{t}$ where all fixed terms are in $\mathbf{t}_1$
 2: if $f$ is commutative and associative and $|\mathbf{t}_1| \neq 0$ then
 3:     $p \leftarrow (\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, f(f(\mathbf{t}_1), \mathbf{t}_2)\bar{\theta} \lhd f(f(\mathbf{t}_1), \mathbf{t}_2)\bar{\theta}')$    // Aggregation
 4: end if
 5: Let $p$ be $(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, f(t_1\bar{\theta}, \dots, t_{k'}\bar{\theta}) \lhd f(t_1\bar{\theta}', \dots, t_{k'}\bar{\theta}'))$
 6: if $f$ is monotonically increasing then
 7:     $Q \leftarrow Q \cup \{(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t_i\bar{\theta} \lhd t_i\bar{\theta}') \mid \forall i \in \{1, \dots, k'\}\}$    // Increasing decomposition
 8: else if $f$ is monotonically decreasing then
 9:     $Q \leftarrow Q \cup \{(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t_i\bar{\theta} \rhd t_i\bar{\theta}') \mid \forall i \in \{1, \dots, k'\}\}$    // Decreasing decomposition
10: else
11:     $Q \leftarrow Q \cup \{(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, t_i\bar{\theta} = t_i\bar{\theta}') \mid \forall i \in \{1, \dots, k'\}\}$    // General decomposition
12: end if
```

Note that after applying binding and deletion to (8) and (10) respectively, we obtain $\theta[z_1] \leq \theta'[z_1] \wedge \theta[z_3] \leq \theta'[z_3]$ and $\min(\theta[z_1], \theta[z_3]) \leq \min(\theta'[z_1], \theta'[z_3])$ as sufficient conditions for p, and the former condition is stronger than the latter one.

The new derivation algorithm replaces line 14 in Algorithm 1 with Algorithm 2, and it has the following properties.

▶ **Theorem 18.** *The new algorithm always terminates.*

Proof. We define a tuple (v_1, v_2, v_3) which is the same as that of Theorem 8 except that v_2 is now the number of *free function terms* in Q. The values v_1 and v_3 decrease similarly, while we argue that v_2 also decreases in the decomposition in Algorithm 2.

- When f is not commutative and associative, decomposition in lines 5 to 11 will decrease the number of function terms, and hence v_2 is also reduced.
- When the function f is commutative and associative, the predicate p is written into $(\forall \bar{\theta} \in \mathcal{D}_{\theta}^{X}, f(f(\mathbf{t}_1), \mathbf{t}_2)\bar{\theta} \lhd f(f(\mathbf{t}_1), \mathbf{t}_2)\bar{\theta}')$ by the aggregation rule, where $f(f(\mathbf{t}_1), \mathbf{t}_2)$ is free and $f(\mathbf{t}_1)$ is fixed. The follow-up decomposition in lines 5 to 11 in Algorithm 2 will decrease v_2 while keeping v_1 unchanged.

Hence, the termination follows directly from the fact that there is no infinite decreasing sequence of triples of natural numbers. ◀

▶ **Theorem 19.** *When the new algorithm terminates, the conjunction of predicates in F is a sufficient condition for the betterment and implied satisfaction of P.*

Proof. By Theorem 7, replacement, binding, deletion and general decomposition all preserve the invariant that $Q \wedge F$ is a sufficient condition for the betterment and implied satisfaction. Algorithm 2 also preserve the invariant by Theorems 11 and 15. The statement holds since Q must be empty upon termination. ◀

We note that the *alldifferent* constraint [32], which enforces variables taking distinct values, is commutative but not associative or monotonic. Rules in Definitions 10 and 14 are not applicable at first glance. We can decompose *alldifferent*$(x_{i_1}, \dots, x_{i_k})$ into a set of constraints $d_v = \sum_{j=1}^{k} bool2int(x_{i_j} = v)$ and $d_v \leq 1$, one for each value $v \in \cup_{j=1\dots,k} D(x_{i_j})$. The variable d_v is a newly introduced variable whose value is the number of variables in $\{x_{i_1}, \dots, x_{i_k}\}$ assigned value v. After decomposition, the set of constraints enjoys the properties of monotonicity, commutativity and associativity, and thus rules in Definitions 10 and 14 are now applicable. The idea can be applied similarly to support other global constraints like the global cardinality constraint [33, 30] and the bin packing constraint [35]. Note that global constraints are decomposed only in synthesizing the generation CSPs, and are untouched in problem solving.

5 Experimental Evaluation

In this section, we give experimental results to show the utility of our proposal. All experiments are run on Xeon E5-2630 2.60GHz processors. We use MiniZinc [28] as the high-level modeling language and implement our nogood generation method by modifying[2] the publicly available MiniZinc compiler with version 2.6.2. In a compiled model, we treat constraints with the annotation "`defines_var`" as functional constraints, while others are non-functional constraints that should be reified. The generated nogoods for each problem are output as text and then appended to the MiniZinc model of the corresponding problem.

The augmented models are submitted to MiniZinc for solving using the Chuffed solver [29] with version 0.10.4. Note that our method aims to analyze a user-defined model, not necessarily that of the best model, and we specify the search strategies for all problems to demonstrate the effect of the additional dominance breaking nogoods in search space pruning. We use six benchmark problems, 20 random instances for each problem size. The models for the following three problems are from public benchmark suites:

- *Talent-n*. The Talent Scheduling Problem [6] is problem 039 in CSPLib [15]. Each actor appears in several scenes and is paid a fixed cost per day if they are present. They need to be present on location from the first scene they are in till the last scene they are in. We need to schedule n scenes to minimize the total cost for a set of actors. The dominance breaking constraints for **manual** are by Chu and Stuckey [7].
- *Warehouse-n*. The Warehouse Location Problem [37] is problem 034 in CSPLib [15]. We need to choose a subset of possible warehouses in different locations to supply a set of n existing customers such that the sum of building costs for warehouses and supply costs for customers is minimized.
- *Team-n-m*. The Team Assignment Problem appears in MiniZinc Challenge 2018 [36]. The problem consists of $n \times m$ players, where players have different ratings and need to be assigned to n teams. There are requests regarding which pair of players want to be in the same team. The objective is to satisfy as many requests as possible while balancing the total rating among all teams.

In addition to publicly available models, we also model three more problems in MiniZinc:

- *MaxCover-n*. The Budgeted Maximum Coverage Problem [21] is a variant of the set cover problem. There is a ground set U and a collection T consisting of n subsets of U, where each subset is associated with a cost c_i. The goal is to find a subset of T such that the union covers the maximum number of elements subject to the constraint that the total cost does not exceed a given budget. The search strategy is to select the unfixed subset in T with the smallest cost first.
- *PartialCover-n*. The Partial Set Cover Problem [20] is another variant of the set cover problem. Given a ground set U and a collection T consisting of n subsets of U, the goal is to find a subset of T with the minimum total cost, whose union covers at least K elements in U. The search strategy is also to select the subset with the smallest cost first.

[2] We modify the embedded Geas solver and use the free search option for solving the generation CSPs. Our implementations are available at `https://github.com/AllenZzw/auto-dom`.

Table 1 Comparison of solving time for the **basic**, **manual** and ***L*-dom** methods.

	basic	**manual**	**2-dom**		**3-dom**		**4-dom**	
Problem	Total	Total	Solving	Total	Solving	Total	Solving	Total
Talent-16	187.79	5929.75	189.95	192.16	130.78	**148.91**	256.46	1988.75
Talent-18	1575.51	7200.00	1565.89	1568.29	672.26	**713.55**	1864.68	5760.68
Talent-20	5013.10	7200.00	4936.18	4960.54	2856.33	**2960.09**	3268.72	7006.10
Warehouse-35	7200.00	N/A	10.29	**52.11**	8.53	2442.71	8.51	3619.87
Warehouse-40	7200.00	N/A	46.08	**111.43**	32.93	3652.15	32.55	3657.33
Warehouse-45	7200.00	N/A	69.41	**140.92**	45.45	3690.84	46.19	3694.63
Team-6-5	24.48	N/A	10.57	**12.49**	9.70	32.00	8.88	427.73
Team-7-5	276.84	N/A	138.88	**146.15**	130.71	225.19	150.83	1745.96
Team-8-5	1983.53	N/A	819.58	**829.05**	767.52	1024.43	724.63	5191.70
MaxCover-45	75.91	N/A	53.47	53.79	5.07	**9.96**	0.27	83.93
MaxCover-50	615.04	N/A	464.81	465.53	26.31	**34.92**	1.12	134.99
MaxCover-55	3576.98	N/A	2859.60	2860.27	78.37	**91.53**	2.54	199.11
PartialCover-45	2383.20	N/A	366.17	368.03	59.44	**70.64**	2.49	90.25
PartialCover-50	3769.26	N/A	780.80	781.73	74.86	**88.45**	6.86	153.90
PartialCover-55	4640.06	N/A	1769.31	1770.42	211.83	**234.41**	15.23	240.68
Sensor-50	156.84	N/A	138.65	139.44	94.05	**108.99**	57.34	297.18
Sensor-60	595.46	N/A	404.27	405.52	269.61	**296.56**	172.43	709.37
Sensor-70	1615.18	N/A	1144.17	1145.83	810.01	**854.61**	651.72	1724.70

- *Sensor-n.* The Sensor Placement Problem [23] is a variant of the facility location problem [8], where we need to select a fixed cardinality subset of n locations to place sensors in order to provide service for customers. If we place a sensor at location i, then it provides service to a subset of reachable customers, and the service value for customer j is M_{ij}. Each customer chooses the facility with the highest service value from the opened sensors, and the goal is to maximize the total service value. The search strategy is to select the unfixed location with the highest service value to the set of customers that are reachable by the sensor placed at the location.

Note that the original method by Lee and Zhong [25] can handle none of the benchmarks effectively because of nested function calls in either the objective or constraints. For all benchmarks, we attempt to generate all dominance breaking nogoods of length up to L (***L*-dom**), and compare our method to the basic problem model (**basic**) and the model with manual dominance breaking constraints (**manual**) whenever they are available. The timeout for the whole solving process (nogood generation + problem solving) is set to 7200 seconds, while we reserve at most 3600 seconds for nogood generation and use the *remaining* time for problem solving in ***L*-dom**. If nogood generation times out, we augment the problem model with all nogoods generated before the timeout.

In Table 1, we report the geometric mean of the problem solving time (Solving) and the total time (Total) for all benchmarks, where "N/A" in the **manual** column indicates that there are no dominance breaking constraints for the problem. We first compare the problem solving time of ***L*-dom** against **basic** to evaluate the usefulness of the generated nogoods, and we observe that the generated dominance breaking nogoods can significantly reduce the solving time in all benchmarks. As the maximum nogood length L increases and more generated nogoods are added to the problem model, the solving time is usually shorter except for *Talent-n*. We note that the solving time of **4-dom** is larger than that of **3-dom** for *Talent-n* due to the overhead caused by a large amount of generated nogoods of length 4.

We also compare the total time (generation time + solving time) of ***L*-dom** against **basic** and **manual**. For each set of benchmarks, we highlight the fastest time in bold. We observe that the nogood generation time of ***L*-dom** increases with the maximum nogood length L of the generated nogoods, and there is a trade-off between search space pruning and generation time. The optimal nogood length depends on the problem structure. For *Warehouse-n* and *Team-n-m*, **2-dom** is the best and reduces up to 99.27% and 58.20% less time than **basic** respectively. In *Talent-n*, *MaxCover-n*, *PartialCover-n* and *Sensor-n*, **3-dom** usually comes on top, and the percentage decrease in runtime is up to 54.71%, 97.44%, 96.95% and 80.48% respectively compared with **basic**. The performance gain of ***L*-dom** in problem solving usually outweighs the generation time in a range of problems when the maximum length L of nogoods is set appropriately.

We note that the solving time of **manual** is even larger than that of **basic** in *Talent-n*. Expressing manual dominance breaking for *Talent-n* in the MiniZinc model requires additional variables and introduces overheads for propagation. Chu and Stuckey [7] implement the manual dominance breaking constraints in Chuffed, which requires sophisticated and bespoke techniques to reduce the overhead. The generated nogoods by our method only involve variables in the original model, and they can be posted in the high-level modeling language without modifying the backend solver.

6 Discovering Dominance Relations

Our method, which is based on that of Lee and Zhong [25, 24] attempts to generate all dominance breaking nogoods before problem solving, and sometimes the number of nogoods is so large that generating all nogoods will cost too much time for each problem instance. We observe that nogoods are the most basic units of constraints. Every high-level constraint can be decomposed into a group of nogoods, and vice versa. By examining the patterns of the generated nogoods, we could discover the embedded high-level dominance breaking constraints. We give two case studies in this section.

The first case study is the Steel Mill Slab Design Problem [19], which is problem 039 in CSPLib [15]. The problem is to assign colored production orders with different sizes to slabs where each slab has a finite number of possible sizes. The total size of orders assigned to a slab cannot exceed the chosen slab size, and each slab cannot contain orders with more than 2 colors. The loss of each slab is the difference between the chosen slab size and the total size of orders assigned to the slab, and the objective is to minimize the total loss of all slabs.

Previous works study different classes of symmetries, one of which is order symmetries [12], that is, two orders with identical size and color are equivalent. We apply our method to generate nogood of length 2 for the model from MiniZinc Challenge 2017 [36], which introduces one variable x_i to specify the slab that orders i is assigned to. The generation always times out within 3600 seconds, and the overhead always outweighs the benefit. Although a single nogood means relatively little, a bunch of them together can derive a meaningful constraint collectively. However, we investigate the semantics of nogoods and discover a new class of symmetries. By generating the nogoods of length 2, we observe that we can group all nogoods involving the same set of variables and find that for some pairs of orders i and j, the nogoods are $x_i \neq v_i \lor x_j \neq v_j$ for all $v_i \in D(x_i), v_j \in D(x_j)$ s.t. $v_i > v_j$, which can be combined into one single inequality constraint $x_i \leq x_j$. These symmetry breaking constraints force the order i to be on a slab whose index is less than or equal to the slab index of order j when orders i and j are equivalent. The surprise is that two orders are identical not only when they have the same size and the same color, but also when *they have the same size and their colors are unique.* To the best of our knowledge, previous studies never reveal and exploit such a symmetry relationship.

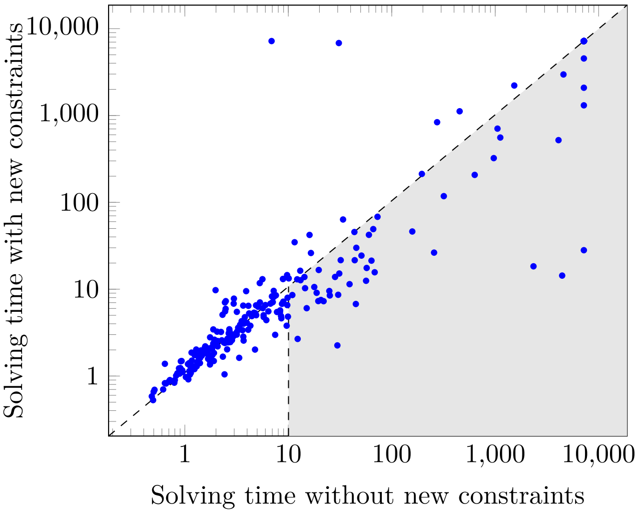


Figure 1 Solving time comparison with/without new symmetry breaking constraints in Steel Mill Slab Design Problem.

We take a constraint model of the steel mill slab design problem from a public benchmark suite[3] and augment it with constraints to break the newly discovered symmetry relationship. Figure 1 shows the solving time for all 380 instances from the steel mill slab library[4], and the dots below the diagonal line represent the instance benefiting from the newly discovered constraints. We observe that the solving time is reduced in the majority of cases, especially more so when the solving time of the original model requires more than 10 seconds. The hard instances are represented by dots in the shaded region in Figure 1. Note that both axes are in log scale, and the speed-up of new constraints is up to two orders of magnitude. Several outliers require substantially more solving time after adding the new symmetry breaking constraints. This is due to the conflict between the search heuristic and the static symmetry breaking constraints [13]. We believe the solving time can be reduced further by using dynamic symmetry breaking methods such as SBDS [14] or SBDD [9].

The other case study is the Balanced Academic Curriculum Problem [5], which is problem 030 in CSPLib [15]. There are n courses each associated with several credits representing the effort required to complete the course, and courses need to be assigned to academic periods subject to the course prerequisite constraints. The workload of each period is the sum of all credits of courses that are assigned to the period. The objective is to minimize the maximum academic load for all periods to balance the loads among academic periods.

We perform experiments using the same experimental setting as that in Section 5 and report the results for problems with different course numbers in Table 2. The dominance breaking constraints for **manual** are by Monette, Jean-Noël et al. [27]. In general, the problem solving time of our method is smaller than that of **basic** but larger than that of **manual**. The overhead of ***L*-dom** mainly comes from the generation of dominance breaking nogoods before solving the COP. In addition, the dominance breaking constraints in **manual** are in the form of inequalities, which can be handled more efficiently than nogoods added by ***L*-dom** in a propagation-based constraint solver. Nevertheless, by analyzing the nogoods of

[3] `https://github.com/MiniZinc/minizinc-benchmarks/tree/master/steelmillslab`
[4] `http://becool.info.ucl.ac.be/steelmillslab`

Table 2 Comparison of solving time for the Balance Academic Curriculum Problem.

	basic	**manual**	**2-dom**		**3-dom**		**4-dom**	
Problem	Total	Total	Solving	Total	Solving	Total	Solving	Total
Curriculum-60	61.82	23.76	27.80	77.44	21.88	3667.22	20.45	3673.27
Curriculum-65	291.35	62.14	71.26	160.95	62.57	3779.78	66.41	3785.61
Curriculum-70	518.31	133.54	148.84	242.62	126.19	3836.27	126.23	3837.26

a small instance, we find that the generated nogoods of length 2 can also be combined into inequality constraints similar to the case of the steel mill slab design problem. The inequality constraints we consider are the same as those proposed by Monette, Jean-Noël et al. [27], which shows that our method can also reveal dominance breaking constraints written by experts in the literature.

7 Concluding Remarks

In this work, we generalize the framework of automatic dominance breaking to constraint optimization problems with nested functions, where the derivation of sufficient conditions in a generation CSP is formulated formally. We identify that common function properties such as monotonicity, commutativity and associativity are useful in deriving weaker sufficient conditions such that more dominance breaking nogoods can be generated. We implement the tool for automatic dominance breaking using the MiniZinc compiler. The experimentation shows that the tool can discover dominance breaking nogoods for COPs with more varying objectives and constraints, and the generated nogoods are effective in pruning the search space and reducing the time for problem solving.

Our tool can compile and synthesize the generation CSPs for problems in the MiniZinc benchmarks[5]. Whether a benchmark can benefit from our method, however, cannot be guaranteed, since solving the generation CSP may sometimes incur a large overhead, or the generated nogoods do not help with problem solving. Our method requires a full constraint instance to synthesize generation CSPs. The automatic detection of dominance relations from constraint models alone is an interesting line of future work. As shown in the case studies in Section 6, nogoods with relevant semantics can be combined into high-level constraints that can be efficiently handled. One direction of future work is to automate the process of deriving high-level constraints by the techniques of automatic discovery of constraint from example solutions [3, 4], where the generated nogoods can be used as examples to learn and discover the desired constraints. The acquired constraints can help users to further understand the target COP and improve the efficiency of the existing models.

References

1 Tariq Aldowaisan. A new heuristic and dominance relations for no-wait flowshops with setups. *Computers & Operations Research*, 28(6):563–584, 2001.

2 Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

3 Nicolas Beldiceanu and Helmut Simonis. ModelSeeker: Extracting global constraint models from positive examples. In *Data Mining and Constraint Programming: Foundations of a Cross-Disciplinary Approach*, pages 77–95. Springer International Publishing, Cham, 2016.

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

4 Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artificial Intelligence*, 244:315–342, 2017.

5 Carlos Castro and Sebastian Manzano. Variable and value ordering when solving balanced academic curriculum problem. In *Proceedings of the ERCIM Working Group on Constraints*, 2001.

6 TCE Cheng, JE Diamond, and Bertrand MT Lin. Optimal scheduling in film production to minimize talent hold cost. *Journal of Optimization Theory and Applications*, 79(3):479–492, 1993.

7 Geoffrey Chu and Peter J Stuckey. A generic method for identifying and exploiting dominance relations. In *International Conference on Principles and Practice of Constraint Programming*, pages 6–22. Springer, 2012.

8 Gérard Cornuéjols, George Nemhauser, and Laurence Wolsey. The uncapicitated facility location problem. Technical report, Cornell University Operations Research and Industrial Engineering, 1983.

9 Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In *International Conference on Principles and Practice of Constraint Programming*, pages 93–107. Springer, 2001.

10 Alan M Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martinez-Hernandez, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

11 A.M. Frisch, I. Miguel, and T. Walsh. Modelling a steel mill slab design problem. In *Proceedings of the IJCAI-01 Workshop on Modelling and Solving Problems with Constraints*, pages 39–45, 2001.

12 A.M. Frisch, I. Miguel, and T. Walsh. Symmetry and implied constraints in the steel mill slab design problem. In *Proceedings of the CP'01 Workshop on Modelling and Problem Formulation*, pages 8–15, 2001.

13 Antoine Gargani and Philippe Refalo. An efficient model and strategy for the steel mill slab design problem. In *International Conference on Principles and Practice of Constraint Programming*, pages 77–89. Springer, 2007.

14 Ian P Gent and Barbara M Smith. Symmetry breaking in constraint programming. In *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 599–603, 2000.

15 Ian P Gent and Toby Walsh. CSPLib: a benchmark library for constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 480–481. Springer, 1999.

16 Lise Getoor, Greger Ottosson, Markus Fromherz, and Björn Carlson. Effective redundant constraints for online scheduling. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Conference on Innovative Applications of Artificial Intelligence*, pages 302–307, 1997.

17 Michel Grabisch, Jean-Luc Marichal, Radko Mesiar, and Endre Pap. *Aggregation Functions*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2009.

18 Toshihide Ibaraki. The power of dominance relations in branch-and-bound algorithms. *Journal of the ACM (JACM)*, 24(2):264–279, 1977.

19 Jayant R Kalagnanam, Milind W Dawande, Mark Trumbo, and Ho Soo Lee. Inventory matching problems in the steel industry. Technical report, IBM TJ Watson Research Center, 1998.

20 Michael J. Kearns. *Computational Complexity of Machine Learning*. MIT Press, Cambridge, MA, USA, 1990.

21 Samir Khuller, Anna Moss, and Joseph Naor. The budgeted maximum coverage problem. *Information Processing Letters*, 70(1):39–45, 1999.

22 Richard E Korf. Optimal rectangle packing: New results. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 142–149, 2004.

23 Andreas Krause, Jure Leskovec, Carlos Guestrin, Jeanne VanBriesen, and Christos Faloutsos. Efficient sensor placement optimization for securing large water distribution networks. *Journal of Water Resources Planning and Management*, 134(6):516–526, 2008.

24 Jimmy H. M. Lee and Allen Z. Zhong. Towards more practical and efficient automatic dominance breaking. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI 2021)*, pages 3868–3876, 2021.

25 Jimmy H.M. Lee and Allen Z. Zhong. Automatic generation of dominance breaking nogoods for a class of constraint optimization problems. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence*, pages 1192–1200, 2020.

26 Kevin Leo. *Making the Most of Structure in Constraint Models*. PhD thesis, Monash University, 2018.

27 Jean-Noël Monette, Pierre Schaus, Stéphane Zampelli, Yves Deville, and Pierre Dupont. A CP approach to the balanced academic curriculum problem. In *Symcon'07, The Seventh International Workshop on Symmetry and Constraint Satisfaction Problems*, 2007.

28 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.

29 Olga Ohrimenko, Peter J Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

30 A Oplobedu, J Marcovitch, and Y Tourbier. Charme: Un langage industriel de programmation par contraintes, illustré par une application chez renault. In *Ninth International Workshop on Expert Systems and their Applications: General Conference*, volume 1, pages 55–70, 1989.

31 Steven Prestwich and J Christopher Beck. Exploiting dominance in three symmetric problems. In *Fourth International Workshop on Symmetry and Constraint Satisfaction Problems*, pages 63–70, 2004.

32 Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 362–367, 1994.

33 Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the 13th National Conference on Artificial intelligence*, volume 1, pages 209–215, 1996.

34 Francesca Rossi, Peter van Beek, and Toby Walsh. Handbook of constraint programming (foundations of artificial intelligence), 2006.

35 Paul Shaw. A constraint for bin packing. In *International Conference on Principles and Practice of Constraint Programming*, pages 648–662. Springer, 2004.

36 Peter J Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15(3):307–316, 2010.

37 Pascal Van Hentenryck. *The OPL optimization programming language*. MIT press, 1999.

A Portfolio-Based Approach to Select Efficient Variable Ordering Heuristics for Constraint Satisfaction Problems

Hongbo Li
School of Information Science and Technology, Northeast Normal University, Changchun, China

Yaling Wu
School of Information Science and Technology, Northeast Normal University, Changchun, China

Minghao Yin
School of Information Science and Technology, Northeast Normal University, Changchun, China

Zhanshan Li[1]
College of Computer Science and Technology, Jilin University, Changchun, China

Abstract

Variable ordering heuristics (VOH) play a central role in solving Constraint Satisfaction Problems (CSP). The performance of different VOHs may vary greatly in solving the same CSP instance. In this paper, we propose an approach to select efficient VOHs for solving different CSP instances. The approach contains two phases. The first phase is a probing procedure that runs a simple portfolio strategy containing several different VOHs. The portfolio tries to use each of the candidate VOHs to guide backtracking search to solve the CSP instance within a limited number of failures. If the CSP is not solved by the portfolio, one of the candidates is selected for it by analysing the information attached in the search trees generated by the candidates. The second phase uses the selected VOH to guide backtracking search to solve the CSP. The experiments are run with the `MiniZinc` benchmark suite and four different VOHs which are considered as the state of the art are involved. The results show that the proposed approach finds the best VOH for more than 67% instances and it solves more instances than all the candidate VOHs and an adaptive VOH based on Multi-Armed Bandit. It could be an effective adaptive search strategy for black-box CSP solvers.

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

Keywords and phrases Constraint Satisfaction Problem, Variable Ordering Heuristic, Adaptive Search Heuristic, Portfolio

Digital Object Identifier 10.4230/LIPIcs.CP.2022.32

Supplementary Material *Software (Source Code)*: `https://github.com/lihb905/sevoh`
archived at `swh:1:dir:566338d2739577697c51179e2071e6bf0bea108f`

Funding *Hongbo Li*: The National Natural Science Foundation of China (61802056). The Plan of Science and Technology Development in Jilin Province (20210101470JC). Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University.
Minghao Yin: The National Natural Science Foundation of China under Grant NO. 61976050.
Zhanshan Li: Open Research Fund of Key Laboratory of Space Utilization, Chinese Academy of Sciences under Grant NO. LSU-KFJJ-2019-08.

1 Introduction

The challenge in a Constraint Satisfaction Problem (CSP) is to find an assignment of values to all variables that satisfies the constraints defined over the variables, or otherwise, to prove that there is no such an assignment. Backtracking search is a complete method for solving

[1] Corresponding author

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 32; pp. 32:1–32:10
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

CSPs. It performs a depth-first traversal of a search tree to solve CSPs. At each node of the search tree, an unassigned variable is selected to assign a value. The ordering in which the variables are assigned is crucial to the efficiency of backtracking search for solving CSPs. Thus, variable ordering heuristics (VOH) play a central role in solving CSPs.

In the past decades, much effort has been done in developing effective variable ordering heuristics [3, 21, 16, 23, 7, 25, 13]. There is no VOH dominating all the others in solving all CSP instances. In other words, different VOHs have different performances on different problems. The performances of different VOHs can vary greatly while solving the same CSP instance. Thus, if we can find the best VOHs for different CSP instances, then the overall performance of a black-box CSP solver will be significantly improved. In recent years, determining an efficient VOH for a given CSP instance has attracted much attention. For instance, a reinforcement learning technique, Multi-Armed Bandit (MAB), has been used to design adaptive VOHs for CSPs [27, 24]. In these approaches, MAB is employed to select VOHs to make decisions and the candidate VOHs are switched over during the search.

In this paper, we propose a new approach to select efficient VOHs for solving different CSP instances. The approach contains two phases. The first phase is a probing procedure that checks how each candidate VOH behaves when trying to solve the CSP instance within limited resources. It runs a portfolio strategy containing the candidate VOHs and monitors the searching process. In each run with a candidate, a search tree will be generated within a limited number of failures. If the failure number reaches the limit, it restarts the search to try the next candidate. For each candidate VOH, the maximum depth and the failure depth of the search trees are collected. If the CSP is not solved during the probing, some measurements utilizing the collected information are used to select an efficient VOH. In the second phase, the selected VOH is used to guide backtracking search until the search completes. Extensive experimentation with the MiniZinc benchmark suite are performed to examine the efficiency of the proposed approach. Four modern VOHs including activity-based search (ABS) [16], conflict-history search (CHS) [7], refined weighted degree (WDEG) [25] and failure rate based VOH (FRBA) [13] are used as candidates. The results show that the proposed approach finds the best VOH from four candidates for more than 67% instances and it solves more instances than all the candidates and an adaptive strategy based on MAB. The approach does not need an offline training and it is easy to be implemented in constraint solvers, so it could be an effective adaptive search strategy for black-box CSP solvers.

2 Background

A constraint satisfaction problem (CSP) $\mathcal{P}$ is a triple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X}$ is a set of n variables $\mathcal{X} = \{x_1, x_2 \ldots x_n\}$, $\mathcal{D}$ is a set of domains $\mathcal{D} = \{dom(x_1), dom(x_2) \ldots dom(x_n)\}$, where $dom(x_i)$ is a finite set of possible values for variable x_i, and $\mathcal{C}$ is a set of e constraints $\mathcal{C}=\{c_1, c_2 \ldots c_e\}$. Each constraint c consists of two parts, an ordered set of variables $scp(c) = \{x_{i1}, x_{i2} \ldots x_{ir}\}$ and a subset of the Cartesian product $dom(x_{i1}) \times dom(x_{i2}) \times \ldots \times dom(x_{ir})$ that specifies the disallowed (or allowed) combinations of values for the variables $\{x_{i1}, x_{i2} \ldots x_{ir}\}$. A solution to a CSP is an assignment of a value to each variable such that all the constraints are satisfied. Solving a CSP $\mathcal{P}$ involves either finding one (or more) solution of $\mathcal{P}$ or proving that $\mathcal{P}$ is unsatisfiable.

To solve real world problems, the users model the problems as CSPs and constraint solvers solve the CSPs. Average users have little knowledge about constraint solving, so an efficient black-box solver is required. Backtracking search is a standard algorithm for solving CSPs. It performs a depth-first traversal of a search tree. At each search tree node, an unassigned variable is selected and a new node is generated after the assignment to this variable, then a propagation algorithm is applied to filter those inconsistent values from the domains of variables. If the propagation leads to a domain wipe out, then a failure is encountered, one or more assignments must be canceled and a backtracking occurs. The ordering in which the variables are assigned is crucial to the efficiency of backtracking search and it is difficult to find an optimal ordering that results in a search tree exploring the smallest number of nodes [14]. Thus, the ordering is usually determined by variable ordering heuristics (VOH).

There exists many efficient VOHs, such as the impact-based search [21], activity-based search [16], the count-based search [19], the correlation-based search [23], the conflict-history search [7], the refined weighted degree [3, 25] and the failure-based VOHs [13]. None of them dominates all the others and their performances may vary greatly in solving a same CSP instance. Thus, finding the right VOH for a given CSP is a key issue for black-box solvers.

3 Related Work

Adaptive constraint solving has been studied in the CP community. A major difference between these methods is whether an offline training phase is required. Relying on the offline training phases, these methods are effective to predict an efficient algorithm, heuristic or even a solver for CSP instances [28, 9, 5, 1]. While these methods have shown their effectiveness, they may be less efficient to solve a new unseen instance (such as a new real-world problem) if it contains some unknown characteristics or structures, e.g., it is not close to any of the instances used in the training data.

On the contrary, the online learning methods do not require an offline training phase or training data. Some of them do learning during constraint solving procedure, such as the the modern VOHs [21, 16, 3, 23, 17, 7, 13, 12], the bandit-based search strategies [15, 27, 24] and the adaptive constraint propagation techniques [18, 26, 2]. Some other methods do online learning before searching starts, such as the learning value heuristics with a linear regression method [4] and the frequent pattern mining-based search [11].

Among these online learning approaches, the closest works are the two adaptive variable ordering heuristics based on Multi-Armed Bandit. Both them associate each candidate VOH with an arm. The first one applies the Upper Confidence Bound algorithm (UCB1) and Thompson Sampling (TS) algorithm to select a candidate VOH to make a decision at each search tree node [27]. If a new node is generated by a candidate H_i, the reward of H_i will be updated with the number of children of the search tree node. The second one applies the exponentially weighted forecaster for exploration and exploitation (EXP3), UCB1 and TS to estimate the best VOH for given CSPs [24]. It exploits a restart mechanism with the MAB-based framework. Besides, the candidate VOHs are switched only after restarts. Its reward function is defined by the pruned tree size (PTS) [24]. The two methods use the reinforcement learning technique to combine different VOH online, e.g., the candidate VOHs are switched over during search. Our approach uses an effective measurement to select the best VOHs for given CSPs. Although our approach also switches the candidate VOH at the probing phase, after the best one is selected, it will be used to guide backtracking search until the search completes.

4 Selecting An Efficient VOH for a CSP

Given a set of k candidate VOHs, H_1, H_2, ..., H_k. We propose an approach, namely Selecting Efficient Variable Ordering Heuristics (SEVOH), to select an efficient VOH for solving a given CSP instance. The approach consists of two phases. The first phase is a probing procedure that runs a simple portfolio strategy to collect some information of search trees built by the candidate VOHs. If the problem is not solved by the probing procedure, we analysis the collected information to select an efficient VOH for the problem. The second phase is a straightforward strategy that uses the selected VOH to guide backtracking search to solve the problem. In the following, we introduce how to design the measurements for selecting an efficient VOH and how to collect search tree information for the measurements in a portfolio-based probing procedure.

4.1 The Measurements for Selecting An Efficient VOH

To select an efficient VOH for a CSP instance, we analysis how the candidate VOHs behave when solving the instance within limited resources, e.g., a limited number of failures. Then some measurements should be used to evaluate the performance of the candidates.

Firstly, many effective VOHs are designed according to the Fail First Principle that "to succeed, try first where you are likely to fail" [8]. Therefore, one intuition is that a more efficient VOH may detect failures at higher levels of the search trees, which has been used by the failure length based VOH [13]. The intuition is reasonable, because the higher level a failure is detected, the more search space is pruned. So our first measurement is the minimum failure depth of the search trees of each candidate.

- **Minimum Failure Depth (`MinFD`)**
  Given a CSP instance and a VOH H_i, we use H_i to guide backtracking search to solve the instance and a search tree will be built. In the search tree, the number of assigned variables when a failure occurs is record as the depth of the failure. A search tree usually contains a number of failures, so we use the average of the depths of all the failures as the failure depth of the search tree. The probing procedure, introduced in next subsection, will generates *roundLimit* search trees for H_i. The minimum one of the failure depth of the *roundLimit* search trees is considered as the `Minimum Failure Depth` of H_i for the instance, denoted by `MinFD`(H_i). This measurement prefers the VOH with smaller `MinFD`.

Secondly, an efficient VOH for a satisfiable CSP instance should find a solution as early as possible. An intuition is that a more efficient VOH may explore a deeper search tree than the others within limited resources. So our second measurement is the maximum depth of the search trees.

- **Maximum Depth (`MaxD`)**
  Given a CSP instance, each candidate H_i will build *roundLimit* search trees during the probing procedure. Each of the search trees has a deepest depth which is the largest number of simultaneously assigned variables. We use the largest one of all the deepest depth of the *roundLimit* search trees as the `Maximum Depth` of H_i for the instance, denoted by `MaxD`(H_i). This measurement prefers the VOH with larger `MaxD`.

Given a CSP instance, for each candidate H_i, we collect the information during the probing procedure to calculate `MinFD`(H_i) and `MaxD`(H_i). Each of the two measurements can be used as the scoring function to evaluate the performance of H_i solving the instance. Besides, we can combine the two measurements to use $\frac{\texttt{MaxD}(H_i)}{log(\texttt{MinFD}(H_i))}$ as the scoring function to evaluate the performance of H_i, which prefers the candidate with the largest score. The logarithmic scaling is to make `MinFD`(H_i) a smaller number, because the `MinFD`(H_i) and `MaxD`(H_i) are quite close in some instances.

4.2 A Portfolio-Based Probing

The portfolio contains several candidate VOHs, H_1, H_2, ..., H_k. The probing procedure is shown in Algorithm 1.

Algorithm 1 Portfolio-Based Probing.

```
   Input: $k$ candidate VOHs: $H_1$, $H_2$, ..., $H_k$; the maximum number of rounds:
          $roundLimit$; the maximum number of failures in each call: $failLimit$
   Output: the collected MinFD and MaxD, or unsatisfiable, or a solution.
 1 for $i$ = 1 to $k$ do
 2     MaxD($H_i$) ← 0;
 3     MinFD($H_i$) ← the number of variables;
 4 $round$ ← 1;
 5 while $round \leq roundLimit$ do
 6     for $i$ = 1 to $k$ do
 7         $failNum$ ← 0;
 8         $totalFailDepth$ ← 0;
 9         while $failNum < failLimit$ do
10             $depth$ ← the number of fixed variables;
11             if $depth >$ MaxD($H_i$) then
12                 MaxD($H_i$) ← $depth$;
13             $x$ ← the variable selected by $H_i$;
14             $v$ ← a value selected for $x$;
15             if the propagation of the assignment $x$=$v$ fails then
16                 $totalFailDepth$ ← $totalFailDepth + depth$;
17                 $failNum$ ← $failNum$ + 1;
18                 if unsatisfiable is proved then
19                     terminate the search and return unsatisfiable;
20                 backtracking occurs;
21             else
22                 if a solution is found then
23                     terminate the search and return the solution;
24         $failDepth$ ← $totalFailDepth/failNum$;
25         if $failDepth <$ MinFD($H_i$) then
26             MinFD($H_i$) ← $failDepth$;
27         restart the search;
28     $round$ ← $round$ + 1;
```

The procedure runs at most *roundLimit* rounds (line 5). In each round, the candidate VOHs are called sequentially. In each call of a candidate H_i (lines 9 to 23), we run backtracking search with H_i as the variable ordering heuristic and set a restart condition to $failLimit$ failures. If the failure number reaches the limit, then we record the search tree information and restart the search with next candidate VOH. Every candidate builds a search tree in each round and the corresponding information is recorded, so if the problem is not solved at the probing procedure, then the information of *roundLimit* search trees will be recorded for each candidate.

Modern VOHs usually use some learning strategies during search, so we make all candidate VOHs to learn during the entire probing procedure. In other words, during the call of candidate H_i, the information used by H_i will be learned and updated. Meanwhile, the information used by the other candidates will be learned and updated. All the information for the candidates, such as the weighted degrees [25], the activities of variables [16], the conflict history [7] and the failure rates [13], will be accumulated throughout the procedure.

The portfolio strategy runs backtracking search with different VOHs, so if a CSP instance can be easily solved by one of the candidate VOHs, then it may be solved during the probing procedure; otherwise, we analysis the recorded information of search trees and use the proposed measurements to select an efficient candidate VOH for the instance.

5 Experiments

To examine the efficiency of the proposed approach, we perform extensive experimentation with the MiniZinc benchmark suite. Four candidate VOHs including ABS [16], $dom/wdeg^{ca.cd}$ (marked by WDEG in the following tables) [25], CHS [7] and FRBA [13] are involved. The performance of searching for the first solution or proving unsatisfiable are measured by cpu time in seconds and numbers of instances solved in a timeout limit of 1200 seconds. In the following, we present the results of all instances (All), the satisfiable instances (Sat) and the unsatisfiable instances (UnSat) respectively. The best one in each comparison is in bold. More details about the experiments can be found in the appendix.

Table 1 The performance of different measurements.

		PTS	MinFD	MaxD	$\frac{\text{MaxD}}{log(\text{MinFD})}$
All (325)	BestOne	15.08%	17.23%	59.69%	**67.69%**
	BestTwo	23.69%	26.77%	72.62%	**81.85%**
Sat (288)	BestOne	15.63%	16.67%	64.58%	**72.22%**
	BestTwo	23.96%	20.14%	75.35%	**83.33%**
UnSat (37)	BestOne	10.81%	21.62%	21.62%	**32.43%**
	BestTwo	21.62%	**78.38%**	51.35%	70.27%

Firstly, we examine the percentage of instances where SEVOH finds the best one among the four candidate VOHs with different measurements. The reward function of the MAB-based VOH can be adapted as a measurement for SEVOH, so we further involve the pruned tree size (PTS) [24] as a measurement here. The results are presented in Table 1. After eliminating the instances solved by the probing procedure and the instances where all candidates result in timeout, the table contains the results of 325 instances. The BestOne (BestTwo) row is the percentage of instances where the VOH selected by SEVOH is the best one (one of the best two candidates). It is shown that PTS is not suitable for the selection here. Although MinFD does not work well in finding efficient VOHs for the satisfiable instances, it works well in finding a good candidate which is one of the best two for the unsatisfiable ones. On the contrary, MaxD works well in the satisfiable instances, but it does not work well in the unsatisfiable ones. The observation indicates that we should use different measurements for satisfiable and unsatisfiable instances. However, the satisfiability of a CSP is not determinable before solving it, so we cannot pre-select a measurement for each instance. Thankfully, combining the two measurements, $\frac{\text{MaxD}}{log(\text{MinFD})}$ performs well in finding efficient VOHs for all the instances. For both the satisfiable and the unsatisfiable, it finds a good candidate which is one of the best two for more than 70% instances. Thus, we uses $\frac{\text{MaxD}}{log(\text{MinFD})}$ as the measurement in the following experiments.

Secondly, we examine how the parameter *roundLimit* affects the percentage of instances where SEVOH finds good candidates. The results are presented in Table 2. The #Solved by Portfolio is the number of instance solved by the probing procedure and the Probing Time is the average time cost of the probing procedure. It is shown that different *roundLimit* makes little affection for the percentages. With the increasing of *roundLimit*, the number of instances solved by the portfolio strategy increases, as well as the probing time cost. But there is a trend that the number of increased instances solved by the probing procedure is diminishing, e.g., at the beginning we have 808-785=23, and then 16, 7, 4, 2. It indicates that increasing the *roundLimit* of the portfolio may not always solve more instances. Thus, we should find the best VOH for the hard instances which can not be easily solved by any of the candidate VOHs. The *roundLimit* 100 results in the largest percentage of BestTwo, so we set *roundLimit* to 100 in the following experiments.

Table 2 The performance of the probing procedure with different *roundLimits*.

	50	100	200	300	400	500
BestOne	66.86%	67.69%	**67.81%**	66.79%	66.79%	67.29%
BestTwo	79.71%	**81.85%**	80.14%	78.83%	79.48%	79.70%
#Solved by Portfolio	785	808	824	831	835	**837**
Probing Time	**63.61**	78.08	94.78	103.29	109.16	114.42

Thirdly, we compare SEVOH with the candidate VOHs and the RestartMAB (marked by $\texttt{RMAB}_{\texttt{PTS}}$) [24] which is a MAB-based VOH with its default reward function PTS. Besides the $\texttt{RMAB}_{\texttt{PTS}}$, we further involve a strategy (marked by $\texttt{RMAB}_{\frac{\texttt{Max}}{\texttt{Min}}}$) that adapts $\frac{\texttt{MaxD}}{log(\texttt{MinFD})}$ as the reward function in the MAB-based framework. In Table 3, we present the number of instances solved (#Solved), the average cpu time in solving the instances that are solved by all the compared VOHs (average time of all-solved instances, AST) and the average cpu time in solving all the instances which are solved by at least one of the VOHs (average time of all instances, AllT). The integer in the brackets after AST is the number of all-solved instances, so is the one after AllT. The time cost of a timeout run is counted as 1200 seconds. It is shown that SEVOH performs better than all the others in solving the satisfiable instances. Although SEVOH is outperformed by $\texttt{RMAB}_{\frac{\texttt{Max}}{\texttt{Min}}}$ in solving the unsatisfiable instances, it solves only 1 instance less than the bests. Thus, SEVOH gets the best overall performance.

Table 3 The overall performance.

		ABS	CHS	WDEG	FRBA	$\texttt{RMAB}_{\texttt{PTS}}$	$\texttt{RMAB}_{\frac{\texttt{Max}}{\texttt{Min}}}$	SEVOH
All	#Solved	630	730	700	985	894	933	**1056**
	AST (464)	34.91	26.67	57.91	23.65	18.29	11.87	**9.18**
	AllT (1131)	573.29	461.88	500.95	190.68	322.87	266.12	**142.38**
Sat	#Solved	570	679	648	919	830	867	**991**
	AST (416)	35.76	27.59	60.98	21.16	20.11	12.81	**8.20**
	AllT (1059)	590.51	467.87	509.69	186.42	333.75	273.07	**137.77**
Unsat	#Solved	60	51	52	**66**	64	**66**	65
	AST (48)	27.57	18.71	31.38	45.26	**2.54**	3.65	17.70
	AllT (72)	320.11	373.76	372.46	253.44	**162.81**	163.81	210.12

It has been shown in Table 2 that the probing procedure of SEVOH solves a number of instances. This is because the portfolio strategy works well in solving most of the instances that can be easily solved by at least one of the candidates. We are wondering how SEVOH performs in solving the hard instances that cannot be solved by the probing procedure, e.g., those cannot be easily solved by any of the candidates. Thus, in Table 4, we present the results of such instances. We can see that SEVOH solves the largest numbers of both satisfiable instances and unsatisfiable instances.

Table 4 Results of the instances that cannot be solved by the probing procedure of SEVOH.

		ABS	CHS	WDEG	FRBA	RMAB$_{PTS}$	RMAB$_{\frac{Max}{Min}}$	SEVOH
All	#Solved	121	91	101	224	133	151	**248**
	AST (53)	59.83	116.44	251.29	130.97	48.57	**38.55**	63.96
	AllT (323)	807.24	902.62	898.90	414.15	776.49	695.10	**395.78**
Sat	#Solved	87	64	75	193	97	115	**212**
	AST (29)	**64.03**	182.01	407.54	164.69	84.61	64.48	87.84
	AllT (280)	876.52	966.57	954.09	400.46	863.27	768.48	**408.00**
Unsat	#Solved	34	27	26	31	**36**	**36**	**36**
	AST (24)	54.76	37.22	62.48	90.23	**5.02**	7.23	35.10
	AllT (43)	356.07	486.19	539.55	503.26	**211.46**	217.28	316.24

Finally, we compare the performance of the three adaptive strategies. Three different combinations of candidates are considered. The results are presented in Table 5. It is shown SEVOH solves the largest numbers of instances in all the combinations of candidates. When combining three candidates, SEVOH costs more time than the others in solving the all-solved instances. This is because it has poor performance in solving some of the instances. However, its average time cost of solving all the instances which are solved by at least one of the 9 strategies is the least one.

Table 5 The performance of the adaptive strategies with different candidates.

	ABS+CHS			ABS+CHS+WDEG			ABS+CHS+WDEG+FRBA		
	RMAB$_{PTS}$	RMAB$_{\frac{Max}{Min}}$	SEVOH	RMAB$_{PTS}$	RMAB$_{\frac{Max}{Min}}$	SEVOH	RMAB$_{PTS}$	RMAB$_{\frac{Max}{Min}}$	SEVOH
#Solved	754	759	**814**	755	796	**846**	894	933	**1056**
AST (683)	26.92	23.35	**21.71**	**24.08**	28.85	28.72	26.73	24.75	**19.48**
AllT (1111)	424.87	415.07	**358.96**	423.39	398.71	**335.55**	307.08	249.31	**123.34**

6 Conclusion

In this paper, we propose a portfolio-based approach to select efficient variable ordering heuristics for CSPs. Extensive experimentations performed on MiniZinc benchmark suite demonstrate that the portfolio strategy is effective in solving the instances that can be easily solved by some of the candidate VOHs. Besides, the measurement combining the information of minimum failure depth and the maximum depth of search trees is effective to select good candidate VOHs for different CSPs. With the measurement, SEVOH finds the best one from four candidate VOHs for more than 67% instances. Consequently, SEVOH solves more instances than all the candidate VOHs and the adaptive VOH based on Multi-Armed Bandit. It could be an effective adaptive search strategy for black-box CSP solvers.

References

1 R. Amadini, M. Gabbrielli, and J. Mauro. Portfolio approaches for constraint optimization problems. *Annals of Mathematics and Artificial Intelligence*, 76:229–246, 2016.

2 A. Balafrej, C. Bessiere, and A. Paparrizou. Multi-armed bandits for adaptive constraint propagation. In *Proc. IJCAI'15*, pages 290–296. AAAI Press, 2015.

3 F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proc. ECAI'04*, pages 146–150, 2004.

4 G. Chu and P. J. Stuckey. Learning value heuristics for constraint programming. In *Proc. CPAIOR'15*, pages 108–123. Springer, 2015.

5 S. Epstein and S. Petrovic. Learning to solve constraint problems. In *Proc. ICAPS'07, Workshop on Planning and Learning*, 2007.

6 C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. AAAI'98*, pages 431–437. AAAI, 1998.

7 D. Habet and C. Terrioux. Conflict history based search for constraint satisfaction problem. In *Proc. of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1117–1122. ACM, 2019.

8 R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

9 H. Hurley, L. Kotthoff, Y. Malitsky, and B. O'Sullivan. Proteus: a hierarchical portfolio of solvers and transformations. In *Proc. CPAIOR'14*, 2014.

10 J. Hwang and D. G. Mitchell. 2-way vs d-way branching for csp. In *Proc. CP'05*, pages 343–357. Springer, 2005.

11 H. Li, J. H. Lee, H. Mi, and M. Yin. Finding good subtrees for constraint optimization problems using frequent pattern mining. In *Proc. AAAI'20*, pages 1577–1584, 2020.

12 H. Li, Y. Liang, N. Zhang, J. Guo, D. Xu, and Z. Li. Improving degree-based variable ordering heuristics for solving constraint satisfaction problems. *Journal of Heuristics*, 22(2):125–145, 2016.

13 H. Li, M. Yin, and Z. Li. Failure Based Variable Ordering Heuristics for Solving CSPs. In *Proc. CP'21*, pages 9:1–9:10, 2021.

14 P. Liberatore. On the complexity of choosing the branching literal in dpll. *Artificial Intelligence*, 116(1):315–326, 2000.

15 M. Loth, M. Sebag, Y. Hamadi, and M. Schoenauer. Bandit-based search for constraint programming. In *Proc. CP'13*, pages 464–480. Springer, 2013.

16 L. Michel and P. Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *Proc. CPAIOR'12*, pages 228–243. Springer, 2012.

17 A. Palmieri and G. Perez. Objective as a feature for robust search strategies. In *Proc. CP'18*, pages 328–344. Springer, 2018.

18 Anastasia Paparrizou and Kostas Stergiou. Evaluating simple fully automated heuristics for adaptive constraint propagation. In *Proc. of ICTAI'12*, volume 1, pages 880–885, 2012. doi:10.1109/ICTAI.2012.123.

19 G. Pesant, C. G. Quimper, and A. Zanarini. Counting-based search: Branching heuristics for constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 43:173–210, 2012.

20 C. Prud'homme, J-G. Fages, and X. Lorca. *Choco Documentation.* TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017. URL: http://www.choco-solver.org.

21 P. Refalo. Impact-based search strategies for constraint programming. In *Proc. CP'04*, pages 557–571. Springer, 2004.

22 T. Walsh. Search in a small world. In *Proc. IJCAI'99*, pages 1172–1177, 1999.

23 R. Wang, W. Xia, and R. H. C. Yap. Correlation heuristics for constraint programming. In *Proc. ICTAI'17*, pages 1037–1041. IEEE, 2017.

24 H. Wattez, F. Koriche, C. Lecoutre, A. Paparrizou, and S. Tabary. Learning variable ordering heuristics with multi-armed bandits and restarts. In *Proc. ECAI'20*, pages 371–378. IOS Press, 2020.

25 H. Wattez, C. Lecoutre, A. Paparrizou, and S. Tabary. Refining constraint weighting. In *Proc. of ICTAI'19*, pages 71–77. IEEE, 2019.

26 R. J. Woodward, A. Schneider, B.Y. Choueiry, and C. Bessiere. Adaptive parameterized consistency for non-binary csps by counting supports. In *Proc. of CP'14*, pages 755–764, 2014.

27 W. Xia and R. H. C. Yap. Learning robust search strategies using a bandit-based approach. In *Proc. AAAI'18*, pages 6657–6665. AAAI, 2018.

28 L. Xu, F. Hutter, H. H Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

A Details of The Experiments

A.1 Environment

The experiments were run in Choco solver (version 4.10.6) [20] where all the candidate VOHs are already implemented. The environment is JDK8 under CentOS 6.4 with 4 Intel Xeon CPU E7-4820@2.00GHz processors and 58 GB RAM. Each run is allocated 1 GB RAM.

A.2 Benchmark

The benchmark suite are from `https://github.com/MiniZinc/minizinc-benchmarks`. The instances are flattened offline. After eliminating some large instances which cannot be flattened in 1 hour and the problems where unsatisfiable is proved at root node, we include 46 `MiniZinc` models with 1876 instances in the experiments.

A.3 Restart

The $failLimit$ of the probing procedure of `SEVOH` is set to the number of variables. If the instance contains less than 100 variables, the $failLimit$ is set to 100. A geometric restart strategy [6, 22] is equipped by the second phase of `SEVOH` and the candidates VOHs. The growing factor is 1.1 and the initial cutoff is 10 failures. The restart strategy of the `MAB`-based framework is the Luby restart which is the default one used in [24].

A.4 Searching

Binary branching strategy [10] is used throughout the experiments. `ABS` uses its default value selector and all the others use lexicographic ordering as the value selector. We forbid the sampling procedure of `ABS` when it is used as a candidate, because the probing procedure of `SEVOH` warms up all the candidates. When `ABS` is used alone, we run it with its default settings.

A.5 Other Details

In Table 1 and Table 2, we need to determine the best candidate for each instance. Thus, for each instance, we run each candidate H_i with 10 random seeds from 1 to 10 and use the average time cost of the 10 runs as the performance of H_i solving the instance. The best VOH for a instance is the one costing least cpu time. Then we run the probing procedure with random seed 0 to select a VOH for each instance. The results in Table 1 are obtained with *roundLimit* 100. In Tables 3, 4 and 5, we have used a unique random seed 0 in the experiments. The instances where all the seven VOHs result in timeout are eliminated.

Large Neighborhood Search for Robust Solutions for Constraint Satisfaction Problems with Ordered Domains

Jheisson López[1]
University College Cork, School of Computer Science, Ireland
SFI Centre for Research Training in Artificial Intelligence, Cork, Ireland

Alejandro Arbelaez
Department of Computer Engineering, Autonomous University of Madrid, Spain

Laura Climent
Department of Computer Engineering, Autonomous University of Madrid, Spain

Abstract

Often, real-world Constraint Satisfaction Problems (CSPs) are subject to uncertainty/dynamism not known in advance. Some techniques in the literature offer robust solutions for CSPs. Here, we analyze a previous exact/complete approach from the state-of-the-art that focuses on CSPs with ordered domains and dynamic bounds. However, this approach has low performance in large-scale CSPs. For this reason, in this paper, we present an inexact/incomplete approach that is faster at finding robust solutions for large-scale CSPs. It is useful when the computation time available for finding a solution is limited and/or in situations where a new one must be re-computed online because the dynamism invalidated the original one. Specifically, we present a Large Neighbourhood Search (LNS) algorithm combined with Constraint Programming (CP) and Branch-and-bound (B&B) that searches for robust solutions. We also present a robust-value selection heuristic that guides the search toward more promising branches. We evaluate our approach with large-scale CSPs instances, including the case study of scheduling problems. The evaluation shows a considerable improvement in the robustness of the solutions achieved by our algorithm for large-scale CSPs.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence

Keywords and phrases Constraint Programming, Large Neighbourhood Search, Robust Solutions

Digital Object Identifier 10.4230/LIPIcs.CP.2022.33

Funding This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant number 18/CRT/6223.

Acknowledgements We thank Christophe Lecoutre for his assistance with the ACE solver.

1 Introduction

Most real Combinatorial Optimization Problems (COP) have unknown dynamism associated. Therefore the techniques used for solving COPs should deal with the uncertainty. The uncertainties can be associated with environmental factors; i.e., unexpected events in the context of the problem, or system factors; e.g., implementation errors or operational failures. Eventually, the uncertainties could affect the feasibility and the cost of a solution [21]. Even if the magnitude of the perturbations in the problem treated is small, they can cause huge deviations in the solutions of the models and in the model itself [3]. Hence, non-robust solutions for real applications under uncertainty can lead to serious economic losses.

[1] Contact Author

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 33; pp. 33:1–33:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

There are two ways to cope with uncertainty: reactive and proactive approaches. Reactive approaches re-solve the problem by using the experience gained to solve the previous states of the problem. Then, they provide a new solution. The disadvantages of these techniques are the extra computation time required in the new solving and the loss of the original solution. In many real applications, such as online planning and scheduling, the time required to compute a new solution may be too long for actions to be taken on time. Instead, proactive approaches use prior information about the uncertainty for finding robust solutions, which have a high probability of remaining valid despite future possible changes [7]. The disadvantage of proactive approaches is that they require a certain degree of prior knowledge about uncertainty. However, in real applications, this information is typically limited.

While many researchers have addressed the topic of robust and stable solutions in CP for problem dependent, much less effort has been made for problem independent, i.e. using CP as a black-box search. For this reason, we propose a generic proactive approach that addresses these scenarios without the need of prior detailed information about the uncertainty of the problem. In the same vein as [7] we assume that future events can restrict the search space (i.e. constraints can become more restrictive) in Constraint Satisfaction Problems (CSPs) with ordered domains. We also consider as robustness objective function the feasibility checking of neighbour values of the solution. Note that the CSPs must have several feasible neighbour solutions so that the robustness measurement is applicable. The authors of [7], propose a Branch and Bound (B&B) algorithm that uses CP to obtain robust solutions for CSPs, but their complete/exact technique presents a low performance in large-scale CSPs even using a large amount of computational time.

The above mentioned disadvantage has motivated the work presented in this paper. We present an incomplete/inexact approach that, unlike complete/exact algorithms, do not guarantee to find a globally optimal solution. The advantage of incomplete/inexact algorithms is that they can provide near-optimal solutions when the computation time is limited. Our approach aims to obtain robust solutions (not necessarily the most robust solution) for CSPs that typically do not scale well with complete techniques (large-scale and NP-complete CSPs). It is especially useful when the computation time available for finding a solution is limited and/or a new solution must be re-computed on-line (because the original one is lost due to the dynamism). Specifically, we present the following contributions:

- A Large Neighbourhood Search (LNS) algorithm with CP and B&B that considers the same assumptions as [7]. And therefore, it computes robust solutions for CSPs that have a high number of feasible neighbour values (Algorithms 1 and 2, Section 4).
- A robust-value selection heuristic (Algorithm 4, Section 4) that guides the search towards more promising branches (more likely to contain more feasible neighbour values solutions).

LNS is a technique that takes an initial solution and gradually improves it by alternately destroying and repairing the solution. LNS combined with CP adds the constraint propagation process into the LNS iterations to perform partial assignments that are feasible.

Our approach works for many domains that can be modeled as CSPs with ordered domains (where the bounds can undergo changes). In this paper, we evaluate general CSPs randomly generated, which show the generality of the applicability of our approach; and the well-known scheduling problems [11, 2]. The experiments suggest that our approach is effective on large-scale CSPs and that it outperforms the current approach from the state-of-the-art.

The rest of the paper is organized as follows: Section 2 is about the literature review. In Section 3 the technical background is described. Section 4 describes our algorithm. Section 5 describes the experiments and results; finally, in Section 6 we explain the conclusions and future work.

2 Literature Review

Since our approach is proactive and it aims to find robust solutions with the use of LNS with CP, in this section, we present a literature review of these two topics.

2.1 Proactive Approaches

The purpose of finding robust solutions is present in both Mathematical Programming (MP) and Constraint Programming (CP). Toklu [20] presents a revision of the approaches to deal with the uncertainty in Mathematical Programming. Furthermore, [22] provides a detailed review of approaches to model uncertainty with CP. The authors state that in CP the efforts had been directed to find solutions that can easily be adapted to obtain a new solution after a change in the conditions of the problem occurs (flexible solutions) and solutions that resist the possible changes in the input data (robust solutions).

Below, we present two subsections about proactive approaches. We classify them based on the amount of information available about the uncertainty that the approaches require.

2.1.1 With detailed uncertainty information

One of the most popular proactive approaches is *Robust Optimization* (RO). RO expresses the uncertainty of the variables as sets of possible values (uncertainty sets). The optimal solution is the one that remains feasible for the constraints, whatever the realization of the data within the uncertainty sets [3]. Another popular proactive approach is *Stochastic Optimization* (SO). In SO some variables are considered random with a probability function associated that expresses its likelihood to take any particular value in its uncertain domain [22]. One technique for solving this type of problem is by using *chance constraints* [1]. These constraints contain at least one random variable and they fix a minimum threshold of probability of being satisfied. A possible objective could be to find a solution that maximises its probability of satisfaction. Finally, some proactive approaches recur to the *Fuzzy Set* theory. In that approach, the uncertainty parameters of the model can be expressed as fuzzy sets and subjective membership functions defined using expert opinions [8].

In the search for robust solutions in CSPs, the *Mixed CSP* [10] approach introduces the concept of uncontrollable variables and the idea is to find assignments to the decision variables that satisfy all the possible values of the uncontrollable variables (a kind of Robust Optimization approach). Another outstanding proposal is the *Stochastic CSP* which assumes a probability distribution associated with the uncertain domain of each uncontrollable variable and tries to find a solution with the maximum probability of feasibility [23].

2.1.2 Without detailed uncertainty information

Even if the proactive approaches mentioned in the previous subsection, work very well when all the needed information about the uncertainty is available, unfortunately, they can not be used if such information is missing. This is the case of many real applications for which it is not possible to obtain a probability distributions associated with the uncertain variables [6].

A prominent work aiming to find flexible solutions in presented in [13]. In this work the author proposed the concept of *(a, b)-super-solution*, a set of solutions for which the loss of values of at most a variables can be repaired by assigning other values to those variables and changing the values of at most b other variables. Flexibility is a desired property of a CSP solution but our main goal is to avoid the necessity of modifying the solution after identifying inconsistencies in the solution due to certain uncertain events.

Climent, et al. [7] state that the solution can be lost when the constraints become more restrictive (and the solution space becomes narrow). Therefore *"the most robust solution of a CSP with ordered domains without detailed dynamism data is the solution that maximizes the distance from all the dynamic bounds of the solution space"*. In this paper, we take the same assumptions about the uncertainty and the same robustness definition.

2.2 Large Neighbourhood Search

Large Neighborhood Search was proposed by P. Shaw [18] to solve Vehicle Routing Problems using CP techniques. Since then, LNS and CP has been applied to tackle multiple problems ranging from the protein structure prediction problem [9] to balance bike-sharing systems [12].

Carchrae and Beck [5] propose three general principles for the design of LNS algorithms: i) define neighbourhoods unassigning the variables that most affect the cost of the current solution and ii) gradually increment the neighbourhood size and iii) apply learning algorithms to determine the most promising neighborhood heuristics. Some works fix a constant size for the neighborhood and other works use a variable neighbourhood size. For example, in [17], Perron, et. al fix the size of the neighbourhood based on the sum of the domain size logarithms of unassigned variables (until reaching an upper bound). For scheduling, for example, Carchrae and Beck [5] fix the size of the neighbourhood based on different time windows (all the variables in such time windows are unassigned) or based on resources (the variables associated with a subset of resources are unassigned). Pacino and Van Hentenryck [16] increment the size of the neighborhoods after five iterations without improvements in the solutions found. The increments are no longer effective when a better solution is found.

However, in this paper, we focus our attention on value selection heuristics as it significantly impacts the robustness of the solutions. For this reason, we have developed Algorithm 4 (see Section 4), which guides the search towards more robust promising branches.

3 Background

In this section, we explain the background associated with CSPs and their robust solutions.

3.1 Constraint Satisfaction Problems

CSPs are characterized by a set of variables with their corresponding domains and a set of associated constraints. A formal definition of a CSPs is presented below.

▶ **Definition 1** (Constraint Satisfaction Problems). *A Constraint Satisfaction Problem (CSP) is represented as a triple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{X} = \{x_1, ..., x_n\}$ is a finite set of variables, $\mathcal{D} = \{\mathcal{D}(x_1), ..., \mathcal{D}(x_n)\}$ is the set of domains of the variables in $\mathcal{X}$, and $\mathcal{C} = \{C_1, C_2, ...C_n\}$ is the set of constraints which restrict the values that the variables can simultaneously take.*

Solving a CSP involves finding a solution, i.e., an assignment of values to variables such that all the constraints are satisfied. If a solution exists, the problem is stated as satisfiable, and unsatisfiable otherwise. CSPs can be tackled using depth-first search backtracking algorithms in which a value is assigned to some variable at each step to compute partial assignments. Below, we present the formal definitions of these concepts.

▶ **Definition 2** (Partial Assignment). *A partial/complete assignment s is an assignment of values to a subset of variables $\mathcal{X}_s \subseteq \mathcal{X}$ in their specific domain $\mathcal{D}_s$.*

▶ **Definition 3** (Solution). *A solution s is a complete assignment to the whole set of variables $\mathcal{X}$ that satisfies all the constraints in $\mathcal{C}$.*

3.2 Robust Solutions by Feasible Neighbour Values

As mentioned in Section 2.1, we take the same limited assumptions about robustness as in [7]. In the absence of detailed information about the uncertainty, the most robust solution is the one with the maximal distance from all the dynamic bounds of the solution space. In Constraint Programming the solution spaces can be non-convex. Then, the distance to the bounds can not be computed as linear equations. Therefore, for CSPs with ordered domains, the authors stated: *"we can only ensure that a solution s is located at least at a distance d from a bound in a certain direction of the n-dimensional space if all the tuples (possible solutions) at distances lower or equal to d from s in this direction are feasible"*.

The authors of [7] define the feasible neighbour values set $\mathcal{N}_k(x, v, s, \oplus)$ by introducing a parameter that fixes the maximum distance k of the feasibility checking for a variable x and its assigned value v, with respect a partial/complete assignment s. $\mathcal{D}_s(x) \subseteq \mathcal{D}(x)$ represents the subset of domains values of the variable x that are consistent with the feasible partial assignment s. $\oplus$ is a set of operators pairs which indicate the directions (order) of the neighbour values to check. The list of operators used is $\oplus = \{\{>, +\}, \{<, -\}\}$. Each operator pair is denoted as $\oplus_i$. The set $\{>, +\}$ ($\oplus_1$) refers to values greater than v (increasing direction) and the set $\{<, -\}$ ($\oplus_2$) refers to values lower than v (decreasing direction). For each operator pair, the operator in the position j is referenced as $\oplus_{ij}$ (for instance, $\oplus_{12}$ references the operator $+$). Below, we describe the equation of the feasible neighbours values presented by the authors.

$$\begin{aligned}\mathcal{N}_k(x, v, s, \oplus) = \{ & w \in \mathcal{D}_s(x) : \exists \oplus_i, w \oplus_{i1} v \land |v - w| \le k \\ & \land \forall \oplus_z \; \forall j \in [1 \; (|v - w| - 1)], (v \oplus_{z2} j) \in \mathcal{D}_s(x)\}\end{aligned} \tag{1}$$

The first condition (first line) of Equation 1 checks that there are values w, in the domain of x ($\mathcal{D}_s(x)$), which are greater or lower than v according to the corresponding operator ($>$ or $<$) and that the distance between v and w is lower or equal to k. The second condition (second line) ensures that all values that are closer to v than w (values selected by applying the operator $+$ and/or $-$ to v with the iterator j) are also feasible values for s. If at least one of them is not feasible, the value w cannot belong to $\mathcal{N}_k(x, v, s, \oplus)$ because it must be a set of contiguous feasible neighbour values. Note that the concept of neighbour values (associated with the values whose feasibility has to be checked) differs from the neighbourhood term used in LNS (associated with the unassigned variables to optimize in each iteration).

3.3 Objective Function

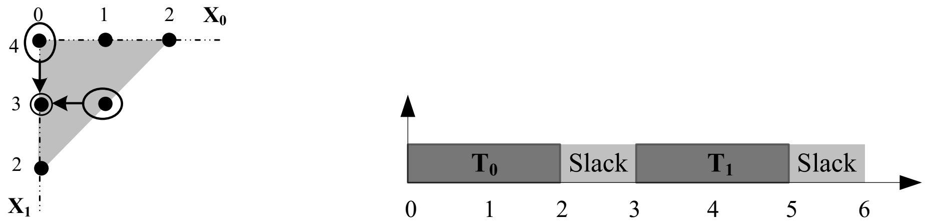
The objective function (o.f.) described in Equation 2 is the sum of the feasible neighbour values sets of all the variables of the solution s. Note that the k parameter is an input used to indicate the grade of robustness checking during the search process. Then, the k parameter determines the upper bound of the neighborhood size $\mathcal{N}_k$ (see Equation 1).

$$f(s, k, \oplus) = \sum_{x \in \mathcal{X}_s} |\mathcal{N}_k(x, s(x), s, \oplus)| \tag{2}$$

For example, in Figure 1a the grey area represents the solution space (for which the bounds can become more restrictive). The most robust solution for $k = 1$, considering only the increasing direction $\oplus_1 = \{>, +\}$, is highlighted ($x_0 = 0, x_1 = 3$). The greater feasible neighbour values (1 for x_0 and 4 for x_1) are also highlighted in the figure. Note that the variable x_0 can also be equal to 1 and the solution would remain valid (in case that 0 is not feasible anymore due to a restrictive change of the upper bound of the solution space).

Realize also of the alternative, which is that the variable x_1 can also be equal to 4 (in case that 3 is not feasible anymore for the same reason mentioned before). Therefore, the o.f. value of this solution is two (because it is the sum of its feasible neighbour values).

3.4 A case study: robust Scheduling



(a) Robust solution.

(b) Associated robust schedule.

Figure 1 Example of a robust solution for scheduling.

Without loss of generality, we selected a scheduling problem as a case study. We would like to recall that our approach works for many applications suitable for being modelled as CSPs with ordered domains in which the bounds can undergo restrictive changes. Therefore, it is not a scheduling specific approach. But considering scheduling problems as a case study is especially interesting because unexpected delays in tasks are common and such uncertainty must be considered in the solving process. Below, we present some robust-schedules concepts.

Scheduling problems consist in assigning a set of n tasks, of varying processing times, to m machines with a limited capacity. In the case study presented in Section 5, we focus on the Job Shop Scheduling Problem (JSP). This type of scheduling problem is characterized by having tasks with a specific precedence order within a job. Typically, the CSPs models of scheduling problems consider as variables the start times of the tasks and the objective is to minimize the makespan (the finish time of the last task). However, as stated in Section 1, we aim to search for robust solutions using the o.f. in Equation 2. For this reason, we consider scheduling instances that have a fixed makespan (then, their models are CSPs and not CSOPs). For example, in Figure 1b, the makespan is fixed to six.

When possible future changes over the constraints are equally probable and independent of each other, the more slacks a schedule has, the more robust it is (ideally, they are uniformly distributed and of similar size). The slack/buffer (see Figure 1b) is a time slot that can absorb a delay of the task placed before the slack without affecting the schedule. Then, we fix the set of operands in Equation 2 to $\{>, +\}$ to check the greater feasible neighbour values (equivalent to the slack/buffer between tasks in the schedules). The selection of these operands is because tasks that last longer than expected can invalidate the solution, while shorter tasks can not. Figure 1b shows the associated robust schedule with the highlighted solution of Figure 1a. Note that each task has a buffer of one unit, which corresponds to each feasible neighbour value associated with each variable. Equation 3 shows a standard measure of the slack of a schedule [19]. R^s_{Slack} is the average size of the slacks minus the standard deviation of their sizes. The β parameter regulates the importance of slack size uniformity. The authors suggest $\beta = 0.25$.

$$R^s_{Slack} = avg(slack) - \beta * std(slack). \tag{3}$$

4 LNS for Robust Solutions (LNSR)

In this section, we explain the LNS algorithm with CP for finding robust solutions for CSPs (LNSR). Figure 2 describes our LNSR algorithm (Algorithm 1). It is an iterative algorithm that takes the most robust solution found (incumbent) and creates a sub-problem by defining a neighbourhood of variables to unassign (Algorithm 2). Subsequently, it optimizes the sub-problem by propagating the constraints (Algorithm 3). For such purpose, we design a robust-value selection heuristic (Algorithm 4) that selects the most robust value based on the count of the feasible neighbour values of the already assigned variables. If a more robust solution is found, then the incumbent solution is updated. The iterative process continues by computing a new sub-problem according to a neighbourhood heuristic. In the following subsections, we present all the above-mentioned algorithms.

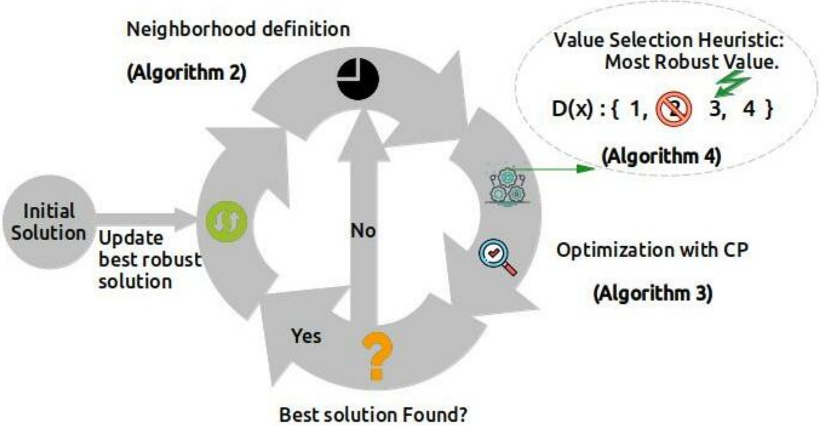


Figure 2 LNS for Robust Solutions (LNSR) (**Algorithm 1**).

4.1 The Main Iterative Process

Algorithm 1 describes the main iterative process of our LNSR. In line 1, the best solution bS and the current solution s are initialized by using a CSP solver that finds a simple solution (non-robust). In line 2, Equation 2 is applied to calculate the number of feasible neighbours values of the initial solution ($b\mathcal{N}$). In line 3, the counter of the number of LNS neighbourhoods explored ($neighC$, a.k.a. no. of restarts) is zero-initialized. Lines 4-10 contain the main LNS loop, which runs during $timeLim$. Every iteration corresponds to a neighbourhood exploration. In line 6, Algorithm 2 (*neighbourhood*) defines the new neighbourhood to explore (a new sub-problem) and returns $\mathcal{X}_s$ (the set of variables that will remain assigned) and the corresponding partial solution s. In line 7, Algorithm 3 (*opt*) explores the neighbourhood until reaching the failures limit ($fLim$). In addition, fC (zero-initialized in line 5) keeps the count of fails committed during the constraints propagation in every neighbourhood exploration. Then, *opt* returns the best solution found so far (the most robust solution found) and its corresponding number of feasible neighbour values. Note that if this algorithm does not find a better solution in the current neighbourhood explored, bS and $b\mathcal{N}$ remain the same. At the end of every iteration, the $fLim$ parameter is updated using a geometrical strategy [2] [24] (line 9).

[2] $fLim = base * (1.1^{neighC})$

Algorithm 1 *LNSR*: LNS for Robust Solutions.

```
Input: $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ , $k$, $\oplus$, $timeLim$, $fLim$
Output: $bS, b\mathcal{N}$
$bS \leftarrow s \leftarrow solve(\mathcal{P})$ // Most robust solution found
$b\mathcal{N} \leftarrow f(s, k, \oplus)$ // Best no. of feasible neighbour values found
$neighC \leftarrow 0$ // Number of LNS neighbourhoods explored
repeat
    $fC \leftarrow 0$ // Number of failures count
    $(\mathcal{X}_s, s) \leftarrow neighbourhood(\mathcal{X},\ k,\ bS,\ s)$
    $(bS, b\mathcal{N}) \leftarrow opt(\mathcal{P}, k, \mathcal{X}_s, \oplus, s, bS, b\mathcal{N}, fLim, fC)$
    $neighC{+}{+}$
    $fLim \leftarrow geometricUpdateFails(neighC)$
until $timeLim$
```

4.2 Neighborhood Selection Heuristic

Algorithm 2 describes the neighbourhood heuristic which determines the new neighbourhood to explore (a new sub-problem) for the next iteration of the Algorithm 1. First, the size of the neighbourhood ($|\mathcal{X}| - n$) is fixed to the 20% of the number of variables (line 1). We use two variable selection heuristics: random or the well-known *Wdeg/domSize* heuristic from the literature [4], which is already implemented in the ACE solver.

In LNSR, if in the previous iteration of the main algorithm a better solution has not been found (in this case the current solution s and the best solution bS are different, line 2), then, the algorithm selects the variables to unassign randomly (line 3). Selecting a random neighbourhood is a typical technique in the literature for achieving getting out from a local optimum (avoiding then the repeatedly exploration of the same neighborhood).

However, when a better solution is found (line 4), the variables with the lowest *Wdeg/domSize* are selected to remain assigned (line 5). The *Wdeg* is the number of unsatisfied constraints (during the propagation) associated with the variable. Variables with small domains tend to fail more during propagation. Therefore, the greatest *Wdeg/domSize* heuristic selects the variables that have the greatest tendency to fail; in such a way difficult sub-problems are explored (fail-first principle [4]). Note that line 5 fixes the set of variables

Algorithm 2 *neighbourhood*: NBHD heuristic.

```
  Input: $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ , $bS$, $s$
  Output: $\mathcal{X}_s$, $s$
1 $n \leftarrow round(80\%\ |\mathcal{X}|)$
2 if $s \neq bS$ then // No better solution found
3     $\mathcal{X}_s \leftarrow selRandomVars(\mathcal{X}, n)$
4 else // New best solution found
5     $\mathcal{X}_s \leftarrow selVarsLowestWdegOnDom(\mathcal{X}, n)$
6 $s \leftarrow \emptyset$
7 foreach $x$ in $\mathcal{X}_s$ do
8     $s \leftarrow s \cup \{x = bS(x)\}$
9 $propagate(\mathcal{P}, s)$
```

that will remain assigned ($\mathcal{X}_s$) to the variables with the lowest *Wdeg/domSize*, consequently, the unassigned variables are the ones with the greatest *Wdeg/domSize*. In lines 6-8, the partial assignment s is built by selecting from bS the values corresponding to $\mathcal{X}_s$.

We would like to remind that because of the inference CP process, all the constraints associated with the variables that will not belong to the new neighbourhood to explore (i.e. assigned variables) have to be propagated (line 9), and consequently, the domains of such variables will be pruned (if there are unfeasible values).

4.3 Optimization Algorithm

The *opt* algorithm (Algorithm 3) explores the new neighbourhood (previously selected by the Algorithm 2) to find a better solution than the best one found so far (bS). This optimization algorithm is similar to the one presented in [7], but we have adapted it to the LNS algorithm behaviour (only a sub-problem is optimized each time). We also include a more informed way to compute the o.f. bound and a robust-value selection heuristic (Section 4.4). The *opt* algorithm is recursive and tries to assign a variable of the sub-problem in each call.

In line 1, the algorithm selects the unassigned variable x with the greatest *Wdeg/domSize* value (explained in Algorithm 2) [4]. Then, it updates $\mathcal{X}_s$ by adding x to this set (line 2). In line 3, it saves all the domains of the variables and the $\mathcal{N}_k$ sets of the already assigned ones (since they must be restored when backtracking occurs). A loop (lines 4-21) iterates over the values in $\mathcal{D}(x)$. In line 5, the values are selected based on the robust value selection heuristic (*selMostRobVal*, Algorithm 4) or using the typical first value selection, a.k.a. lexicographical order, (*selFirstVal*). We tested several combinations of both heuristics in the evaluation section (Section 5). As is typical in CSP solvers, the search process restarts after several value assignment failures are reached ($fLim$) (second condition of line 4). In this case, LNSR would restart in a new iteration of Algorithm 1 with a new neighbourhood.

In line 6, the *propagate* procedure prunes the domains of the unassigned variables and the $\mathcal{N}_k$ sets of the assigned ones according to the constraints propagation when $x = val$. Typically, CSP solvers propagate the constraints only over the domains of the unassigned variables. However, we also need to keep updated the $\mathcal{N}_k$ sets. For this reason, we extend the *propagate* procedure so that it also prunes the $\mathcal{N}_k$ sets (Equation 1) of the previously assigned variables and creates the $\mathcal{N}_k$ set for the recently assigned one. This procedure returns *true* if the assignation is feasible, otherwise *false*. In line 7, the algorithm updates the current solution s ($x = val$), so that it can be used in the o.f. calculation.

In line 8, the algorithm calculates an upper *bound* of the o.f. (Equation 2) as the sum of the o.f. of the assigned variables (i.e. the partial assignment s) and the max. possible feasible neighbour values of the unassigned variable. The latest is calculated as the lowest value between $|\oplus| * k$ and its domain size. Thus, the maximum size of the set of neighbour values for each variable is $2k$ if $\oplus$ is composed of two operator pairs (such as in random CSPs) or k if $\oplus$ is composed of only one operator pair (such as in scheduling CSPs). Note that, as mentioned above, the $\mathcal{N}_k$ sets of the already assigned variables have to be updated after every variable assignment so that the o.f. value can be properly computed.

Then, in line 9, if the *bound* of robustness is greater than the best one found so far ($b\mathcal{N}$), it means that it is possible to find a better solution by exploring such branch (i.e. partial assignment s). In this case, the search continues by calling recursively to the *opt* algorithm (line 14). When a complete solution is found (line 10) the best solution and its associated robustness are updated, which are denoted as bS and $b\mathcal{N}$ correspondingly (lines 11-12).

However, if the *bound* of robustness is lower (line 15) or s is not feasible (line 17), the count of failures fC is incremented by one (lines 16 and 18). In both cases, this branch of the search tree is discarded since it cannot produce a better solution than the best one so

Algorithm 3 *opt*: optimization algorithm.

```
   Input: $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle, k, \mathcal{X}_s, \oplus, s, bS, b\mathcal{N}, fLim, fC$
   Output: $bS$, $b\mathcal{N}$
1  $x \leftarrow selVarGreatestWdegOnDom(\mathcal{X} \setminus \mathcal{X}_s)$
2  $\mathcal{X}_s \leftarrow \mathcal{X}_s \cup x$
3  $save(\ \mathcal{D}, \mathcal{N}_k)$
4  while $\mathcal{D}(x) \neq \emptyset \wedge (fC < fLim)$ do
5      $val \leftarrow selMostRobVal(\mathcal{P}, \mathcal{X}_s, s, x, \oplus)$ or $selFirstVal(\mathcal{P}, x)$
6      if $propagate(\mathcal{P},\ s,\ x = val)$ then
7          $s \leftarrow s \cup \{x = val\}$
8          $bound \leftarrow f(s, k, \oplus) + \sum_{y \in \mathcal{X} \setminus \mathcal{X}_s} \min((|\oplus| * k), |\mathcal{D}(y)|)$
9          if $bound > b\mathcal{N}$ then
10             if $\mathcal{X}_s = \mathcal{X}$ then
11                 $b\mathcal{N} \leftarrow bound$
12                 $bS \leftarrow s$
13             else
14                 $opt(\mathcal{P}, k, \mathcal{X}_s, \oplus, s, bS, b\mathcal{N}, fLim, fC)$
15         else
16             $fC++$ // Number of failures count
17     else
18         $fC++$ // Number of failures count
19     $restore(\mathcal{D} \setminus \mathcal{D}(x), \mathcal{N}_k))$
20     $\mathcal{D}(x) \leftarrow \mathcal{D}(x) \setminus val$
21     $s \leftarrow s \setminus \{x = val\}$
22 $restore(\mathcal{D}(x))$
23 $\mathcal{X}_s \leftarrow \mathcal{X}_s \setminus x$
```

far (Branch and Bound) [25]. Since a different branch has to be explored, it is necessary to restore all the domains, except the one of the analyzed variable ($\mathcal{D}(x)$), and the $\mathcal{N}_k$ sets of the already assigned ones (line 19), so that the propagation effect of the assignation $x = val$ is re-established. In addition, the value val is erased from $\mathcal{D}(x)$ (line 20) and s (line 21). The search will continue with the next most robust value in $\mathcal{D}(x)$ (line 5). When a wipeout occurs ($\mathcal{D}(x) = \emptyset$, in line 4), it is necessary to restore the domain of the variable x (line 22), exclude it from $\mathcal{X}_s$ (line 23) and make a backtrack to the previous variable.

4.4 Robust Value Selection Heuristic

This section describes the robust-value selection heuristic, which is novel and represents a contribution of this paper. The main idea of this heuristic is to assign a value to the current variable to assign that has the lowest negative impact over the robustness of the partial assignment s (composed by the previous variables assigned). This means that the value assigned will be the one that less reduces the total number of feasible neighbours of s.

In line 1 of Algorithm 4 , the Q queue is initialized with the assigned variables that share a constraint with the variable x (the variable to assign). We use $var(c)$ to denote the scope of the constraint c (i.e. the variables involved in such constraint). Line 2 initializes the set

Algorithm 4 *selMostRobVal*: value sel. heuristic.

```
  Input: $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ , $\mathcal{X}_s$, $s$, $x$, $\oplus$
  Output: $bestValue$
1 $Q \leftarrow \{y, \forall y \in var(c), y \in \mathcal{X}_s \wedge x \in var(c)\ , \forall c \in \mathcal{C}\}$
2 $sum\mathcal{N}_x \leftarrow \emptyset$ // var(c) is the scope of c
3 foreach $val \in \mathcal{D}(x)$ do
4     $s \leftarrow s \cup \{x = val\}$
5     $sum\mathcal{N}_x(val) = \sum_{y \in Q} |\mathcal{N}_k(y, s(y), s, \oplus)|$ // Eq.2
6     $s \leftarrow s \setminus \{x = val\}$
7 $bestValue \leftarrow val$ , $\max(sum\mathcal{N}_x(val))\ \forall val \in \mathcal{D}(x))$
```

$sum\mathcal{N}_x$, which will be used to determine the estimation of robustness of the values of the domain of the variable x. Subsequently, a loop through all possible values val in the $\mathcal{D}(x)$ begins (line 3). In line 4, the value val is added to the partial solution to observe its effect over the neighbour feasible values of the assigned variables. This calculation is similar to the objective function (Equation 1) but considering only the variables in Q instead of all the variables (line 5). In line 6, the previously added value is erased from the partial solution s to continue with the next iteration (the next value of $\mathcal{D}(x)$). Finally, the value with the greatest number of feasible neighbour values in $sum\mathcal{N}_x$ is selected as $bestValue$ (line 7). The lexicographical order resolve ties.

5 Evaluation

For the evaluation, we implemented LNSR and the complete algorithm from [7] (we denote it as B&B) and we embedded them into the ACE CSP solver[3] (previously ABSCON) [15]. We used ACE for the constraint propagation, the restarts, the weighted degree (*Wdeg*) computation and the B&B. The experiments were run on a 2xIntel(R) Xeon(R) CPU (E5620 @ 2.40GHz) and 32 GB of RAM memory running Ubuntu Server 18.04. We run three times each LNSR configuration for each instance (because LNSR sometimes selects random variables, see Algorithm 2) and we present the average results.

5.1 Experimental Settings

We performed experiments for $k = 1, k = 3$ and $k = 5$. For the LNSR algorithm, we empirically tested several constant sizes of the neighbourhood, 20% was the most adequate. We used the same configurations for the common parameters of B&B and LNSR. The function in line 8 of Algorithm 3 was used as the upper bound of robustness. For the geometric restart strategy, we use the formula $fLim = base * (1.1^{neighC})$ with $base$ initially equal to 100 and double it every 50 restarts. We evaluated the following value selection heuristics:

- *First* (LNSR-1 and B&B-1): first value selection (lexicographical order).
- *First-Rob* (B&B-2): first value selection combined with the robust value selection heuristic: $selMostRobVal$ (Algorithm 4). In the beginning, the first value selection heuristic is used in every restart only until reaching the first solution, then $selMostRobVal$ is used for the rest of the search (including the backtracking).

[3] https://github.com/xcsp3team/ace

- *First-Rob* variant (LNSR-2 and B&B-2’): this is a *first-rob* variant. The first solution is computed with the first value selection heuristic. Then, for LNS $selMostRobVal$ (Algorithm 4) is applied to the variables in the LNS neighbourhood. This is equivalent to applying $selMostRobVal$ to the last n variables to assign in B&B, where n is the size of the LNS neighbourhood. Note that B&B-2’ has the same value selection heuristic as LNS-2, while B&B-2 does not.

We also evaluated *Rob-Rob* value selection, where the first solution is also computed selecting the robust value ($selMostRobVal$). However, its performance was extremely poor, especially in scheduling instances. The problem is that selecting the most robust values for all the variables of the scheduling leads to dead-ends because such assignments exceed the maximum makespan allowed (especially in the critical path). Therefore, this algorithm configuration could not find even the first solution in the allocated time.

5.2 Evaluation with General CPSs

In this section, we present the results of our algorithm using general CSP instances generated with the uniform random generator URBCSP[4]. The random instances used have 110 variables, domain sizes of 120 and 1119 constraints (corresponding to the 20% of all the possible binary constraints). We use three different tightness values to generate each one of the instances: 1.3, 1.6 and 1.9. We fixed a time limit of 10 min.

The results are presented in Table 1. We show the number of solutions found ($\#S$), the number of visited nodes ($\#Nodes$), the number of failed assignments ($\#FAssigns$), the number of variables with at least one neighbour value ($\#varR$), the number of neighbour values of the solution ($\#N$) and a measure of the distribution of the feasible neighbour values across the solution ($Ndist$). The $Ndist$ measurement is computed with Equation 3, but considering the number of feasible neighbour values instead of the slack.

Table 1 Comparison of value selection heuristics and algorithms in random CSPs.

k	***Heuristic***	***Algorithm***	***#S***	***#Nodes***	***#FAssigns***	***#varR***	***#N***	***Ndist***
1	First	B&B-1	7.7	5291006.0	1928269.7	39.0	39.0	0.236
		LNSR-1	16.0	4076714.4	1619716.0	47.3	47.7	0.309
	First-Rob	B&B-2	7.3	5373612.3	1978766.7	39.7	39.7	0.241
		B&B-2’	9.0	4406818.3	1620720.0	44.0	46.3	0.290
		LNSR-2	**17.0**	3163868.8	1285432.4	**62.7**	**74.3**	**0.520**
3	First	B&B-1	8.7	4837787.3	2022192.0	36.7	43.7	0.245
		LNSR-1	15.3	3912019.8	1674863.6	43.3	51.3	0.308
	First-Rob	B&B-2	10.0	4838004.7	2045275.3	40.7	46.0	0.270
		B&B-2’	6.7	4010733.7	1699203.3	41.7	48.0	0.287
		LNSR-2	**18.0**	2920458.8	1245622.8	**59.0**	**88.7**	**0.581**
5	First	B&B-1	7.3	4803303.3	2067019.0	38.0	42.3	0.244
		LNSR-1	18.0	3845131.6	1650786.6	44.3	54.3	0.327
	First-Rob	B&B-2	9.3	4683463.0	1965024.7	38.0	43.7	0.249
		B&B-2’	6.7	4085214.7	1746265.3	41.7	49.0	0.291
		LNSR-2	**25.0**	2946410.4	1264325.4	**60.0**	**90.3**	**0.590**

Table 1 clearly shows that LNSR-2 has the best results for the three robustness measurements for all the k values. LNSR-2 finds solutions with the greatest number of robust variables ($\#varR$), which are the variables that have at least one feasible neighbour value.

[4] `http://www.lirmm.fr/~bessiere/generator.html`

This robustness measure shows how many variables of the solution would resist a perturbation of magnitude one in the bounds of the solution space (delimited by the constraints and the domains). LNSR-2 also obtains the greatest sum of the feasible neighbour values across all the variables (a.k.a. no. of neighbours, $\#N$). In addition, the solutions found by LNSR-2 have the best distribution of the feasible neighbour values through all the variables ($Ndist$). These results confirm our hypothesis that our LNSR algorithm performs better than B&B.

Note that the LNSR algorithm finds a higher number of solutions ($\#S$) than the B&B algorithm (especially, LNSR-2). Accordingly, the number of failed assignments ($\#FAssigns$) is lower. We believe that this is due to the diversification in the search space exploration that LNSR offers by exploring neighbourhoods around the current most robust solution. On the contrary, B&B gets stuck exploring very specific branches of the search tree.

Another important conclusion from the evaluation is that our heuristic for selecting the most robust values (Algorithm 4) has a very good performance, especially when used in LNSR. For the B&B algorithm, there is an improvement of B&B-2' over B&B-2 and from both over B&B-1. We believe that the robust value selection heuristic did not have a big impact on B&B because the instances are large-scale and therefore, as mentioned above, B&B get stuck exploring very specific branches of the search tree. The improvement is even more pronounced in the case of the LNSR algorithm. In almost all the cases, LNSR-2 obtains a robustness increment of about 70% of the values obtained by LNSR-1 (for the three robustness measures). This fact shows the usefulness of our robust value selection heuristic to guide the search toward more robust solutions.

The number of visited nodes ($\#Nodes$) is lower for B&B-2' and LNSR-2 than the other configurations. It is due to the most robust value selection heuristic is more costly than the first value selection heuristic. Even if our heuristic is costly, it has shown to be very effective, especially combined with our LNSR (LNSR-2).

5.3 Evaluation with Scheduling Instances

We evaluated our LNRS algorithm using six Taillard Job Shop scheduling instances with 300 tasks each[5]. Recall that these instances are modeled as CSPs (rather than CSOPs) because the makespan is fixed. The variables of the CSPs models are the starting and ending times of the tasks and the domains are limited by their possible maximum ending times. Note that the variables associated with the ending times are excluded from the robustness search since they are just equal to the start time variables plus the duration of the tasks. Equality constraints are used to ensure that the end time of a job is equal to the end time of its last operation. Precedence constraints are used to indicate the order of the operations in every job and non-overlapping constraints are used to indicate the correspondence between operations and the single capacity resources. Finally, the last constraints are used to indicate the minimum possible duration of every job.

In these executions, we fixed a time limit of 20 min. The average results of the evaluation are presented in Table 2. We show the same measurements as in the previous section, except for $\#B$, which is the number of buffers and R^s_{Slack}, which measures the buffers' distribution (see Equation 3). Note that $\#B$ is equivalent to $\#varR$ and R^s_{Slack} is equivalent to $Ndist$.

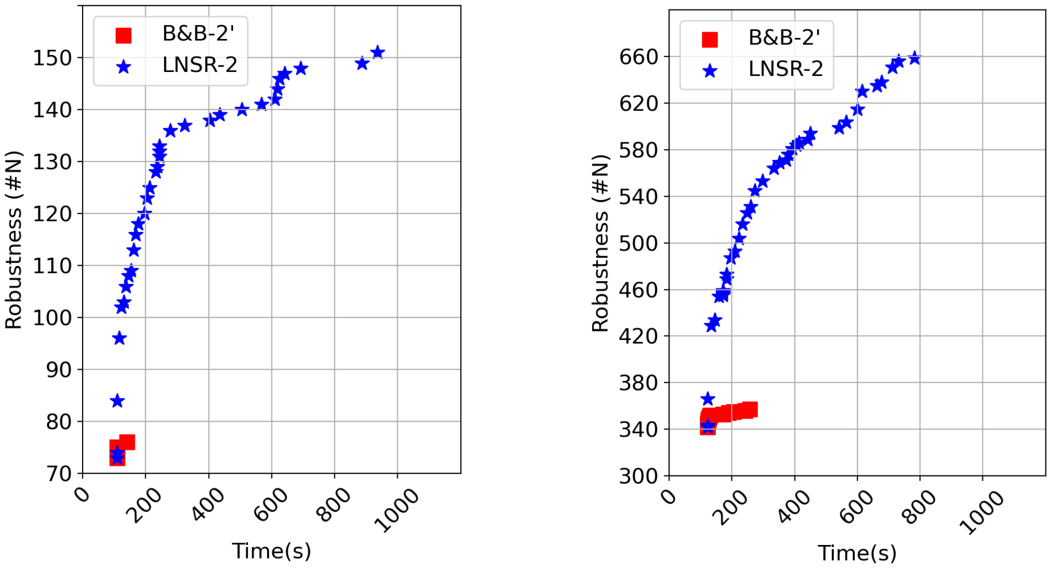
The results obtained in the scheduling instances are similar, in terms of the approaches ranking, to the results obtained in general CSPs instances. LNSR-2 outperforms the other approaches in the robustness measurements. Although in the scheduling instances, the differences between the three variants of B&B are lower than in the random CSPs (Table 1).

[5] Instances from `http://www.xcsp.org/instances/`: Taillard-js-020-15-{0,2,3,6,8,9}. Instances {1,4,5,7} could not be solved in the given cut-off time.

Table 2 Comparison of value selection heuristics and algorithms in Taillard-js-020-15 instance.

k	*Heuristic*	*Algorithm*	*#S*	*#Nodes*	*#FAssigns*	*#B*	*#N*	R^s_{Slack}
1	First	B&B-1	2.5	5752220.8	2389769.0	75.7	75.7	0.144
		LNSR-1	14.2	3861320.2	501682.4	87.7	87.7	0.179
	First-Rob	B&B-2	2.5	5760396.8	2390411.2	75.7	75.7	0.144
		B&B-2'	2.5	5780979.8	2400829.3	75.7	75.7	0.144
		LNSR-2	**30.0**	3432879.6	508581.8	**157.2**	**157.2**	**0.400**
3	First	B&B-1	5.3	5156986.5	2125819.3	75.7	219.3	0.412
		LNSR-1	**40.0**	3153637.7	491531.3	88.8	257.5	0.524
	First-Rob	B&B-2	4.0	5146224.7	2118215.8	75.7	219.5	0.413
		B&B-2'	5.3	5167244.8	2132102.5	75.7	219.3	0.412
		LNSR-2	30.8	2739318.9	488374.4	**143.5**	**417.7**	**1.024**
5	First	B&B-1	8.0	4788771.7	1965952.7	75.7	352.5	0.656
		LNSR-1	**54.8**	2734267.1	484041.2	86.7	405.8	0.809
	First-Rob	B&B-2	5.5	4634154.7	1891387.0	75.7	352.7	0.656
		B&B-2'	8.0	4717497.0	1934862.7	75.7	352.5	0.656
		LNSR-2	33.9	2352468.3	478499.9	**139.6**	**661.7**	**1.599**

Table 2 shows that the robustness of the solutions obtained by LNSR-2 is about two thirds more than the other approaches (for $\#B$ and $\#N$). Regarding the R^s_{Slack} measurement, LNSR-2 achieves results close to double the other approaches. This result indicate that LNSR-2 offers a better distribution of robustness (number of feasible neighbour values) across all the variables.



(a) k=1

(b) k=5

Figure 3 Solutions found over the time(s) and their robustness (Taillard-js-020-15-9 instance).

The results of the scheduling evaluation show a very high difference in the total number of feasible neighbours achieved ($\#N$) when the k is increased (for LNSR-2, it is more than 200 neighbours for each k increment of 2 units, i.e. $k = 3, k = 5$). Such increment is much lower in the random CSPs (see Table 1). We believe that the reason could be associated with the tightness of the instances. The higher it is, the lower the likelihood of finding a great number of neighbours.

We also present Figures 3a and 3b which show the solutions found over time for B&B-2' and LNSR-2 for $k = 1$ and $k = 5$ (for a particular instance and execution). For other instances, executions and k's, the graphs are similar to the presented here. Note that LNSR-2 finds a much higher number of solutions than B&B-2'. The robustness of the solutions found by LNSR-2 increments quickly over time, specially at the beginning of the execution. However, B&B-2' finds solutions that have similar robustness between them. In addition, B&B-2' is unnable to improve the robustness after a short time (less than 200 sec. for k=1 and less than 300 sec. for k=5). We believe that this is because the B&B approaches spend a lot of time searching in the same branches of the search tree while LNSR diversifies more the search by exploring a large number of neighbours of the current most robust solution.

6 Conclusions and Future Work

In this paper, we presented an LNS algorithm with CP and B&B for searching for robust solutions for CSPs (LNSR). LNSR considers the robustness concepts and assumptions as in [7]. And therefore, it computes robust solutions for large-scale CSPs that have a high number of feasible neighbour values (Algorithms 1 and 2, from Section 4). In addition, we presented a robust-value selection heuristic (Algorithm 4, from Section 4) for effectively guiding the search towards more promising branches of the search space (that are more likely to contain more feasible neighbour values solutions). Specifically, the focus is on problems that do not have associated detailed information about the uncertainty and with ordered domains.

In this paper, we used general CSPs instances and scheduling problems as a case study (due to their eligibility for such criteria). The experimental evaluation shows that our LNSR algorithm, combined with our robust value selection heuristic, outperforms the previous approach from the state-of-the-art. This good performance is especially usefull in large-scale problems in which the computation time available for finding a solution is limited and/or when several solutions must be re-computed over time (due to the dynamism associated with the real-world application problems).

As a future work, among others, we would like to explore other domains, such as data centres, sports, etc. In addition, we plan to apply adaptive strategies to tune the neighbourhood size, the variables selection and the limit of failures allowed during the neighbourhood exploration. We believe that it could be useful to propose a variable selection heuristic that considers the robustness of the variables (for example, combining a robustness measurement with *Wdeg*). We will explore ideas such as Cost-Impact based variable selection, We will also consider the combination of several heuristics into a portfolio, in the same vein as previous "adaptive" LNS approaches from the literature.

Furthermore, we would like to explore different ways of applying our LNSR to CSOPs by using multi-objective strategies and computing the Pareto frontier. In addition, we believe that it is also interesting to evaluate the performance of accepting less robust solutions in some iterations of the LNSR algorithm (as proposed in [14, Chapter 4]).

References

1 A. Charnes and W. W. Cooper. Chance-Constrained Programming. *Management Science*, 6(1):73–79, 1959.

2 J. Christopher Beck, T. K. Feng, and Jean Paul Watson. Combining constraint programming and local search for job-shop scheduling. *INFORMS Journal on Computing*, 23(1):1–14, 2011.

3 Aharon Ben-Tal, Laurent El Ghaoui, and Arkadi Nemirovski. *Robust optimization*. Princeton University Press, 2009.

4 Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. *Frontiers in Artificial Intelligence and Applications*, 110:146–150, 2004.

5 Tom Carchrae and J. Christopher Beck. Principles for the design of large neighborhood search. *Journal of Mathematical Modelling and Algorithms*, 8(3):245–270, 2009.

6 Laura Climent, Richard J. Wallace, Barry O'Sullivan, and Eugene C. Freuder. Extrapolating from limited uncertain information in large-scale combinatorial optimization problems to obtain robust solutions. *International Journal on Artificial Intelligence Tools*, 25:1–21, 2016. In this paper a way to use extrapolation to enable Stochastic approach with limited knowledge is treated. Important: Ordered Domains. doi:10.1142/S0218213016600058.

7 Laura Climent, Richard J. Wallace, Miguel A. Salido, and Federico Barber. Robustness and stability in constraint programming under dynamism and uncertainty. *Journal of Artificial Intelligence Research*, 49:49–78, 2014.

8 Thierry Denœux. Belief functions induced by random fuzzy sets: A general framework for representing uncertain and fuzzy evidence. *Fuzzy Sets and Systems*, 1:1–29, 2020.

9 Ivan Dotu, Manuel Cebrián, Pascal Van Hentenryck, and Peter Clote. Protein Structure Prediction with Large Neighborhood Constraint Programming Search. In Peter J. Stuckey, editor, *Principles and Practice of Constraint Programming*, pages 82–96. Springer, 2008.

10 Helene Fargier, Jerome Lang, and Thomas Schiex. Mixed constraint satisfaction: a framework for decision problems under incomplete knowledge. *Proceedings of the National Conference on Artificial Intelligence*, 1(January 1996):175–180, 1996.

11 Markus P J Fromherz. Constraint-based scheduling. *Proceedings of the American Control Conference*, pages 3231–3244, 2001.

12 Luca Di Gaspero, Andrea Rendl, and Tommaso Urli. Balancing bike sharing systems with constraint programming. *Constraints*, 21(2):318–348, 2016.

13 Emmanuel Hebrard. *Robust Solutions for Constraint Satisfaction and Optimisation under Uncertainty*. PhD thesis, University of New South Wales, 2007.

14 Michellgendreauu · Jean-Yvesspotvin. *Handbook of Metaheuristics*. International Series in Operations Research & Management Science, third edit edition, 2010.

15 Christophe Lecoutre and Sebastien Tabary. Abscon 112 : towards more robustness. In *3rd International Constraint Solver Competition*, pages 41–48, 2013.

16 Dario Pacino and Pascal Van Hentenryck. Large neighborhood search and adaptive randomized decompositions for flexible jobshop scheduling. *IJCAI International Joint Conference on Artificial Intelligence*, pages 1997–2002, 2011.

17 Laurent Perron, Paul Shaw, and Vincent Furnon. Propagation guided Large Neighbourhood Search. In Mark Wallace, editor, *Principles and Practice of Constraint Programming*, volume 3258, pages 469–481, Toronto, 2004. Springer.

18 Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1520:417–431, 1998.

19 Michele Surico, Uzay Kaymak, David Naso, and Rommert Dekker. Hybrid Meta-Heuristics for Robust Scheduling. *ERIM Report Research in Management*, 2006.

20 Nihat Engin Toklu. *Matheuristics for Robust Optimization – Application to Real-World Problems*. PhD thesis, Università della Svizzera Italiana, 2014.

21 Behnam Vahdani, Reza Tavakkoli-Moghaddam, Fariborz Jolai, and Arman Baboli. Reliable design of a closed loop supply chain network under uncertainty: An interval fuzzy possibilistic chance-constrained model. *Engineering Optimization*, 45(6):745–765, 2013.

22 Gérard Verfaillie and Narendra Jussien. Constraint solving in uncertain and dynamic environments: A survey. *Constraints*, 10(3):253–281, 2005.

23 Toby Walsh. Stochastic constraint programming. *Proceedings of the 15th Eureopean Conference on Artificial Intelligence*, pages 111–115, 2002.

24 Huayue Wu. *Randomization and Restart Strategies*. PhD thesis, University of Waterloo, 2006.

25 Weixiong Zhang. Branch-and-Bound Search Algorithms and Their Computational Complexity. Technical report, Information Sciences Institute, Marina del Rey, California, 1996.

Scheduling the Equipment Maintenance of an Electric Power Transmission Network Using Constraint Programming

Louis Popovic ✉
Computer Engineering and Software Engineering Department, Polytechnique Montréal, Canada

Alain Côté ✉
IREQ, Varennes, Canada

Mohamed Gaha ✉
IREQ, Varennes, Canada

Franklin Nguewouo ✉
Hydro-Québec, Canada

Quentin Cappart ✉
Computer Engineering and Software Engineering Department, Polytechnique Montréal, Canada

Abstract

Modern electrical power utilities must maintain their electrical equipment and replace it when the end of its useful life arrives. The Transmission Maintenance Scheduling (TMS) problem consists in generating an annual maintenance plan for electric power transportation equipment while maintaining the stability of the network and ensuring a continuous power flow for customers. Each year, a list of equipment (power lines, capacitors, transistors, etc.) that needs to be maintained or replaced is available and the goal is to generate an optimal maintenance plan. This paper proposes a constraint-based scheduling approach for solving the TMS problem. The model considers two types of constraints: (1) constraints that can be naturally formalized inside a constraint programming model, and (2) complex constraints that do not have a proper formalization from the field specialists. The latter cannot be integrated inside the model due to their complexity. Their satisfaction is thus verified by a black box tool, which is a simulator that mimics the impact of a maintenance schedule on the real power network. The simulator is based on complex differential power-flow equations. Experiments are carried out at five strategic points of Hydro-Québec power grid infrastructure, and involve more than 200 electrical equipment and 300 withdrawal requests. Results show that the model is able to comply with most of the formalized and unformalized constraints. It also generates maintenance schedules within an execution time of only a few minutes. The generated schedules are similar to the ones proposed by a field specialist and can be used to simulate maintenance programs for the next 10 years.

2012 ACM Subject Classification Applied computing → Operations research

Keywords and phrases Transmission maintenance scheduling, Electric power network, Constraint programming

Digital Object Identifier 10.4230/LIPIcs.CP.2022.34

1 Introduction

Modern electrical power utilities must maintain their electrical equipment and replace them as they reach the end of their useful life. Asset management is becoming strategically important for transmission utilities around the world. The International Electrotechnical Commission (IEC) has recognized that a specific power network approach to monitoring and managing assets is required [18]. To respond to these challenges, Hydro-Québec, a public utility company operating in Quebec, started the PRIAD-project, which aims to

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 34; pp. 34:1–34:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

develop an integrated decision support system including predictive modeling methods for asset management [9]. This project includes different modules for cloud data warehouses, asset behavior, reliability, transmission system simulation, risk and optimization. Such initiatives to develop decision systems for identifying and prioritizing the replacement and maintenance of electrical equipment with an asset risk framework have been undertaken by many other utilities [3]. The main objective of every electric power system is to transport electricity from the generating units to the load centers in a secure manner. To do so, one of the main tasks of the *network control center* (NCC) is to use a contingency approach to ensure that maintenance activities do not lead to interrupted power supply [19]. To reduce the number of failures and improve power network reliability, assets are periodically removed from the grid network for preventive maintenance. Scheduling the preventive maintenance activities of electrical power utilities relates to two well-known problems: (1) the *generator maintenance scheduling* (GMS) problem, and (2) *the transmission maintenance scheduling* (TMS) problem. Solving both of these problems efficiently is crucial for network reliability and fluidity. However, they are also NP-hard due to the complexity of the constraints included. On the one hand, many approaches have been proposed for solving the GMS problem. For a recent review of these studies, the reader is referred to the survey proposed by Froger et al. [8].

On the other hand, the TMS problem has been less studied in the literature. The goal is to generate an annual maintenance plan for electric power transportation equipment while maintaining the stability of the network and ensuring a continuous power flow for customers. It is noteworthy to highlight that the TMS problem is limited to transmission equipment and does not include, for instance, distribution equipment. This is intended as most of the distribution equipment are not maintained and are used as run-to-fail, yielding a specific resolution process. Pandžić et al. [15] proposed a bi-objective mathematical model for solving a TMS problem involving only transmission lines. The idea is to compute an appropriate trade-off between ensuring transmission capacity and minimizing the maintenance impact on power system operation and then the market. To do so, they proposed to recast a non-linear formalization of the problem into a mixed-integer linear program, and to solve it using a standard branch-and-cut algorithm. In addition, Mei et al. [14] proposed another mixed-integer program that aims to maximize the maintenance willingness of transmission lines under security and capacity constraints of the transmission power system. To improve the computational efficiency, the authors proposed a machine learning approach accelerating the branching procedure of the solving algorithm. These two works applied the timetable obtained to IEEE 24-bus and 30-bus reliability test systems, which are relatively simple and not representative of networks involving complex electrical constraints. More recently, Rocha et al. [17] proposed a mixed-integer program for solving the TMS problem on a IEEE-24 system, similar to the one considered in this paper. Unlike the previous works, they consider a complete transmission grid and not only transmission lines. Solving is carried out by splitting the initial problem into two smaller optimization problems with the use of Benders decomposition [16]. However this approach does not consider advanced constraints related to the limitations of power transit inside the grid.

To the best of our knowledge, there are no related works that solve TMS problems on a complete transmission grid with various electrical equipment and transit-power constraints. Ensuring that transit-power limits are never violated is a critical concern in practice. This motivates our work to solve a TMS problem from the point of view of NCC operation. Each year, the NCC operator receives the annual maintenance plan with a suggested starting maintenance date and duration. The operator tries to satisfy the proposed maintenance plan, while satisfying electrical constraints known as *transit-power limits*. Such constraints guarantee power network stability during maintenance. In addition, a list of withdrawal rules that ensure stable power system operation is available. These rules, based on expert knowledge and power network analysis, represent restrictions on equipment that can or cannot be removed simultaneously from the grid. These constraints can be naturally formalized inside a mathematical model. However, transit-power constraints are trickier to handle. The theoretical values of the transit-power limits inside a power network subject to inactive equipment are based on complex constrained differential power-flow equations and are tedious to compute. For this reason, these constraints do not have a proper closed-form expression from the field specialists and cannot be easily integrated inside a mathematical model. In practice, the satisfaction of such constraints can be verified by a simulator that mimics the impact of a maintenance schedule on the real power network.

Based on this context, the goal of this paper is to find the optimal periods for removing specific transmission equipment from the grid for maintenance without impeding energy delivery. By doing so, we aim to provide planners with insight in order to help them in their decisions, which are currently done manually based on their field expertise. The complexity of the maintenance task will be reduced and they will be able to dedicate a specific focus on the most challenging aspects of the task. The specific contributions of this paper are as follows: (1) an approach based on constraint programming (CP) for solving a TMS problem on a transmission grid with transit-power constraints; (2) the use of a black-box simulator approximating the electrical impact for each proposed maintenance schedule in order to validate the satisfaction of transit-power constraints; (3) a two-step objective function dedicated to maximizing the balance of the schedule and to maximizing the overlap of withdrawal requests involving the same equipment; and (4) experiments on five strategic points of a real power grid infrastructure that involves more than 200 pieces of electrical equipment and 300 withdrawal requests. Results show that the model is able to comply with most of the power-transit constraints. The maintenance schedules are generated within an execution time of few minutes and are similar to the ones proposed by field specialists. The next section formalizes the TMS problem and introduces the constraint programming model that we have designed. The solving process is then described in Section 3. Lastly, experiments and results are presented in Section 4.

2 Modelling the Transmission Maintenance Scheduling Problem

The goal is to generate an annual maintenance plan of withdrawal requests for electric power transportation equipment (power lines, capacitors, transistors, etc.) while maintaining the stability of the network. Let W be the set of withdrawal requests that must be scheduled inside the planning horizon, and let E be the set of electrical equipment involved in the network grid. Each withdrawal request $w_i \in W$ has a duration $l_i \in \mathbb{N}$, a list of equipment $E_i \in 2^E$ to withdraw, and a size $n_i \in \mathbb{N}$, corresponding to the number of equipment pieces related to the request ($n_i = |E_i|$). Each equipment $e_j \in E$ can be associated to one or several withdrawal requests, indicating that the equipment e_j must be withdrew when the request is fulfilled. We use the notation $e^i_k \in E_i$ to refer to the k-th equipment associated to the withdrawal request w_i. Similarly, $id(e^i_k) \in E$ and $ch(e^i_k) \in \mathbb{R}$ refer to the equipment identifier of the k-th equipment associated to the withdrawal request w_i, and the corresponding electrical charge $[Mvar]$[1], respectively. Finally, the planning horizon is defined as the days

[1] Megavolt-ampere of reactive power; an AC electrical measurement unit.

between d_0 and d_m. The annual period during which withdrawals are permitted is limited from March 15th to November 15th. This gives 245 consecutive days ($m = 245$). All withdrawal requests must be started and finished within this horizon. No maintenance is allowed outside this period. The reason is that the peak of electrical power consumption in Quebec happens during winter. The parameters introduced are summarized in Table 1.

Table 1 List of parameters used in the constraint programming model.

Entity	Parameter	Description
Request (W)	l_i	Duration (in days) of the withdrawal request w_i
	n_i	Number of equipment to withdraw for the request w_i
	E_i	Set of equipment to withdraw for the request w_i
Equipment (E)	$id(e^i_k)$	Identifier of the k-th equipment of request w_i
	$ch(e^i_k)$	Electrical charge $[MVar]$ of the k-th equipment of request w_i
Horizon	d_0	First day allowed for the maintenance
	d_{245}	Last day allowed for the maintenance

2.1 Decision Variables

We model this problem as a constraint-based scheduling model with time-interval variables, also referred to as *activities*, using the formalism proposed by Laborie et al. [11, 13]. Each withdrawal request $w_i \in W$ is modelled as an activity and is composed of four variables: a start time $s(w_i)$, a duration $d(w_i)$, a completion time $c(w_i)$, and a binary execution status $x(w_i)$. In our case, the duration of each request is known ($d(w_i) = l_i$), and all the requests must be executed, yielding $x(w_i) = 1$ for all $w_i \in W$. Each piece of equipment $e^i_k \in E_i$ related to a withdrawal request w_i is also associated to an activity. Then, a situation involving 10 requests and 20 pieces of equipment will generate at most 200 activities, as a specific piece of equipment can be involved in several withdrawal requests. As a simple synchronization constraint, an equipment item must be withdrawn during the same period as its request. The domain of all the activities are presented below. A visualization of the decision variables and the temporal relations is proposed in Figure 1.

$$\forall w_i \in W : \begin{cases} s(w_i) \in [d_0, d_{245} - l_i] \\ d(w_i) = l_i \\ c(w_i) = s(w_i) + d(w_i) \\ x(w_i) = 1 \end{cases} \qquad \forall e^i_k \in E_i : \begin{cases} s(e^i_k) = s(w_i) \\ d(e^i_k) = d(w_i) \\ c(e^i_k) = c(w_i) \\ x(e^i_k) = x(w_i) \end{cases} \tag{1}$$

2.2 Constraints

The model leverages the *cumul function* introduced in constraint-based scheduling by Laborie et al. [13]. Briefly, such a function is used to represent the accumulated consumption of a resource by activities over a timing horizon. When a new activity is started, the consumption of the resource increases. Similarly, the consumption goes down when the activity is completed. This behaviour is related to the *cumulative* global constraint [2, 10]. Besides, our model is based on the *alwaysIn* and *noOverlap* constraints. Following the formalization of Laborie et al. [13], they are defined as follows:

- `alwaysIn`$(f, u, v, h_{min}, h_{max})$ ensures that the accumulated consumption of the cumul function f remains between h_{min} and h_{max} inside the interval $[u, v)$.
- `noOverlap`(A) ensures that the activities $a \in A$ do no overlap in time.

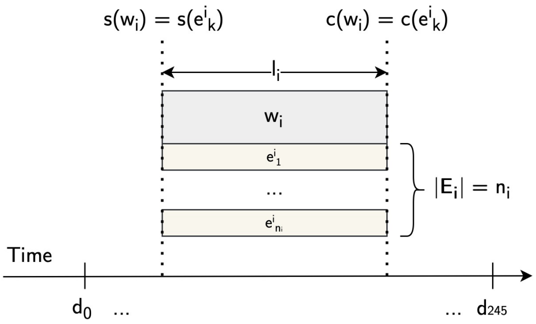


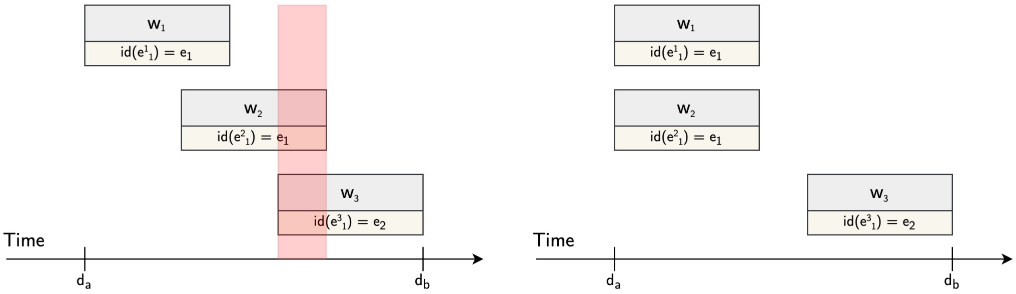
Figure 1 Illustration of the decision variables considered in the model.

Four constraints are involved in our model, two of them are based on *alwaysIn* constraint and the other twos are based on *noOverlap* constraint. The remaining of this section is dedicated to describe them.

Constraint 1: Limitation on Simultaneous Equipment Withdrawals. Let $S \in 2^E$ be an arbitrary set of equipment. This constraint states that a maximum of h pieces of equipment from the set S can be withdrawn together between the days d_a and d_b. We introduce a cumul function $f_1 : [d_0, d_{245}] \times S \to \mathbb{N}$ indicating the number of equipment items from S that are currently withdrawn for each time step of the planning horizon. The restriction is then modelled using the *alwaysIn* constraint [1] as follows.

$$\texttt{alwaysIn}(f_1, d_a, d_b, 0, h) \tag{2}$$

It ensures that the number of equipment items withdrew from S (returned by f_1) is always included between the range $[0, h]$ during the time interval $[d_a, d_b)$. A valid and an invalid solution for the configuration $S = \{e_1, e_2\}$ and $h = 1$ is illustrated in Figures 2a and 2b.



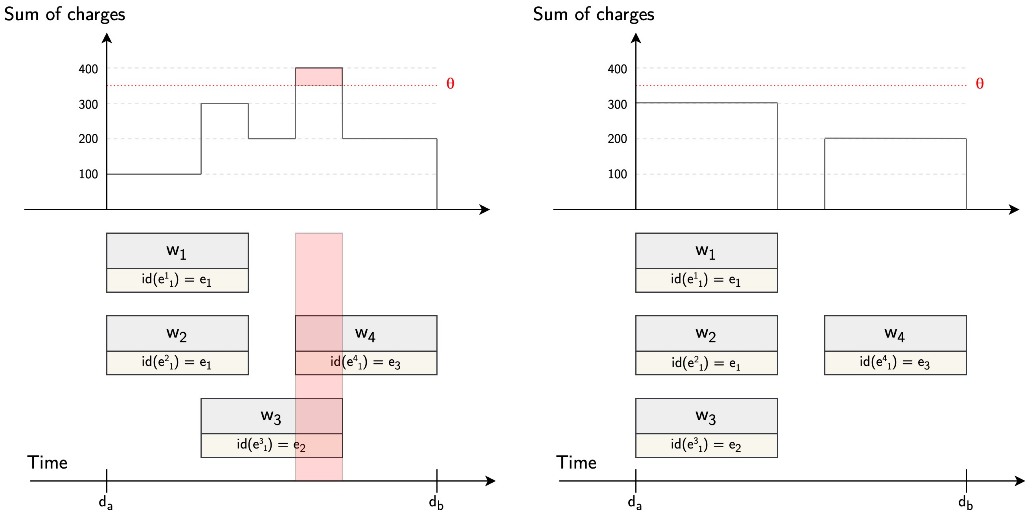
(a) Example of an invalid solution. **(b)** Example of a valid solution.

Figure 2 Illustration of Constraint 1 (limitation on simultaneous equipment withdrawal).

Constraint 2: Limitation on the Electrical Charge during Withdrawals. Let $S \in 2^E$ be an arbitrary set of equipment and θ a threshold of an electrical charge. This constraint states that the sum of the charges of the withdrawn equipment from S must always be below θ. We introduce a cumul function $f_2 : [d_0, d_{245}] \times S \to \mathbb{N}$ indicating the accumulated charge of the equipment, i.e. $\sum_{e \in S} ch(e)$, that is currently withdrawn for each time step of the planning horizon. The constraint is modelled as follows.

$$\texttt{alwaysIn}(f_2, d_0, d_{245}, 0, \theta) \tag{3}$$

It ensures that the electrical charge returned by f_2 is always included between the range $[0, \theta]$ during the complete planning horizon $[d_0, d_{245})$. A valid and an invalid solution for the configuration $S = \{e_1, e_2, e_3\}$, $\theta = 350$, $ch(e_1) = 100$, $ch(e_2) = 200$, and $ch(e_3) = 200$ is illustrated in Figures 3a and 3b.



(a) Example of an invalid solution. **(b)** Example of a valid solution.

Figure 3 Illustration of Constraint 2 (limitation on the electrical charge during withdrawals).

Constraint 3: No Overlap on Equipment Withdrawals. Let $\Lambda = \{S_1, S_2, \ldots, S_K\}$ be a set containing K sets of equipment $S_k \in 2^E$. This constraint states that equipment coming from different sets of Λ cannot be withdrawn together. Only equipment included in the same set or that are identical (same identifier) can be withdrawn together. The *noOverlap* constraint [1] is used for this purpose.

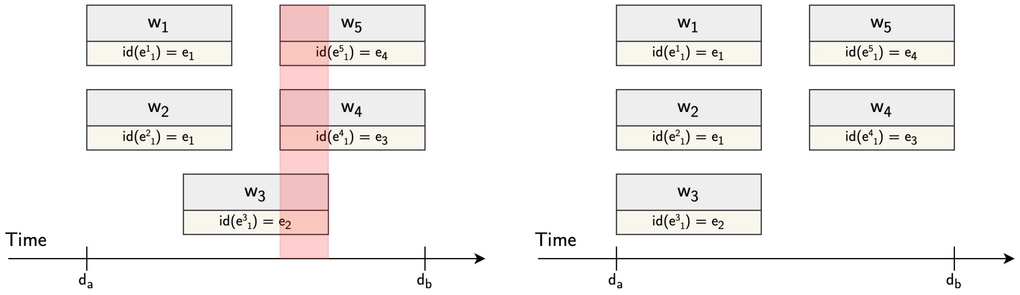
$$\texttt{noOverlap}\Big(\{e_i, e_j\}\Big) \;\; \forall e_i \in S_i \wedge \forall e_j \in S_j \wedge \forall S_i \in \Lambda \wedge \forall S_j \in \Lambda \wedge S_i \neq S_j \wedge id(e_i) \neq id(e_j) \quad (4)$$

This constraint ensures that for each pair of different equipment belonging to different sets, one of them must have its activity ended before the other one starts. A valid and an invalid solution for the configuration $\Lambda = \big\{\{e_1, e_2\}, \{e_3, e_4\}\big\}$ is illustrated in Figures 4a and 4b.

Constraint 4: No Overlap inside a Set of Equipment. Let $S \in 2^E$ be an arbitrary set of equipment. This constraints states that equipment from this set cannot be removed together and is modelled as follows.

$$\texttt{noOverlap}\Big(\{w_i \mid \forall w_i \in S\}\Big) \quad (5)$$

It ensures that for each pair of equipment in set S, one of them must finish before the other one starts.



(a) Example of an invalid solution.

(b) Example of a valid solution.

Figure 4 Illustration of Constraint 3 (no overlap on equipment withdrawals).

2.3 Objective Functions

A solution is currently feasible if all the withdrawal requests have been successfully scheduled while ensuring the satisfaction of the four constraints presented in the previous section. However, the transit-power constraints are not yet taken into account and can break the feasibility of a solution. The challenge is that these constraints do not have a closed-form expression and cannot be integrated inside the model. That being said, as a heuristic rule from field specialists, a solution is more likely to satisfy the transit-power constraints when (1) the withdrawal activities are properly balanced inside the planning horizon, and (2) when the activities related to a same equipment are scheduled together. We propose to integrate these rules inside the model through two objective functions having a lexicographic importance.

Objective 1: Maximizing the Schedule Balance. The goal is to balance the withdrawal requests inside the planning horizon. This is related to the *balance* constraint introduced by Bessiere et al. [4]. Generally speaking, the planning horizon has a length of $d_m - d_0$ days. We split this interval into r sub-periods of p days, i.e. $r = \lceil \frac{d_m - d_0}{p} \rceil$. In our case, $m = 245$ and following the recommendations of field specialists, a value of 50 days (p) has been selected, yielding 5 sub-periods (r). This value has been fixed empirically and validated by planners. It is possible that a request overlaps over several sub-periods. For instance, a request between day 0 and day 60 will be counted in the sub-periods $[0, 50]$ and $[51, 100]$. Let R be the set of sub-periods and Ω be a list storing, for each sub-period $r \in R$, the number of requests withdrawn during the sub-period r. In practice, Ω is computed using the well-known function *count*, which is dedicated to counting the number of variables in a list that has a given value [7]. The objective function is then as follows. It drives the solving process to find a schedule that minimizes the largest difference between the number of activities scheduled across the sub-periods.

$$\texttt{minimize} \left| \max_{r \in R} \left(\Omega_r\right) - \min_{r \in R} \left(\Omega_r\right) \right| \tag{6}$$

Objective 2: Maximizing Same Equipment Withdrawal Overlaps. The goal is to maximize the number of overlapping withdrawals of the same equipment. The rationale is that when the same equipment is involved in different withdrawal requests, it is preferable to withdrawn them simultaneously. Let $D = E_1 \cup E_2 \cup \ldots \cup E_n$ be a set containing the sets of equipment from all the withdrawal requests. The objective function is defined as follows. It is based on the *overlapLength* function that computes the number of overlapping days between two activities [12]. All the overlaps are then summed up and maximized.

$$\texttt{maximize}\left(\sum_{e_1\in D}\sum_{e_2\in D}\left\{\texttt{overlapLength}(e_1,e_2)\;\middle|\; e_1\neq e_2\wedge id(e_1)=id(e_2)\right\}\right) \tag{7}$$

It is noteworthy to highlight that the standard objective of minimizing the *makespan* is not considered. In our application, there are no benefits to finishing the maintenance as soon as possible. Spreading the maintenance schedule in the complete horizon is much more desirable as it allows a better flexibility for the field operators. For instance, it allows the planner to readjust dynamically the maintenance schedule when unpredictable events happen, such as an equipment outage.

3 Solving the Transmission Maintenance Scheduling Problem

So far, the transit-power constraints have not been taken into account. Although such constraints cannot be integrated inside the model, their satisfaction can be easily checked thanks to a simulator that mimics the impact of a maintenance schedule on the real power network. The simulator is based on complex differential power-flow equations and has been developed internally by Hydro-Québec. It simulates power flow thanks to PSS/E software.

We propose to leverage this simulator, as a black-box tool, in order to verify the satisfiability of a schedule. Let γ be the power grid considered, let s be a maintenance schedule obtained as a solution for γ, and let d a specific day on the planning horizon. The simulator consists of two black-box functions: (1) $\psi_1(\gamma,s,d)\rightarrow\mathbb{R}$ which computes the transit-power generated by the solution for a specific day, and (2) $\psi_2(\gamma,s,d)\rightarrow\mathbb{R}$, which computes a lower bound on the transit-power that must be satisfied for the obtained schedule, also for a specific day. A solution s on the power grid γ is feasible if it is always above the threshold during the planning horizon.

$$\psi_1(\gamma,s,d)\geq\psi_2(\gamma,s,d) \qquad \forall d\in[d_0,d_{245}] \tag{8}$$

The idea of the solving process is to generate diverse solutions using the constraint programming model and to filter them using the simulator. Solutions that are compliant with the simulator are feasible and can be used in practice. This process is illustrated in Figure 5.

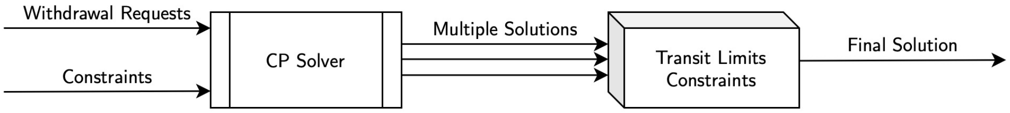
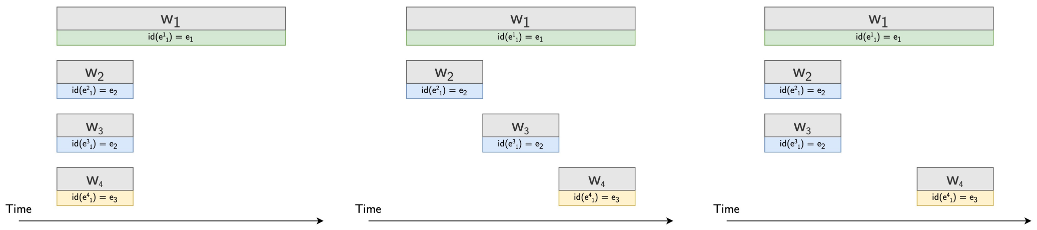


Figure 5 Illustration of the solving pipeline.

One challenge is to generate solutions that are diverse in order to maximize the chance of having at least one schedule accepted by the simulator. We resort to three mechanisms to ensure the diversity of solutions: (1) integrating domain knowledge as objective function, (2) adding constraints dynamically when a solution has been found, and (3) directing the search by a *multi-point* strategy. This section presents how these ideas are integrated into the solving process. We assume that a solution satisfying all the formalized constraints is obtained.

Mechanism 1: Injecting Domain Knowledge as Objective Function. Field specialists have heuristic rules for creating scheduling satisfying the transit-power constraint. Those have been formalized in Equations (6) and (7). The idea is to integrate such rules as a two-steps objective function. The problem is first solved by maximizing the balance of the schedule. From the solution obtained, a second solving process is executed in order to maximize the withdrawal overlaps of the same equipment. The value of the first objective is allowed to decrease up to a given threshold ϵ. This process is illustrated in Figure 6.



(a) Initial solution. **(b)** Adding Equation (6). **(c)** Adding Equation (7).

Figure 6 Illustration of the two-steps objective function.

Mechanism 2: Adding new Constraints Dynamically. Two solutions successively generated by a CP solving process are likely to share many characteristics. We improve the diversity of the solutions obtained by adding a new constraint each time a new feasible solution has been found. Let $w^\star$ refer to the value of the variable w in the last solution found. The new constraint ensures that the next generated solution must have at least l withdrawal requests moved from at least d days of the previous solution. The start time is used as reference. We empirically set $d = 1$ and $l = 1$, which already yielded diverse enough solutions.

$$\sum_{w \in W} \Big(\big|s(w) - s(w^\star)\big| \geq d \Big) \geq l \tag{9}$$

Mechanism 3: Directing the Search by a Multi-Point Strategy. Finally, a *multi-point* strategy with the default search heuristics proposed by CP Optimizer is used for driving the solving process [13]. This strategy creates an initial set of solutions and combines them together in order to produce improved solutions. It has the benefit of providing a more diversified solution than a standard depth-first search. However, it acts as an incomplete search procedure and cannot prove the optimality of a solution. That being said, this limitation is not restrictive in our case, as we only need to find feasible solutions. The objective functions are only used as heuristics.

4 Experimental results

The goal of the experiments is to show the adequacy of the approach to generate schedules that can be used in practice for the geographic area considered. To do so, the maintenance planning designed by field specialists for the year 2020 is considered and compared with the planning obtained by our approach. In total, 359 withdrawal requests were considered and 271 electrical equipment items are involved. Each withdrawal request involves at most 8 items, yielding a maximum of 2872 activities. The maintenance schedule has an impact on five strategic points of the power grid infrastructure, also referred to as *interface*. Each interface has its own transit-power constraints. It is interesting to mention that due to Québec geography, the network is not intensively meshed. The production infrastructures are all located in the North while most of the consumption is made in the South. The experiments are executed on an Intel i5-8520 processor (1.6 GHz) with 16 GB of RAM. The model is implemented in C++ using CP Optimizer 20.1. In total, the solving process took 1000 seconds. Roughly 85% of the execution time was dedicated to finding solutions and 15% was used to verify the transit-power constraints with the simulator.

4.1 Visualization of a Feasible Schedule

A visualization of the maintenance schedule obtained for the first interface is proposed in Figure 7. The x-axis represents the planning horizon from day 0 to day 245. As commonly done for scheduling problems, each gray bar represents the execution of the withdrawal request associated with each equipment item. For practical reasons, only equipment that affects the transit-power limit is displayed. For reasons of confidentiality, equipment names are omitted. For instance, they can correspond to power lines, capacitors, transistors, etc. The red curve indicates the transit-power generated by the maintenance schedule (output of ψ_1 function) while the blue curve indicates the transit-power limit (output of ψ_2 function). Consequently, a schedule satisfies the transit-power constraints if and only if the red curve is always above the blue curve, which is the case for this interface. A similar result for three other interfaces is presented in Appendix A. These results demonstrate that our approach is sufficient to satisfy the transit-power constraints on these interfaces.

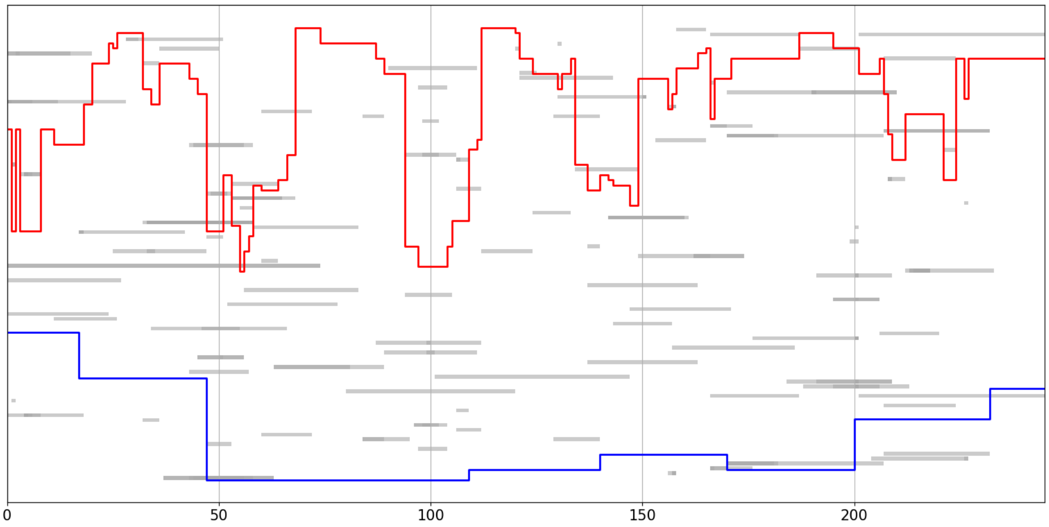


Figure 7 Visualization of a feasible maintenance schedule (first interface).

4.2 Visualization of an Unfeasible Schedule

Among the five interfaces considered in our power grid, four of them satisfy the transit-power constraints. A visualization of the maintenance schedule obtained for the last one is proposed in Figure 8. Interestingly, the transit-power constraints are violated only a few times (e.g., around day 50 and after day 200). Generally speaking, we also observe that the safety margin between the two curves is tinier than the one presented in Figure 8. This case was discussed with field specialists. They confirmed that ensuring the satisfaction of transit-power constraints at this interface is challenging. In practice, they regularly have to accommodate with a schedule that does not respect the constraints at this interface. Addressing this challenging interface is part of future work.

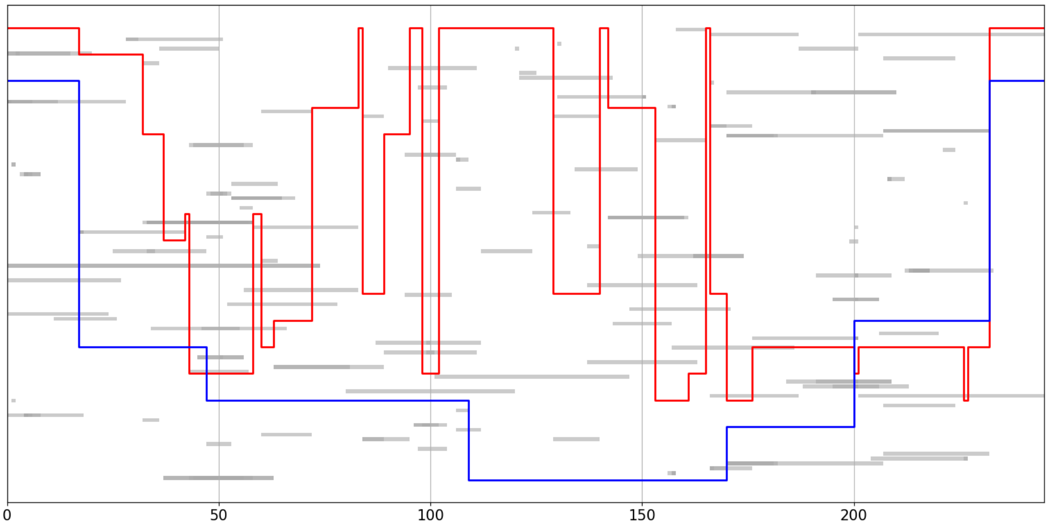


Figure 8 Visualization of an unfeasible maintenance schedule (fifth interface).

4.3 Evaluating the Similarity with the Historical Schedule

The goal of this analysis is to highlight the similarities and the differences between the solutions allowed by our constraints and the one designed by field specialists that was used in 2020. Then, we will be able to assess if the decisions made are consistent with the ones historically done, and otherwise discover potential sources of discrepancies. We propose a visualization of this information using a confusion matrix. Each request can be either *accepted* or *refused* by the field operator. By replaying the decision of the operator on historical requests of 2020, Table 2 shows the proportion of withdrawal requests that have the same or different status with our constraints and the historical model.

Table 2 Proportion of accepted or refused requests between both schedules in 2020.

		Schedule allowed by the constraints	
		Approved Requests	*Refused requests*
Historical schedule	*Approved requests*	**61.5%**	12.5%
	Refused requests	17%	**9%**

Interestingly, we notice that 70.5% (61.5% + 9%) of the requests have the same status. It means that the decision regarding these requests is identical as what has been done in 2020. In addition, 12.5% of the requests have been refused by our model while being accepted in 2020. This corresponds to situations where the field operator has accepted a request that will cause a constraint violation. This has been done either intentionally (e.g., constraint assessed to be too restrictive) or unintentionally given the complexity of this task. Finally, 17% of the requests have been approved by our model but were refused in 2020. This may be an indication that some constraints used in practice by field specialists are missing in the model. These can be either other technical constraints or non-related constraints such as budget or workforce constraints.

4.4 Evaluating the Schedule Balance

This experiment assesses the schedule balance obtained with our model. To do so, we count the number of requests per period of 31 days (one month) and analyze if the schedule provided by our model has a similar balance as the one designed in 2020. This is summarized in Table 3 for each period of 31 days since March 15. The spread value indicates the difference between the maximum and the minimum values inside the planning horizon. The lower the value is, the more balanced is the schedule. We can observe that our generated schedule is slightly more balanced.

Table 3 Comparison of the maintenance schedule maintenance.

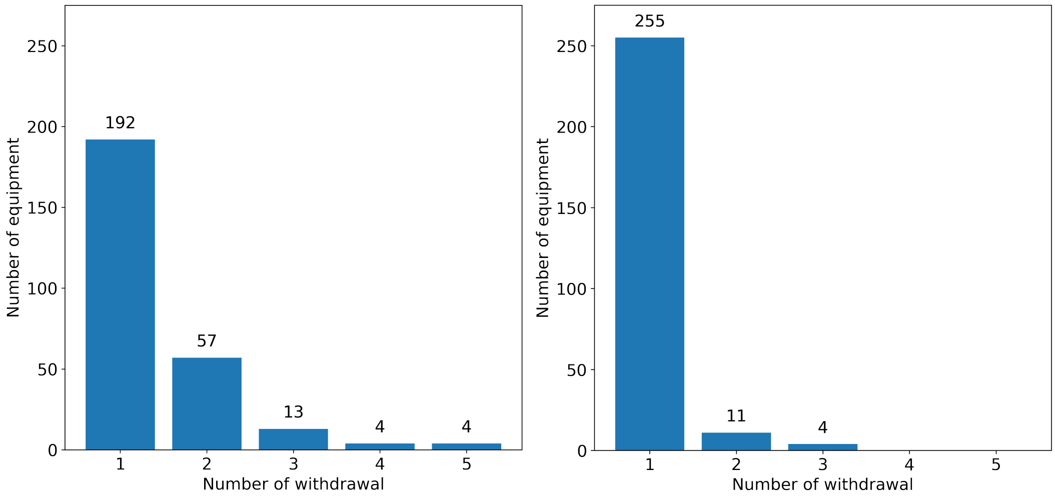
	Planning horizon (split into 8 months)								**Spread value**
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	
Historical schedule	45	57	72	55	36	42	62	21	51
Our schedule	58	61	32	56	63	40	63	19	42

4.5 Evaluating the Overlaps between Equipment Withdrawals

Most of the time, the same equipment unit is involved in different withdrawal requests. The second objective function is dedicated to maximizing the overlaps between these requests. To evaluate this objective, we count the number of times an equipment has been withdrawn from the network. Figure 9 shows the distribution of the number of withdrawals required per equipment for the historical schedule (left) and for our model (right). We can observe that the model is able to remove each equipment less often which is what is intended by this objective function.

5 Conclusion and perspective

Modern electrical power utilities must maintain their electrical equipment and replace them as they reach the end of their useful life. Generating an annual maintenance plan for the electric power transportation equipment while maintaining the stability and efficiency of the network is still a challenge at present. Based on this context, we proposed a constraint programming approach for solving a realistic transportation maintenance scheduling problem. The focus was to design an approach that could handle two types of constraints: (1) constraints that could be naturally formalized inside a constraint programming model, and (2) constraints that were too complex to be implemented but could be verified using a black-box tool. The objective was to generate schedules similar to what is currently being done by field specialists in order to simulate maintenance programs for the next 10 years. Experimental results show that the model captures most of the unformalized constraints and is able to generate realistic schedules. It is important to highlight that two kinds of constraints are not yet considered: budget constraints, and specialized crew availability constraints. In future work, we shall attempt to integrate such constraints into the model. Another interesting aspect is the assessment of increased risk of failures during maintenance. For such a criticality analysis,



(a) Historical schedule.

(b) Our schedule.

Figure 9 Comparison of the number of overlaps between equipment withdrawal.

other modeling and solving tools (e.g., stochastic programming [6]) may be considered. Finally, another idea is to leverage methods from *constraint acquisition* in order to learn new constraints from the interaction with the black-box simulator [5].

References

1 ILOG CPLEX – Cumul functions in CP Optimizer. `https://www.ibm.com/docs/en/icos/20.1.0?topic=concepts-cumul-functions-in-cp-optimizer`. Accessed: 2022-02-06.

2 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.

3 David L Alvarez, Diego F Rodriguez, Alben Cardenas, F Faria da Silva, Claus Leth Bak, Rodolfo García, and Sergio Rivera. Optimal decision making in electrical systems using an asset risk management framework. *Energies*, 14(16):4987, 2021.

4 Christian Bessiere, Emmanuel Hebrard, George Katsirelos, Zeynep Kiziltan, Émilie Picard-Cantin, Claude-Guy Quimper, and Toby Walsh. The balance constraint family. In *International Conference on Principles and Practice of Constraint Programming*, pages 174–189. Springer, 2014.

5 Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artificial Intelligence*, 244:315–342, 2017.

6 John R Birge and Francois Louveaux. *Introduction to stochastic programming*. Springer Science & Business Media, 2011.

7 Mats Carlsson, Johan Widen, Johan Andersson, Stefan Andersson, Kent Boortz, Hans Nilsson, and Thomas Sjöland. *SICStus Prolog user's manual*, volume 3. Swedish Institute of Computer Science Kista, Sweden, 1988.

8 Aurélien Froger, Michel Gendreau, Jorge E. Mendoza, Eric Pinson, and Louis-Martin Rousseau. Maintenance scheduling in the electricity industry: a literature review. *European Journal of Operational Research*, 251(3):695–706, June 2016. `doi:10.1016/j.ejor.2015.08.045`.

9 Mohamed Gaha, Bilal Chabane, Dragan Komljenovic, Alain Côté, Claude Hébert, Olivier Blancke, Atieh Delavari, and Georges Abdul-Nour. Global methodology for electrical utilities maintenance assessment based on risk-informed decision making. *Sustainability*, 13(16), 2021. `doi:10.3390/su13169091`.

10 Steven Gay, Renaud Hartert, and Pierre Schaus. Simple and scalable time-table filtering for the cumulative constraint. In *International conference on principles and practice of constraint programming*, pages 149–157. Springer, 2015.

11 Philippe Laborie and Jerome Rogerie. Reasoning with conditional time-intervals. In *FLAIRS conference*, pages 555–560, 2008.

12 Philippe Laborie and Jérôme Rogerie. Temporal linear relaxation in IBM ILOG CP Optimizer. *Journal of Scheduling*, 19(4):391–400, 2016.

13 Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP Optimizer for scheduling. *Constraints*, 23(2):210–250, 2018.

14 Jingcheng Mei, Guojiang Zhang, Donglian Qi, and Jianliang Zhang. Accelerated solution of the transmission maintenance schedule problem: A bayesian optimization approach. *Global Energy Interconnection*, 4(5):493–500, 2021. doi:10.1016/j.gloei.2021.11.001.

15 Hrvoje Pandzic, Antonio J. Conejo, Igor Kuzle, and Eduardo Caro. Yearly maintenance scheduling of transmission lines within a market environment. *IEEE Transactions on Power Systems*, 27(1):407–415, 2012. doi:10.1109/TPWRS.2011.2159743.

16 Ragheb Rahmaniani, Teodor Gabriel Crainic, Michel Gendreau, and Walter Rei. The benders decomposition algorithm: A literature review. *European Journal of Operational Research*, 259(3):801–817, 2017.

17 Mariana Rocha, Miguel F. Anjos, and Michel Gendreau. Optimal planning of preventive maintenance tasks on electric power transmission systems. In *31st European Safety and Reliability Conference (ESREL)*, pages 1025–1025, January 2021. doi:10.3850/978-981-18-2016-8_721-cd.

18 Hiroki Shigetsugu, Stewart Whyte, Paul Penserini, Boudewijn Neijens, David Neilson, and Jos Wetzer. Standardization activities of management of network assets in power systems in IEC, April 2022.

19 Gilles Trudel, Jean-Pierre Gingras, and Jean-Robert Pierre. Designing a reliable power system: Hydro-quebec's integrated approach. *Proceedings of the IEEE*, 93(5):907–917, 2005.

A Appendix: Solutions at Other Interfaces

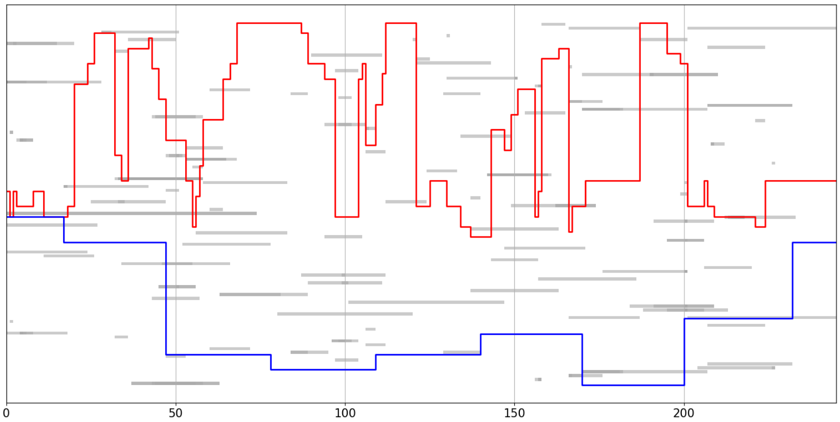


Figure 10 Visualization of a feasible maintenance schedule (second interface).

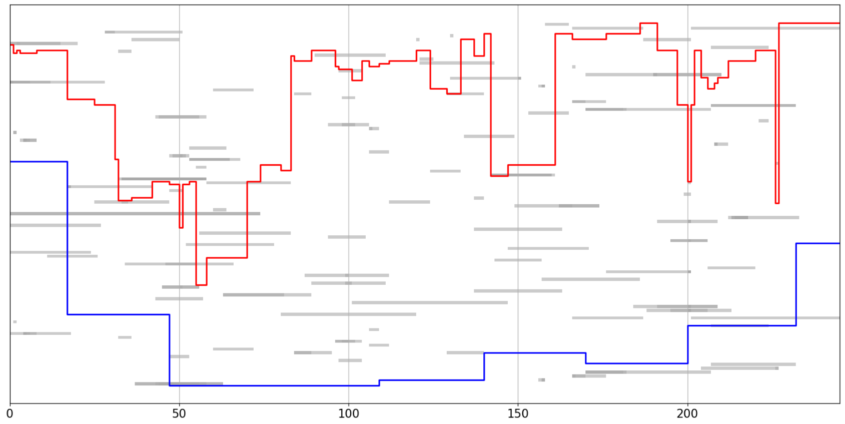


Figure 11 Visualization of a feasible maintenance schedule (third interface).

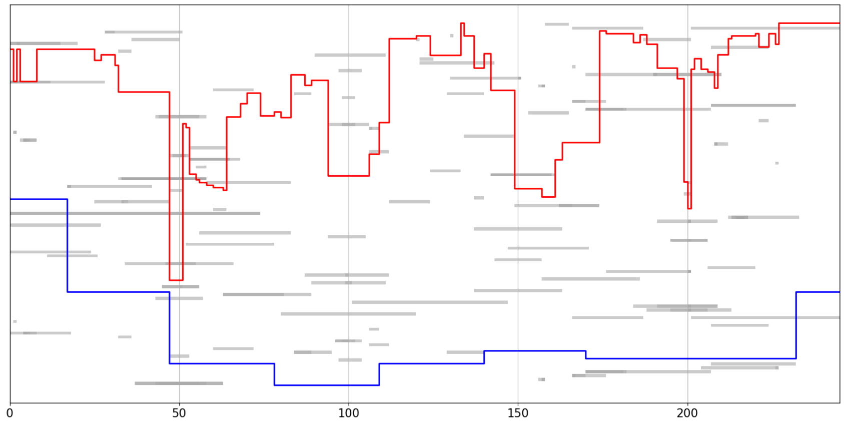


Figure 12 Visualization of a feasible maintenance schedule (fourth interface).

Peel-And-Bound: Generating Stronger Relaxed Bounds with Multivalued Decision Diagrams

Isaac Rudich ✉
Mathematics and Industrial Engineering Department, Polytechnique Montréal, Canada

Quentin Cappart ✉
Computer Engineering and Software Engineering Department, Polytechnique Montréal, Canada

Louis-Martin Rousseau ✉ 🏠
Mathematics and Industrial Engineering Department, Polytechnique Montréal, Canada

Abstract

Decision diagrams are an increasingly important tool in cutting-edge solvers for discrete optimization. However, the field of decision diagrams is relatively new, and is still incorporating the library of techniques that conventional solvers have had decades to build. We drew inspiration from the *warm-start* technique used in conventional solvers to address one of the major challenges faced by decision diagram based methods. Decision diagrams become more useful the wider they are allowed to be, but also become more costly to generate, especially with large numbers of variables. We present a method of *peeling* off a sub-graph of previously constructed diagrams and using it as the initial diagram for subsequent iterations that we call *peel-and-bound*. We test the method on the *sequence ordering problem*, and our results indicate that our *peel-and-bound* scheme generates stronger bounds than a branch-and-bound scheme using the same propagators, and at significantly less computational cost.

2012 ACM Subject Classification Applied computing → Operations research

Keywords and phrases decision diagrams, discrete optimization, branch-and-bound, sequencing, constraint programming

Digital Object Identifier 10.4230/LIPIcs.CP.2022.35

Supplementary Material *Other (Source Code for Experiments)*:
`https://github.com/IsaacRudich/PnB_SOP`

1 Introduction

Multivalued decision diagrams (MDDs) are a useful graphical tool for compactly storing the solution space of discrete optimization problems. In the last few years, a staggering number of new applications for MDDs have been proposed [8], such as representing global constraints [26, 27, 28], handling stochastic variables [20, 19], and performing post-optimality analysis [25]. MDDs are particularly useful for generating strong dual bounds [6, 7, 14, 21], especially on optimization problems where linear relaxations perform poorly. There is a subset of MDD research that uses a highly paralellizable *branch-and-bound* algorithm based on decision diagrams [5, 10, 11, 22] to maximize the utility of using MDD based relaxations. This paper furthers the work on the decision diagram based branch-and-bound by introducing a method, referred to as *peel-and-bound*, of reusing the graphs generated at each iteration of the algorithm. Specifically, the contributions are as follows: (1) we present the *peel-and-bound* algorithm, (2) we identify several heuristic decisions that can be used to adjust *peel-and-bound*, and discuss their implications, (3) we show that *peel-and-bound* outperforms *branch-and-bound* on the *sequence ordering problem* (SOP), and (4) we provide insight into how the algorithm can be applied to other problems.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 35; pp. 35:1–35:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The paper is structured as follows. The next section provides the necessary technical background information and notation, as well as implementation details for the decision diagram relaxations used in our experiments. In Section 3 we introduce the core contribution, namely the *peel-and-bound* procedure. The algorithm is presented, and its limitations are discussed. Computational experiments are proposed and discussed in Section 4.

2 Technical Background

The idea of using multivalued decision diagrams (MDDs) to generate relaxed bounds for optimization problems was introduced by Andersen et al. (2010) [1]. This has been generalized by Hadzic et al. (2008) [12] and Hoda et al. (2010) [13]. Following those papers, Bergman et al. [5, 4] demonstrated the potential for a decision diagram based branch-and-bound solver to be effective, and provided an efficient parallelization scheme. Gillard et al. (2021) [10] further improved the decision diagram based branch-and-bound solver by adding pruning techniques that can be used while the decision diagrams are being constructed, as well as to remove nodes from the branch-and-bound queue.

This paper presents a new peel-and-bound scheme for combining restricted and relaxed decision diagrams to find exact solutions. This section provides the required technical background on how decision diagrams can be used to model sequencing problems, and how to construct restricted/relaxed diagrams. It also introduces the notations used in this paper, and details the existing algorithms considered in our experiments.

2.1 Decision Diagrams (DDs)

Let $\mathcal{P}$ be an instance of a discrete minimization problem with n variables $\{x_1, ..., x_n\}$, let $Sol(\mathcal{P})$ be the set of feasible solutions to $\mathcal{P}$, let $z^*(\mathcal{P})$ be an optimal solution to $\mathcal{P}$, and let $D(x_i)$ be the domain of variable $x_i, i \in \{1, ..., n\}$. Let $\mathcal{M}$ be a multivalued decision diagram that contains potential solutions to $\mathcal{P}$. $\mathcal{M}$ is a directed acyclic graph divided into $n+1$ layers; let ℓ_u be the index of the layer containing node $u, u \in \mathcal{M}$, and let L_i be the set containing the nodes on layer i. Layer 1 contains only a root node r (with no *in* arcs), and layer $n+1$ contains just a terminal node t (with no *out* arcs). Each arc $a_{uv} \in \mathcal{M}$ goes from a node u on layer $\ell_u \in \{1, ..., n\}$ to a node v on layer ℓ_{u+1} ($\ell_{u+1} = \ell_v$). Each arc a_{uv} has a label representing the assignment of variable x_{ℓ_u} to $l \in D(x_{\ell_u})$. An arc a_{uv} with label l ($a_{uv} \to l$) also has a value $v(a_{uv})$ equal to the value of being at node u and assigning x_{ℓ_u} to l ($x_{\ell_u} = l$). For simplicity, we sometime refer to $v(a_{uv})$ as $v(a)$. Thus, each path from r to t represents the assignment of the n variables to values, and a potential solution to $\mathcal{P}$.

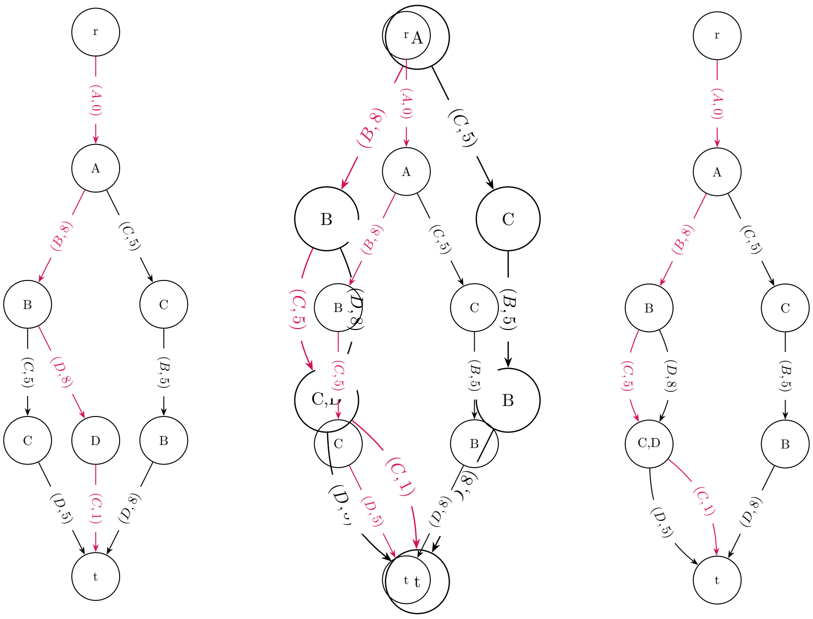
Let $Sol(\mathcal{M})$ be the set of all paths in $\mathcal{M}$ from r to t, and let $T^*(u)$ be the value of the shortest path from r to a node u. If $Sol(\mathcal{M}) = Sol(\mathcal{P})$, then $\mathcal{M}$ perfectly represents the solution space of $\mathcal{P}$, and we call $\mathcal{M}$ *exact*. If $\mathcal{M}$ is exact, then the value of the shortest path through the diagram is $z^*(\mathcal{P})$ (an optimal solution to $\mathcal{P}$). Let the shortest path through $\mathcal{M}$ be $z^*(\mathcal{M})$. If $Sol(\mathcal{M}) \subseteq Sol(\mathcal{P})$, then $\mathcal{M}$ represents only feasible solutions to $\mathcal{P}$, but does not necessarily represent all feasible solutions to $\mathcal{P}$. In this case, we call $\mathcal{M}$ *restricted*, and use the notation $\overline{\mathcal{M}}$ to mean that $\mathcal{M}$ is restricted. The shortest path through $\overline{\mathcal{M}}$ is not guaranteed to be optimal, but it is guaranteed to be feasible. If $Sol(\mathcal{P}) \subseteq Sol(\mathcal{M})$, then $\mathcal{M}$ represents all of the feasible solutions to $\mathcal{P}$, but potentially represents infeasible solutions as well. In this case, we call $\mathcal{M}$ *relaxed*, and use the notation $\underline{\mathcal{M}}$ to mean that $\mathcal{M}$ is relaxed. The shortest path through $\underline{\mathcal{M}}$ is guaranteed to be at least as good as $z^*(\mathcal{P})$, but is not guaranteed to be feasible.

Constructing an exact decision diagram for $\mathcal{P}$ is often intractable for large values of n. Observe that having an exact decision diagram means that the solution to $\mathcal{P}$ can be read in polynomial time by recursively calculating the shortest path through $\mathcal{M}$, so creating an exact decision diagram for NP-hard problems, such as for the *travelling salesperson problem* (TSP), is NP-hard as well [3]. The focus of most research that uses decision diagrams for optimization is on the construction of $\overline{\mathcal{M}}$ and/or $\underline{\mathcal{M}}$. Let $w = w(\mathcal{M})$ be the width of the largest layer of $\mathcal{M}$. The creation of an exact decision diagram potentially leads to w being an exponential function of n, but when creating $\overline{\mathcal{M}}$ and/or $\underline{\mathcal{M}}$, w can be constrained to be any natural number, limiting the number of operations construction will take. Let w_m be the largest width allowed during construction. As w_m approaches the width necessary to create an exact decision diagram, $z^*(\overline{\mathcal{M}})$ and $z^*(\underline{\mathcal{M}})$ approach $z^*(\mathcal{P})$, but the number of operations necessary to construct the diagram also increases.

Table 1 Example of a SOP instance: transition costs $c_{row,col}$, with X indicating infeasible edges.

To From	A	B	C	D
A	X	8	5	0
B	X	X	5	8
C	X	5	X	5
D	X	X	1	X

Figure 1: Sequence Ordering Problem Instance



Exact Diagram
$z^* = [A, B, D, C]$
$T^* = 17$

Restricted Diagram
$z^* = [A, B, C, D]$
$T^* = 18$

Relaxed Diagram
$z^* = [A, B, C, C]$
$T^* = 14$

Figure 1 MDD 16 Representation for the SOP 17 instance presented in Table 18 1. Each arc a has the format: $(l, v(a))$. The red path in each diagram 19 indicates the shortest path from r to t.

Table 1 gives an instance of *sequence ordering problem* (SOP), and Figure 1 contains simple examples of exact, restricted, and relaxed decision diagrams for that instance where $w_m = 2$ for $\overline{\mathcal{M}}$ and $\underline{\mathcal{M}}$. The SOP requires finding the minimum-cost sequence of n elements that includes each element exactly once, subject to transition costs c_{ij} of following x_i with

x_j, and subject to precedence constraints requiring that certain elements precede others in the sequence. In other words, the SOP is an asymmetric TSP with precedence constraints. The label of each node matches the union of the labels of the incoming arcs. Each arc a_{uv} is labeled in the format $(l, val(a_{uv}))$, representing the assignment of x_{ℓ_u} to l, and $val(a)$ represents the cost of the shortest path from the label of u to l. In other words, an arc with label l leaving layer i, represents the assignment of l to the i^{th} position of the sequence. The red path in each diagram indicates the shortest path through the diagram, and T^* indicates the cost of the shortest path through the diagram.

2.2 Restricted Decision Diagrams

Constructing $\overline{\mathcal{M}}$ for a given width w_m is a straightforward process that can be thought of as a generalized greedy algorithm. Beginning with the root node r, an arc is generated for every element in the domain of r, and a node is generated at the end of each arc in the second layer. The process is repeated for each layer, except layer n where all outgoing arcs point to the terminal, unless $w(\overline{\mathcal{M}})$ exceeds w_m. Then the least promising node is removed from the offending layer until $w(\overline{\mathcal{M}})$ is equal to w_m. The definition of *least promising* is a heuristic decision. For the purposes of this paper, the least promising node is the node u such that the shortest path from r to u is longer than the shortest path from r to any other node $v \neq u$ in layer ℓ_u.

It is of note that another method of reducing the width of $\overline{\mathcal{M}}$ is merging equivalent nodes. In the SOP, two nodes can be considered equivalent if they have the same state (last element in the sequence), and all incoming paths have visited the same set of elements. For example, a node with exactly one incoming path $[A, B, C]$ could be merged with a node in the same layer with exactly one incoming path $[B, A, C]$. In many MDD applications this is a valuable insight, and it helps motivate the algorithm for constructing $\underline{\mathcal{M}}$. However, for the SOP, we observed that the work of finding equivalent nodes in $\overline{\mathcal{M}}$ often outweighed the benefit of being able to merge nodes.

2.3 Relaxed Decision Diagrams

There are many methods of constructing relaxed decision diagrams, and many heuristic decisions that must be made when doing so. In this paper, we focus on the method described by Cire and van Hoeve (2013) [9] for sequencing problems. As opposed to the top-down construction described in Section 2.2, here $\underline{\mathcal{M}}$ will be constructed by separation. Constructing DDs by separation uses $\underline{\mathcal{M}}$ as a domain store over which constraints can be propagated. This method starts with a weak relaxation, and then strengthens it by splitting nodes until each layer is either exact, or has a width equal to w_m. The algorithm begins from a 1-width MDD with an arc from the node on layer ℓ_i to the node on layer ℓ_{i+1} for each element that can be placed at position ℓ_i in the sequence. Thus, even though each layer only has one node, there can be several arcs between layers (see the relaxed diagram in Figure 1). Then a node u is selected and split to strengthen the relaxation. The process of splitting u involves creating two new nodes u'_1 and u'_2, and then distributing the *in* arcs of u between u'_1 and u'_2. Then for each *out* arc a_{uv} from u, arcs $a_{u'_1v}$ and $a_{u'_2v}$ are added such that a_{uv}, $a_{u'_1v}$ and $a_{u'_2v}$ all have the same label. Finally u'_1 and u'_2 are filtered to remove infeasible and sub-optimal arcs. A collection of filtering rules are used to check each arc. As an example, given a feasible solution to $\mathcal{P}$ with objective value z_{opt}, an arc a can be removed if all paths containing a have an objective value greater than z_{opt}. The full process of identifying which arcs can be removed is detailed in Cire and van Hoeve (2013) [9], and is not replicated here.

The following notation and definitions are critical to understanding these algorithms. Let $All^{\downarrow}_{u}$ be the set of arc labels that appear in every path from r to u. Let $Some^{\downarrow}_{u}$ be the set of arc labels that appear in at least one path from r to u. Let $All^{\uparrow}_{u}$ and $Some^{\uparrow}_{u}$ be defined as above, except that they refer to paths from u to t. Let $\mathcal{J}$ be the set of all possible arc labels. For the SOP, we define an *exact* node u as a node where $Some^{\downarrow}_{u} = All^{\downarrow}_{u}$ and all arcs ending at u originate from exact nodes. Intuitively, a node u is exact if all paths to u contain the same set of labels, and all parents of u are exact. Algorithm 1 formalizes the process of strengthening $\underline{\mathcal{M}}$.

Algorithm 1 Refining Decision Diagrams for Sequencing [9].

```
Let $\underline{\mathcal{M}}$ be an MDD such that $Sol(\underline{\mathcal{M}}) \supseteq Sol(\mathcal{P})$
for layer $L_j \in \underline{\mathcal{M}}$ from $j = 1$ to $j = n$ do
    while $|L_j| < w_m$ and $\exists$ some node $y \in L_j$ such that $y$ is not exact do
        $\mathcal{J} \leftarrow$ getAssignmentOrdering($\mathcal{P}$)
        The getAssignmentOrdering() function returns a heuristically defined ordering of the
         values that can be assigned to decision variables
        for $\phi \in \mathcal{J}$ while $|L_j| < w_m$ do
            $S \leftarrow$ selectNodes($L_j$,$\phi$)
            The selectNodes() function returns the set of nodes $u \in L_j$ such that
             $\phi \in Some^{\downarrow}_{u} \backslash All^{\downarrow}_{u}$
            for $u \in S$ while $|L_j| < w_m$ do
                Create two new nodes $u'_1, u'_2$
                $L_j \leftarrow (L_j \cup \{u'_1, u'_2\})$
                foreach arc $a_{vu}$ do
                    if $\phi \in (All^{\downarrow}_{v} \cup$ the label of $a)$ then
                        Redirect $a$ such that $a_{vu'_1}$
                    else
                        Redirect $a$ such that $a_{vu'_2}$
                    end
                end
                foreach arc $a_{uv}$ do
                    Create arcs $a_{u'_1 v}$ and $a_{u'_2 v}$ such that
                     $label(a_{uv}) = label(a_{u'_1 v}) = label(a_{u'_2 v})$
                    filter($a_{u'_1 v}$), filter($a_{u'_2 v}$)
                    filter(a) runs a list of quick checks to see if an arc can be removed
                end
                $L_j \leftarrow (L_j \backslash u)$
            end
        end
    end
end
return $\underline{\mathcal{M}}$
```

Deciding which nodes to split, and how to split them, are heuristic decisions with a significant impact on the bound that can be achieved without exceeding w_m [3]. The algorithm discussed here selects nodes that can be split into equivalency classes, such that every path to the new node contains a certain label. Deciding which equivalency classes to produce first is another heuristic decision. The details of ordering the importance of the labels are specific to the problem being solved, and are not discussed here. However, it is important to note that the ordering for this implementation is static, and does not change between iterations.

2.4 Branch-and-Bound with Decision Diagrams

In a typical branch-and-bound algorithm, the branching takes place by splitting on the domain of the variables. With decision diagrams, the branching takes place on the nodes themselves by selecting a set of exact nodes to represent the problem. The solver outlined by Bergman et al. [4] defines an exact node as a node u for which every path from r to u ends in an equivalent state. As mentioned above, we can be more specific when applying this to sequencing problems, and define an exact node u as a node where $Some_u^{\downarrow} = All_u^{\downarrow}$ and all arcs ending at u originate from exact nodes. An *exact cutset* is defined as a set of exact nodes that contain every path from r to t. Let $\underline{\mathcal{M}}(u)$ be a relaxed decision diagram with root u, and let $\overline{\mathcal{M}}(u)$ be a restricted decision diagram with root u. The branch-and-bound algorithm for MDDs proceeds by selecting an exact cutset of $\underline{\mathcal{M}}$, and using each node u in the cutset as the root for a new restricted decision diagram $\overline{\mathcal{M}}(u)$ and relaxed decision diagram $\underline{\mathcal{M}}(u)$. A node can be removed from the queue if the relaxation of that node is not better than the best known solution to $\mathcal{P}$, otherwise the exact cutset of the new node is added to the queue, and the process repeats until the queue is empty. This is detailed by Algorithm 2.

Gillard et al. [10] expanded on Algorithm 2 by incorporating a local search. A heuristic is used to quickly calculate a *rough relaxed bound*[1] at each node, and if the length of the shortest path to that node plus the rough relaxed bound is worse than the best known solution, the node can be removed. More formally, let $rrb(u)$ be a rough relaxed bound on $\mathcal{P}$ starting from node u, and let z_{opt} be the value of best known solution so far. If $T^*(u) + rrb(u) > z_{opt}$, the node can be removed. They also provide evidence that if $rrb(u)$ is inexpensive to compute, it can be used to filter nodes in $\overline{\mathcal{M}}$ and $\underline{\mathcal{M}}$. The method of using a rough relaxed bound to trim nodes is used in this paper, but the details are problem specific and are discussed in a later section.

3 Peel-and-Bound Algorithm

The motivation for peel-and-bound stems from an observation about Algorithm 1. When implemented in a branch-and-bound structure, a large portion of the work done while generating each $\underline{\mathcal{M}}$ is repeated at every iteration. Creating the relaxation for some exact node u in the queue requires creating a 1-width decision diagram, iterating over each layer from the top down, and splitting nodes in a predetermined order. The static order of node splits means that for each node y such that $\ell_y > \ell_u$, the first equivalency class created when splitting y is the same in $\underline{\mathcal{M}}(r)$ and $\underline{\mathcal{M}}(u)$. The existing arcs for both diagrams will be sorted in the same way, and the only difference is the possibility of filtering arcs in $\underline{\mathcal{M}}(u)$ that could not be filtered in $\underline{\mathcal{M}}(r)$ due to the added constraint that all paths must pass through u. The extra filtered arcs are the reason that $\underline{\mathcal{M}}(u)$ may produce a stronger bound than $\underline{\mathcal{M}}(r)$. However, because equivalency classes are chosen in the same order each time, many arcs that were filtered while constructing $\underline{\mathcal{M}}(r)$ will also be filtered again while constructing $\underline{\mathcal{M}}(u)$.

[1] Gillard et al. [10] call the value *rough upper bound*, but since we are testing a minimization problem in this paper, we use the term *rough relaxed bound* instead.

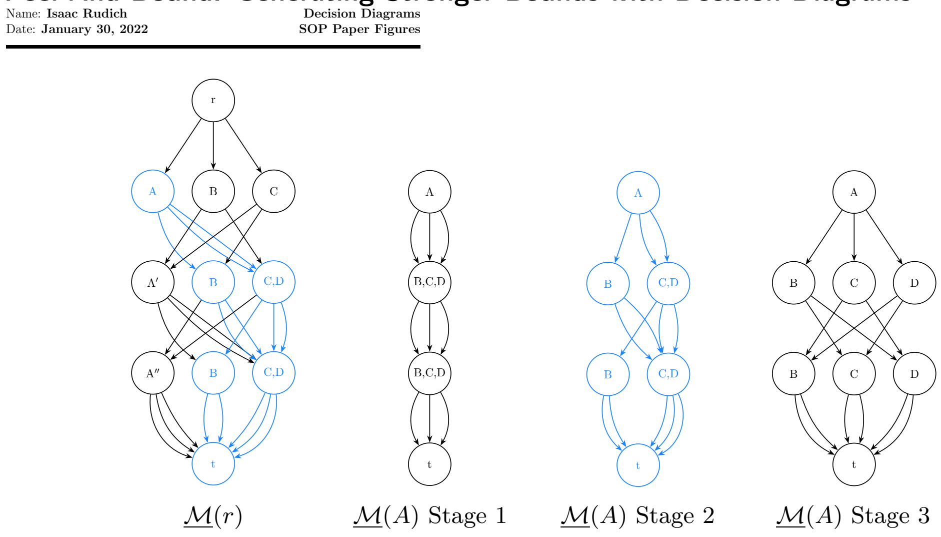
Algorithm 2 Decision Diagram based Branch-and-Bound (BnB) [4].

```
Let $\underline{\mathcal{M}_{uu'}}$ be a partial diagram with root $u$ and terminal $u'$
Let $v^*(u)$ be the lower bound of $\mathcal{P}$ resulting from starting at node $u$
Let $z_{opt}$ be the value of the best known solution
$Q = \{r\}$
$v^*(r) \leftarrow 0$
$z_{opt} \leftarrow \infty$
while $Q \neq \emptyset$ do
    $u \leftarrow$selectNode$(Q)$, $Q \leftarrow Q \backslash \{u\}$
    $\overline{\mathcal{M}} \leftarrow \overline{\mathcal{M}}(u)$
    if $v^*(\overline{\mathcal{M}}) < z_{opt}$ then
        $z_{opt} \leftarrow v^*(\overline{\mathcal{M}})$
    end
    if $\overline{\mathcal{M}}$ is not exact then
        $\underline{\mathcal{M}} \leftarrow \underline{\mathcal{M}}(u)$
        if $v^*(\underline{\mathcal{M}}) < z_{opt}$ then
            $S \leftarrow$ exactCutset$(\underline{\mathcal{M}})$
            foreach $u' \in S$ do
                let $v^*(u') = v^*(u) + v^*(\underline{\mathcal{M}_{uu'}})$
                $Q \leftarrow Q \cup u'$
            end
        end
    end
end
return $z_{opt}$
```

There is a sub-graph of $\underline{\mathcal{M}}(r)$, induced by node u, that contains all of the paths that will be encoded in $\underline{\mathcal{M}}(u)$, but does not contain the arcs that are filtered from both diagrams during construction. Thus, less work needs to be performed at each iteration of branch-and-bound by starting from that sub-graph instead of a 1-width diagram. If the split order is static, the same diagram is generated starting from either the 1-width diagram, or the sub-graph induced by u. If the split order changes between branch-and-bound iterations, the sub-graph induced by u is still a valid relaxation, but the generated diagram will differ from one that began at width 1.

Consider a SOP instance where the goal is to order the elements $[A, B, C, D]$, subject to the precedence constraint that A must precede D, an alphabetical ordering heuristic, and $w_m = 3$. Figure 2 shows $\underline{\mathcal{M}}(r)$, and $\underline{\mathcal{M}}(A)$ in three stages. The first stage is the initial 1-width diagram. The second stage is after one split on each layer, and the third stage is the complete diagram. The sub-graph shared by $\underline{\mathcal{M}}(r)$ and $\underline{\mathcal{M}}(A)$ is highlighted in blue, indicating that in this case the first two splits could have been read from $\underline{\mathcal{M}}(r)$ instead of being re-created from scratch. For the sake of legibility, arc values and arc labels are not included.

This mechanism can be embedded into a slightly modified version of the standard branch-and-bound algorithm based on decision diagrams (Algorithm 2). In peel-and-bound, the queue stores diagrams instead of nodes. After the initial relaxation $\underline{\mathcal{M}}(r)$ is generated, the entire diagram is placed into the queue Q such that $Q = \{\underline{\mathcal{M}}(r)\}$. Then a diagram $\underline{\mathcal{M}}(u)$ is selected from Q (for the first iteration $\underline{\mathcal{M}}(u) = \underline{\mathcal{M}}(r)$). However, instead of selecting an



$\underline{\mathcal{M}}(r)$ $\underline{\mathcal{M}}(A)$ Stage 1 $\underline{\mathcal{M}}(A)$ Stage 2 $\underline{\mathcal{M}}(A)$ Stage 3

Figure 2 Example of an induced sub-graph for a SOP instance (shown in blue), and the associated relaxed decision diagram with the same root.

exact cutset of $\underline{\mathcal{M}}(u)$, a single exact node e from $\underline{\mathcal{M}}(u)$ is selected. The process of selecting a diagram and exact node are heuristic decisions that are discussed in Section 3.1. The process of *peeling* e is as follows. Create an empty diagram $\underline{u}$, remove e from $\underline{\mathcal{M}}(u)$, and then put e into $\underline{u}$ such that e is the root of $\underline{u}$, and the arcs leaving e still end in $\underline{\mathcal{M}}(u)$. Then for each node y in $\underline{\mathcal{M}}(u)$ with an *in* arc that originates in $\underline{u}$, a new node y' is made and added to $\underline{u}$. Each *in* arc a_{oy} of y that originates in $\underline{u}$ is removed and then arc $a_{oy'}$ is added to $\underline{u}$. Then the *out* arcs of y and y' are filtered using the same *filter* function as Algorithm 1. The process of removing and adding arcs is repeated until there are no arcs ending in $\underline{\mathcal{M}}(u)$ that originate in $\underline{u}$. This procedure accomplishes a top-down reading of the sub-graph induced by e, and potentially strengthens $\underline{\mathcal{M}}(u)$ by removing nodes and arcs in the process. If the shortest path through the modified $\underline{\mathcal{M}}(u)$ is less than the best known solution, $\underline{\mathcal{M}}(u)$ is put back into Q. $\underline{u}$ is relaxed using Algorithm 1, let $\underline{\mathcal{M}}(\underline{u})$ be the result; then if the shortest path through the refined diagram $\underline{\mathcal{M}}(\underline{u})$ is less than the best known solution, $\underline{\mathcal{M}}(\underline{u})$ is added to Q. The whole procedure is repeated until there are no nodes left in the queue ($Q = \emptyset$). A peel operation is illustrated and explained in Figure 3. Peel-and-bound is formalized in Algorithm 3, and the peel operation is formalized in Algorithm 4.

Separating each node u during a peel requires creating a new node u', moving the *in* arcs of u that originate in the peeled diagram $\underline{u}$ to u', copying the *out* arcs of u to u', and then filtering the *out* arcs of u and u'. Creating a new node in our implementation has a time in $\mathcal{O}(n)$ due to storing state information that has a size in $\mathcal{O}(n)$ (such as $All^{\downarrow}_{u'}$). However, it is possible that in other applications the size of a node is in $\mathcal{O}(1)$. The number of *in* arcs of u is at most w, although this worst case is unlikely in practice because it requires $\underline{u}$ to have width w and for each node in $\underline{u}$ on layer $\ell_u - 1$ to have an arc ending at u. Thus, moving the *in* arcs of u has a time in $\mathcal{O}(w)$. The number of *out* arcs of u is at most n, and each arc has a size in $\mathcal{O}(1)$, so copying the *out* arcs has a time in $\mathcal{O}(n)$. Each individual filtering process has a time in $\mathcal{O}(1)$ as it uses only existing state information from u and u', and it is performed on the at most $2n$ *out* arcs of u and u'. Thus, filtering the *out* arcs has a time in $\mathcal{O}(n)$. Therefore, separating one node during the peel process has a time in $\mathcal{O}(n + w)$. Separations during a standard relaxation procedure require selecting a node ($\mathcal{O}(w)$), making a new node ($\mathcal{O}(n)$), partitioning the *in* arcs ($\mathcal{O}(nw)$), copying the *out* arcs ($\mathcal{O}(n)$), and filtering the *out* arcs ($\mathcal{O}(n)$). The reason that there can be more *in* arcs during a standard relaxation procedure is

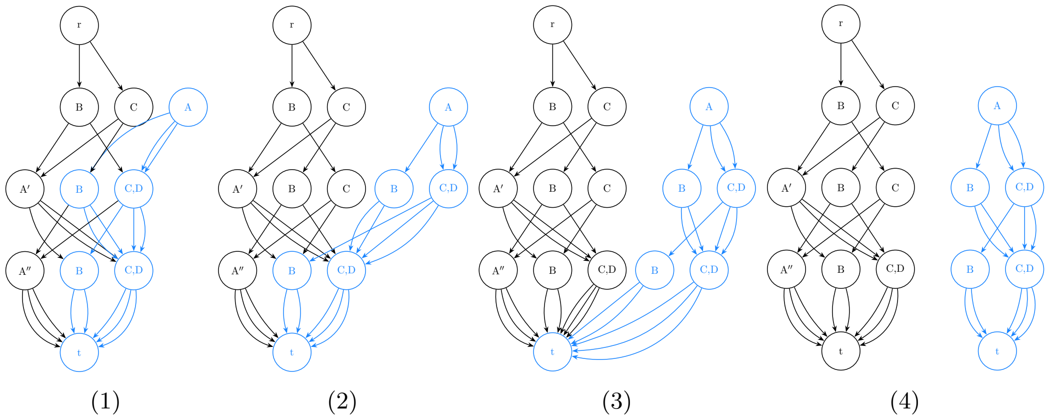


Figure 3 An example of a peel operation. In (1), A is selected to induce the peel process and removed from the the original diagram ($\underline{\mathcal{M}}(r)$ from Figure 2). In (2) the arcs that connect A to the original diagram are moved to copies of the nodes they originally ended at, and infeasible arcs are filtered. In (3) and (4) the process is repeated until the diagrams are disconnected.

5 6 7 8

Algorithm 3 Peel-and-Bound (PnB) Algorithm.

```
Let $v^*(u)$ be the lower bound of $\mathcal{P}$ resulting from starting at node $u$
Let $z_{opt}$ be the value of the best known solution
$Q = \{\underline{\mathcal{M}}(r)\}$
$z_{opt} \leftarrow \infty$
while $Q \neq \emptyset$ do
    $\mathcal{D} \leftarrow$selectDiagram$(Q)$, $Q \leftarrow Q \backslash \{\mathcal{D}\}$
    $u \leftarrow$selectExactNode$(\mathcal{D})$
    $\underline{u}, \mathcal{D}^* \leftarrow$ peel$(\mathcal{D}, u)$ (See Algorithm 4)
    if $v^*(\mathcal{D}^*) < z_{opt}$ then
        $Q \leftarrow Q \cup \{\mathcal{D}^*\}$
    end
    $\overline{\mathcal{M}} \leftarrow \overline{\mathcal{M}}(u)$
    if $v^*(\overline{\mathcal{M}}) < z_{opt}$ then
        $z_{opt} \leftarrow v^*(\overline{\mathcal{M}})$
    end
    if $\overline{\mathcal{M}}$ is not exact then
        $\underline{\mathcal{M}} \leftarrow \underline{\mathcal{M}}(\underline{u})$
        if $v^*(\underline{\mathcal{M}}) < z_{opt}$ then
            $Q \leftarrow Q \cup \{\underline{\mathcal{M}}\}$
        end
    end
end
return $z_{opt}$
```

because the nodes in a 1-width diagram can have *in* arcs with different labels coming from the same node, whereas the structure of the diagram during a peel guarantees that each node u can have only one *in* arc from each node on the layer $\ell_u - 1$. Thus, the total time for a separation in a standard relaxation is in $\mathcal{O}(nw)$.

Algorithm 4 Peeling process used in Algorithm 3.

```
Let $in(u)$ for some node $u$ be the set of arcs that end at node $u$
Let $out(u)$ for some node $u$ be the set of arcs that originate from node $u$
Let $in(\mathcal{M})$ for some MDD $\mathcal{M}$ be the set of arcs that end in $\mathcal{M}$
Let $out(\mathcal{M})$ for some MDD $\mathcal{M}$ be the set of arcs that originate in $\mathcal{M}$
input: a relaxed MDD $\mathcal{D}$, and an exact node $u$ in $\mathcal{D}$
Let $\underline{u}$ be an empty decision diagram
$in(u) \leftarrow \emptyset$
$\mathcal{D} \leftarrow \mathcal{D} \backslash u$
$\underline{u} \leftarrow u$
while $in(\mathcal{D}) \cap out(\underline{u}) \neq \emptyset$ do
    foreach node $m \in \mathcal{D}$ with an in arc that originates in $\underline{u}$ do
        create a new node $m'$, and add it to $\underline{u}$
        foreach arc $a \in in(m)$ that originates in $\underline{u}$ do
            change the destination of $a$ to $m'$
            filter($a$)
        end
        foreach arc $a \in out(m)$ do
            filter($a$)
        end
    end
end
while $\exists$ some node $m \in D$ with $in(m) = \emptyset$ or $out(m) = \emptyset$ (excluding $r$ and $t$) do
    $in(m) \leftarrow \emptyset$
    $out(m) \leftarrow \emptyset$
    $\mathcal{D} \leftarrow \mathcal{D} \backslash \{m\}$
end
return ($\underline{u}$, $\mathcal{D}$)
```

The maximum number of separations during a peel is the maximum number of nodes in the peeled diagram. A peeled diagram can have at most $(n-3) \times w + 2$ nodes, and thus the number of nodes is in $\mathcal{O}(nw)$. Therefore, the entire peel process has a time in $\mathcal{O}(n^2w + nw^2)$. The maximum number of separations during a standard relaxation is the exact same as during a peel, since the resulting diagram will be the same size. Thus, the standard relaxation has a total time in $\mathcal{O}(n^2w^2)$. However, peel-and-bound uses a peel to generate some fraction of the nodes, then a standard relaxation to generate the rest. Let α be the percent of nodes that are peeled during the peel. It follows that the total time for an iteration of peel-and-bound is in $\mathcal{O}(\alpha(n^2w + nw^2) + (1-\alpha)(n^2w^2))$. Therefore, the larger that α grows, the more time peel-and-bound saves over branch-and-bound.

3.1 Advantages and Implementation Decisions

The branch-and-bound algorithm proposed by Bergman et al. (2016) [4] requires selecting an exact cutset of $\underline{\mathcal{M}}$. Peel-and-bound requires selecting a diagram from the queue, and an exact node to start the peel process. The choice of node has a substantial impact on how quickly the process converges to an optimal solution, because it serves two purposes simultaneously. As discussed earlier, the first purpose of peeling is to avoid recreating a portion of the diagram at each iteration. The second purpose is to strengthen the overall relaxation. Let $\underline{u}$ be a diagram peeled from $\underline{\mathcal{M}}$, and let $\underline{\mathcal{M}}^*$ be $\underline{\mathcal{M}}$ after the peel operation. If $Sol(\mathcal{P}) \subseteq Sol(\underline{\mathcal{M}})$ then $Sol(\mathcal{P}) \subseteq Sol(\underline{\mathcal{M}}^*) \cup Sol(\underline{u})$. The only step of peel-and-bound that removes paths is the *filter* step, which only removes an arc if no feasible solutions can pass through that arc. If the node the peel is induced from contains the shortest path through $\underline{\mathcal{M}}$, then there will be a new shortest path through $\underline{\mathcal{M}}^*$ with $T^*(\underline{\mathcal{M}}^*) \geq T^*(\underline{\mathcal{M}})$. Similarly after peeling, the peeled diagram is going to be strengthened and $T^*(\mathcal{M}(\underline{u})) \geq T^*(\underline{u})$. Therefore, when implementing the *selectDiagram* and *selectExactNode* functions from Algorithm 3, we propose selecting the diagram $\mathcal{D}$ with the weakest bound, and an exact node from $\mathcal{D}$ that contains $z^*(\mathcal{D})$ at each iteration. Using these parameters, the peel step of peel-and-bound strengthens the relaxed bound of $\mathcal{P}$, in addition to providing a stronger initial diagram to use when generating $\mathcal{M}(\underline{u})$.

We propose two heuristics for selecting a node from $\mathcal{D}$ that contains $z^*(\mathcal{D})$. The first heuristic picks the first node in the shortest path through the diagram with at least one child that is not exact, we call this the *last exact node*. The second heuristic picks the *frontier* node, the highest-index exact node that contains $z^*(\mathcal{D})$. Taking the last exact node is more of a breadth-first search, taking the largest possible set of nodes that can be strengthened (anything above the last exact node is exact, and cannot be improved). In contrast, taking the frontier node is more of a depth-first search, taking fewer nodes and exploring those nodes at greater depth.

Cire and van Hoeve (2013) [9] propose that each iteration of Algorithm 1 starts from a 1-width MDD. However, for peel-and-bound with a non-separable objective function, starting from a 1-width MDD poses a problem. The arcs in such a diagram do not have exact values, because they are dependent on the state of the node they originate from. As nodes are peeled, the values of those arcs must be updated, and the operation becomes computationally expensive at scale. This problem can be avoided by creating the initial diagram using a structure where all of the arcs ending at a given node have the same label. The resulting initial diagram has a width of n, and each node on the layer is assigned to one state $s \in \{1, ..., n\}$. Then every possible feasible arc between consecutive layers is added. Thus, the nodes of $\underline{\mathcal{M}}$ do not have relaxed states, and each arc can only take one possible value. Starting from such a diagram not only removes the need to update arc values, it ensures that every arc generated during peel-and-bound is an exact copy of an arc that exists in the initial diagram, since arcs are only copied or removed, never updated or added. An alternative method of handling non-separable objective functions is explored by Hooker [15, 16, 17].

3.2 Limitations and Handling Memory

The focus of this paper is sequencing problems, but peel-and-bound can be easily applied to other optimization problems. However, some existing MDD based methods conflict with peel-and-bound. For example, some MDD algorithms use a dynamic variable order [18], such that the variables the layers on $\overline{M}$ are mapped to in one iteration of branch-and-bound, are different in the next. Peel-and-bound as it is presented in this paper cannot be paired with a dynamic variable order. Furthermore, the method in this paper is specific to decision diagrams generated using separation. We believe the method can be extended to decision diagrams that use a merge operator, but it has not been shown here.

Memory limitations present a problem for peel-and-bound in theory, but not in practice. Each open diagram remains in the queue, and thus must be stored in memory. However, this problem can be handled in many ways; two are given here. A dynamic method of handling the problem is to start targeting large diagrams with bounds close to z_{opt} as memory limitations start to become a problem. Such diagrams can usually be closed quickly, and subsequently removed from memory, freeing up space for the algorithm to continue. Alternatively, the diagrams with bounds closest to z_{opt} can be deleted in favor of storing just the root node, then when they need to be processed, initial diagrams are generated for those once again. This method essentially falls back to branch-and-bound until memory limitations cease to be a problem. Additional approaches for working with memory limitations, and evidence that the problem can be handled efficiently, are presented in Perez and Régin (2018) [23].

3.3 Integrating Rough Relaxed Bounds

This implementation incorporates the rough relaxed bounding method proposed by Gillard et al. [10]. Rough relaxed bounding was used to trim the domain of each node during construction of the restricted DDs, and was also added as a check to the *filter* function in Algorithm 1. When the initial model is created, a map is also created from each node u, to a list of the other nodes sorted by their distance from u. The rough relaxed bound $rrb(a)$ of an arc a_{fg} was calculated as follows. For each node u that has not necessarily been visited ($u \notin All^{\downarrow}_{g}$), look up the shortest distance from that node to a different node that has also not been visited. Then, sort the resulting list, and repeatedly remove the largest value until the list has a length equal to the number of remaining decisions. The sum of the values in the list, plus the value of the shortest path from r to the end of a, is the rough relaxed bound of a. If $rrb(a)$ is worse than the best known solution, the arc is removed.

4 Experiments on the Sequence Ordering Problem

The goal of this section is to assess the performances of the peel-and-bound algorithm (PnB, Algorithm 3). To do so, a comparison with the standard decision diagram based branch-and-bound algorithm (BnB, Algorithm 2) is proposed. Both algorithms are implemented in Julia and are open-source[2]. To ensure a fair comparison, both algorithms resort to the same function for generating relaxed decision diagrams (Algorithm 1), and the same function for generating restricted decision diagrams. While the functions being called are the same, there are two differences at run-time. At the end of line 26 in Algorithm 1, an additional operation runs during BnB where the values of the arcs leaving layer j are updated. The second difference is that BnB starts each relaxation from a 1-width DD, while PnB passes a partially completed diagram to the relaxation function as a starting point.

The testing environment was built from scratch to ensure a fair comparison, so it lacks the many propagators used by cutting-edge solvers like CPO to remove nodes from the PnB/BnB queue [2, 9]. However, it provides a clean comparison of the two algorithms by requiring that every function used by both BnB and PnB is exactly the same between the two, with the only differences arising due to PnB's ability to ensure that all arcs are exact from the beginning. All of the heuristic decisions that were made are identical for both algorithms.

4.1 Description of the Heuristics Considered

The *sequence ordering problem* can be considered as an asymmetric *travelling salesperson problem* with precedence constraints. The objective is to find a minimum cost path that visits each of the n elements exactly once, and respects the precedence constraints. The method

[2] https://github.com/IsaacRudich/PnB$_$SOP

used for generating relaxed DDs requires creating a heuristic ordering of all possible arc assignments by importance. The arc values in this case are representative of the n elements in the path. The ordering used was generated by sorting the n elements, first by their average distance from the other elements, and then by the number of elements each element must precede. The resulting order places a higher importance on elements that are far away from other elements and must precede many other elements.

The branch-and-bound algorithm processes nodes in an order designed to try and improve the existing relaxed bound at each iteration. When a node u is added to the BnB queue, it is assigned a value equal to the value of the shortest path from the root r to the terminal t, that passes through u. The best known relaxed bound on the problem is the smallest value of a node in the queue, and that node is always chosen to be processed. Peel-and-bound is implemented with the same goal of improving bounds at each iteration. However, PnB stores diagrams, not nodes. Let the value of a diagram be the value of the shortest path to the terminal. At each iteration of peel-and-bound, the diagram with the lowest value is selected, and then a node is chosen from that diagram to induce the peel process. All of the experiments here used a process where the selected node is the first node in the shortest path from r to t with a child node that is not exact (the last exact node). Testing was done to determine whether using the last exact node or the frontier node would perform better for the problem being considered, but there was not a significant difference between the two during any of the tests. Several of the benchmark problems were run using various decision diagram widths, and the last exact node was chosen because it sometimes showed a very slight improvement over the frontier node. While it is likely that this choice makes a difference on some problems, it does not matter for the SOP.

4.2 Experimental Results

The experiments were performed on a computer equipped with an AMD Rome 7532 at 2.40 GHz with 64Gb RAM. The solver was tested using DD widths of $64, 128$, and 256 on the 41 SOP problems available in TSPLIB [24]. For comparisons between PnB and BnB, a timestamp, new bounds, and the length of the remaining queue were recorded each time the bounds on a problem were improved. Another experiment was performed to test the scalability of PnB at width 2048, for which only the final bounds were recorded. Execution time was limited to $3,600$ seconds.

The smallest DD width tested for both methods was 64, and the largest DD width tested was 256. Table 2 has summary statistics for those widths as the percentage improvement demonstrated by PnB. A positive percentage always indicates that PnB performed better than BnB in that category, while a negative percentage indicates that BnB performed better. Figure 4 shows performance profiles for all of the experiments. Table 3 contains summary statistics comparing PnB at width 256 to PnB at width 2048, where a positive percentage always indicates that the width of 2048 performed better.

Table 2 Summary Statistics: percentage improvement of peel-and-bound over branch-and-bound. RB = Relaxed Bound, BS = Best Solution, OG = Optimality Gap, QL = Queue Length. Tables 4 and 5 in Appendix A show the comprehensive results.

	Width: 64				Width: 256			
	RB	BS	OG	QL	RB	BS	OG	QL
Average % Improvement	114%	0.5%	22.8%	1,647%	545%	3.3%	181%	308%
Median % Improvement	26%	0.05%	17.4%	734%	80%	1.7%	35%	141%

Table 3 Summary Statistics: percentage improvement of peel-and-bound at width 2048 over peel-and-bound at width 256. Table 6 in Appendix A shows the comprehensive results.

	PnB: 2048 v PnB: 256		
	Relaxed Bound	Best Solution	Optimality Gap
Average % Improvement	19.5%	0.8%	18.6%
Median % Improvement	16.3%	0.5%	13.7%

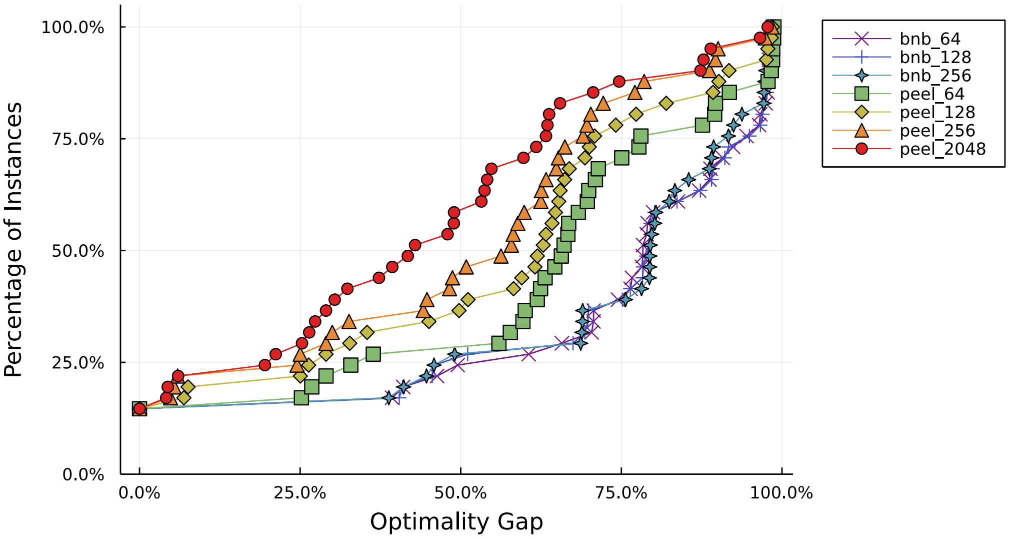
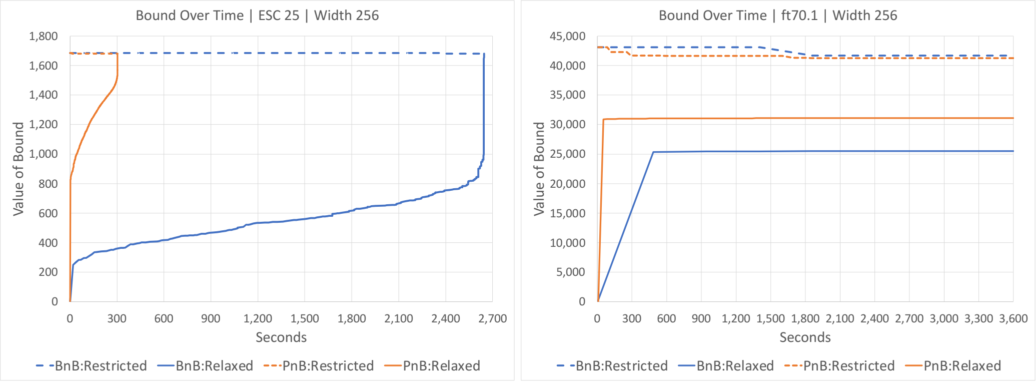


Figure 4 Performance Profiles: the optimality gap $= \frac{upper\_bound - lower\_bound}{upper\_bound}$.

As shown in Table 2, peel-and-bound vastly outperforms branch-and-bound in these experiments. The average and median improvements from using peel-and-bound at both widths are significant in terms of the relaxed bound, the remaining optimality gap, and the number of nodes that still need to be processed. The best solution found by the end of the runtime also tends to be slightly better with peel-and-bound, but the found solutions are often so close to the real optimal solutions that there is little room for improvement. At both widths, six of the problems were solved to optimality. BnB was faster in only one of those cases, and in that case the difference was .04 seconds. The median time for PnB to close in these cases was 191% faster at a width of 64, and 580% faster at a width of 256. The relaxed bound produced by PnB at a width of 64 was better for 28 of the remaining 35 problems, and at a width of 256 was better for 34 of the remaining 35 problems. The optimality gap was similarly better for peel-and-bound on every problem except the ones where branch-and-bound found a better relaxed bound. However, of the problems where branch-and-bound had a better optimality gap, the improvement was less than 1% for all but one problem. Figure 4 reinforces that even though there are some instances where a specific branch-and-bound setting slightly outperforms a specific peel-and-bound setting, the gap in those cases is small relative to the general gap between all peel-and-bound settings and all branch-and-bound settings.

As shown in Table 3, increasing the width to 2048 from 256 led to an 19.5% average improvement (16.3% median improvement) in the relaxed bound. Figure 4 shows that the performance of peel-and-bound nearly uniformly increases with the maximum allowable width. Similar to the difference between branch-and-bound and peel-and-bound, some specific instances see a small out-performance of the peel-and-bound running at a smaller width, but the gap is small relative to the usual gap between the 2048-width experiment and the rest of the experiments. Additionally, Figure 4 shows that peel_2048 solved 50% of instances to within a 42% optimality gap, peel_64 solved 50% of instances to within a 67% optimality gap, and the best performing branch and bound (bnb_64) solved 50% of instances to within only a 79% optimality gap. The overall performance of peel-and-bound improves as more problems are considered, especially as the maximum allowable width for the decision diagrams is increased.



Solved SOP

Unsolved SOP

Figure 5 Dual bounds of ESC25 and ft70.1 over the runtime of the experiment.

The selected graphs shown in Figure 5 are representative of the two main types of behavior observed over the problem set. On problems where the underlying relaxation method works well, the relaxed bound moves quickly towards convergence with the best found solution. On problems where the underlying relaxation does not work well, both algorithms slowly improve the relaxed bound, but PnB starts stronger as it can use exact arc values, and it maintains the advantage throughout. It is clear from the time-series data that to be competitive with cutting-edge solvers, peel-and-bound must be combined with other constraint programming propagators. However, it is also clear that peel-and-bound can have a significant edge over a propagator that generates the required decision diagrams from scratch at each iteration.

5 Conclusion and Future Work

This paper presented a peel-and-bound algorithm as an alternative to branch-and-bound. In peel-and-bound, constructed decision diagrams are repeatedly reused to avoid unnecessary computation. Additionally, peel-and-bound can be used in combination with a decision diagram structure that only admits exact arc values, to increase scalability and further reduce the amount of work needed at each iteration of the algorithm. We identified several heuristic decisions that can be used to adjust peel-and-bound, and provided insight into how the algorithm can be applied to other problems.

We compared the performance of a peel-and-bound scheme to a branch-and-bound scheme using the same DD based propagator. We tested both algorithms on the 41 instances of the SOP from TSPLIB. Results show that peel-and-bound significantly outperforms branch-and-bound on the SOP by generating substantially stronger relaxed bounds on instances that were not closed during the experiment, and reaching optimality faster when the instances were closed. This paper provides strong support for the value of re-using work in DD based solvers.

Furthermore, peel-and-bound benefits from scaling the maximum allowable width. Thus, relaxed DDs that yield strong bounds at scale, but are too costly to generate iteratively, only need to be constructed once. The method detailed in this paper focused on DDs generated by separation; future research could focus on DDs generated using a merge operator.

References

1 Henrik Andersen, Tarik Hadzic, John Hooker, and Peter Tiedemann. A constraint store based on multivalued decision diagrams. In *Bessière, C. (eds) Principles and Practice of Constraint Programming – CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 118–132, September 2007.

2 Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems.* International Series in Operations Research and Management Science, Kluwer. Springer International Publishing, 2001.

3 David Bergman, Andre Cire, Willem-Jan van Hoeve, and John Hooker. *Decision Diagrams for Optimization.* Springer International Publishing, January 2016.

4 David Bergman, Andre Cire, Willem-Jan van Hoeve, and John Hooker. Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28:47–66, February 2016.

5 David Bergman, André A. Cire, Ashish Sabharwal, Horst Samulowitz, Vijay Saraswat, and Willem-Jan van Hoeve. Parallel combinatorial optimization with decision diagrams. *Proceedings of the International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 351–367, 2014.

6 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.

7 Margarita Castro, Chiara Piacentini, Andre Cire, and J. Beck. Solving delete free planning with relaxed decision diagram based heuristics. *Journal of Artificial Intelligence Research*, 67:607–651, March 2020.

8 Margarita P. Castro, Andre A. Cire, and J. Christopher Beck. Decision diagrams for discrete optimization: A survey of recent advances, 2022. `arXiv:2201.11536`.

9 André A. Cire and Willem-Jan van Hoeve. Multivalued decision diagrams for sequencing problems. *Operations Research*, 61(6):1259, 1462, 2013.

10 Xavier Gillard, Vianney Coppé, Pierre Schaus, and André A. Cire. Improving the filtering of branch-and-bound mdd solver. *Integration of Constraint Programming, Artificial Intelligence, and Operations Research, 18th International Conference, CPAIOR 2021*, 2021.

11 Jaime González, Andre Cire, Andrea Lodi, and Louis-Martin Rousseau. Integrated integer programming and decision diagram search tree with an application to the maximum independent set problem. *Constraints*, 25, April 2020.

12 Tarik Hadzic, John Hooker, Barry O'Sullivan, and Peter Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In *Stuckey, P.J. (eds) Principles and Practice of Constraint Programming. CP 2008*, Lecture Notes in Computer Science, pages 448–462, September 2008.

13 Samid Hoda, Willem-Jan van Hoeve, and John Hooker. A systematic approach to mdd-based constraint programming. In *Cohen, D. (eds) Principles and Practice of Constraint Programming – CP 2010. CP 2010*, volume 6308 of *Lecture Notes in Computer Science*, pages 266–280, September 2010.

14 Willem-Jan Hoeve. Graph coloring with decision diagrams. *Mathematical Programming*, May 2021.

15 John Hooker. Decision diagrams and dynamic programming. In *Gomes, C., Sellmann, M. (eds) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. CPAIOR 2013*, volume 7874 of *Lecture Notes in Computer Science*, May 2013.

16 John Hooker. Job sequencing bounds from decision diagrams. In *Beck, J. (eds) Principles and Practice of Constraint Programming. CP 2017*, Lecture Notes in Computer Science, pages 565–578, August 2017.

17 John Hooker. Improved job sequencing bounds from decision diagrams. In *Schiex, T., de Givry, S. (eds) Principles and Practice of Constraint Programming. CP 2019*, volume 11802 of *Lecture Notes in Computer Science*, pages 268–283, September 2019. doi:10.1007/978-3-030-30048-7_16.

18 Anthony Karahalios and Willem-Jan Hoeve. Variable ordering for decision diagrams: A portfolio approach. *Constraints*, pages 1–18, January 2022.

19 Anna Latour, Behrouz Babaki, and Siegfried Nijssen. Stochastic constraint propagation for mining probabilistic networks. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pages 1137–1145, August 2019.

20 Leonardo Lozano and J. Smith. A binary decision diagram based algorithm for solving a class of binary two-stage stochastic programs. *Mathematical Programming*, August 2018.

21 Johannes Maschler and Günther Raidl. Multivalued decision diagrams for prize-collecting job sequencing with one common and multiple secondary resources. *Annals of Operations Research*, 302, July 2021.

22 Augustin Parjadis, Quentin Cappart, Louis-Martin Rousseau, and David Bergman. Improving branch-and-bound using decision diagrams and reinforcement learning. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 446–455. Springer, 2021.

23 Guillaume Perez and Jean-Charles Régin. Parallel algorithms for operations on multi-valued decision diagrams. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), April 2018.

24 Gerhard Reinelt. Tsplib. a traveling salesman problem library. *INFORMS Journal on Computing*, 3:376–384, November 1991.

25 Thiago Serra and John Hooker. Compact representation of near-optimal integer programming solutions. *Mathematical Programming*, 182, April 2019.

26 Diego Uña, Graeme Gange, Peter Schachte, and Peter Stuckey. Compiling cp subproblems to mdds and d-dnnfs. *Constraints*, 24, January 2019.

27 Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Compact-mdd: Efficiently filtering (s)mdd constraints with reversible sparse bit-sets. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 1383–1389, July 2018.

28 Julien Vion and Sylvain Piechowiak. From mdd to bdd and arc consistency. *Constraints*, 23, October 2018.

A Experimental Data

Table 4 Comparison Data for width 64 experiments: RB = Relaxed Bound, BS = Best Solution, T = Time in Seconds, OG = Optimality Gap, QL = Queue Length. Full time series data is available in the GitHub repository.

Problem Info		BnB: width 64					PnB: width 64					Percent Improvements				
Name	n	RB	BS	T	OG	QL	RB	BS	T	OG	QL	RB	BS	T	OG	QL
ESC07	9	2,125	2,125	0.03	0%	-	2,125	2,125	0.07	0%	-			-57%		
ESC11	13	2,075	2,075	0.65	0%	-	2,075	2,075	0.42	0%	-			55%		
ESC12	14	1,675	1,675	1.99	0%	-	1,675	1,675	0.64	0%	-			211%		
ESC25	27	1,681	1,681	956	0%	-	1,681	1,681	353	0%	-			171%		
ESC47	49	334	1,542		78%	8,842	368	1,676		78%	1,295	10.2%	-8.0%		0.4%	583%
ESC63	65	8	62		87%	2,756	44	62		29%	15	450.0%	0.0%		200.0%	18273%
ESC78	80	2,230	19,800		89%	1,040	5,000	20,045		75%	316	124.2%	-1.2%		18.2%	229%
br17.10	18	55	55	260	0%	-	55	55	5	0%	-			4652%		
br17.12	18	55	55	138	0%	-	55	55	21	0%	-			546%		
ft53.1	54	1,785	8,478		79%	8,841	3,324	8,244		60%	917	86.2%	2.8%		32.3%	864%
ft53.2	54	1,946	8,927		78%	7,356	3,450	8,633		60%	938	77.3%	3.4%		30.3%	684%
ft53.3	54	2,546	12,179		79%	5,594	4,234	12,327		66%	1,147	66.3%	-1.2%		20.5%	388%
ft53.4	54	3,780	14,811		74%	11,907	6,500	14,753		56%	2,372	72.0%	0.4%		33.1%	402%
ft70.1	71	25,444	41,926		39%	4,781	31,123	41,607		25%	412	22.3%	0.8%		56.0%	1060%
ft70.2	71	25,239	42,805		41%	3,998	31,195	42,623		27%	427	23.6%	0.4%		53.1%	836%
ft70.3	71	25,810	48,073		46%	4,036	31,872	47,491		33%	475	23.5%	1.2%		40.8%	750%
ft70.4	71	28,593	56,644		50%	8,642	35,974	56,552		36%	1,087	25.8%	0.2%		36.1%	695%
kro124p.1	101	10,773	46,158		77%	2,173	17,579	46,158		62%	105	63.2%	0.0%		23.8%	1970%
kro124p.2	101	11,061	46,930		76%	1,898	17,633	46,930		62%	109	59.4%	0.0%		22.4%	1641%
kro124p.3	101	12,110	55,991		78%	1,055	18,586	55,991		67%	117	53.5%	0.0%		17.3%	802%
kro124p.4	101	13,838	85,533		84%	2,990	24,388	85,316		71%	244	76.2%	0.3%		17.4%	1125%
p43.1	44	630	29,450		98%	12,945	380	29,380		99%	1,022	-39.7%	0.2%		-0.9%	1167%
p43.2	44	440	29,000		98%	8,519	420	29,080		99%	1,125	-4.5%	-0.3%		-0.1%	657%
p43.3	44	595	29,530		98%	12,802	490	29,530		98%	1,122	-17.6%	0.0%		-0.4%	1041%
p43.4	44	1,370	83,855		98%	21,105	1,050	83,890		99%	4,694	-23.4%	0.0%		-0.4%	350%
prob.42	42	99	289		66%	16,742	97	286		66%	2,613	-2.0%	1.0%		-0.5%	541%
prob.100	100	170	1,841		91%	1,731	182	1,760		90%	117	7.1%	4.6%		1.2%	1379%
rbg048a	50	76	379		80%	12,938	47	380		88%	1,551	-38.2%	-0.3%		-8.8%	734%
rbg050c	52	63	566		89%	11,480	154	512		70%	1,481	144.4%	10.5%		27.1%	675%
rbg109a	111	91	1,196		92%	2,773	379	1,196		68%	612	316.5%	0.0%		35.3%	353%
rbg150a	152	63	1,874		97%	241	565	1,865		70%	222	796.8%	0.5%		38.6%	9%
rbg174a	176	119	2,157		94%	809	626	2,156		71%	117	426.1%	0.0%		33.1%	591%
rbg253a	255	113	3,181		96%	403	708	3,180		78%	39	526.5%	0.0%		24.1%	933%
rbg323a	325	89	3,519		97%	437	289	3,529		92%	17	224.7%	-0.3%		6.2%	2471%
rbg341a	343	68	3,038		98%	366	321	3,064		90%	8	372.1%	-0.8%		9.2%	4475%
rbg358a	360	69	3,359		98%	289	73	3,373		98%	6	5.8%	-0.4%		0.1%	4717%
rbg378a	380	52	3,429		98%	266	50	3,429		99%	5	-3.8%	0.0%		-0.1%	5220%
ry48p.1	49	5,201	17,555		70%	10,480	6,171	17,454		65%	1,395	18.7%	0.6%		8.9%	651%
ry48p.2	49	5,291	18,046		71%	9,286	6,577	17,840		63%	1,445	24.3%	1.2%		12.0%	543%
ry48p.3	49	6,207	21,161		71%	9,039	6,985	20,962		67%	1,707	12.5%	0.9%		6.0%	430%
ry48p.4	49	13,610	34,517		61%	15,819	14,293	33,804		58%	3,217	5.0%	2.1%		4.9%	392%

Table 5 Comparison data for width 256 experiments: RB = Relaxed Bound, BS = Best Solution, T = Time in Seconds, OG = Optimality Gap, QL = Queue Length. Full time series data is available in the GitHub repository.

Problem Info		BnB: width 256					PnB: width 256					Percent Improvements				
Name	n	RB	BS	T	OG	QL	RB	BS	T	OG	QL	RB	BS	T	OG	QL
ESC07	9	2,125	2,125	0.04	0%	-	2,125	2,125	0.04	0%	-			0%		
ESC11	13	2,075	2,075	0.48	0%	-	2,075	2,075	0.41	0%	-			17%		
ESC12	14	1,675	1,675	1.66	0%	-	1,675	1,675	0.34	0%	-			388%		
ESC25	27	1,681	1,681	2,643	0%	-	1,681	1,681	303	0%	-			771%		
ESC47	49	312	1,590		80%	720	658	1,339		51%	740	110.9%	18.7%		58.0%	-3%
ESC63	65	9	62		85%	53	44	62		29%	3	388.9%	0.0%		194.4%	1667%
ESC78	80	2,230	20,345		89%	59	5,600	20,135		72%	109	151.1%	1.0%		23.3%	-46%
br17.10	18	55	55	275	0%	-	55	55	3	0%	-			9468%		
br17.12	18	55	55	105	0%	-	55	55	5	0%	-			2146%		
ft53.1	54	1,708	8,424		80%	760	4,603	8,244		44%	271	169.5%	2.2%		80.5%	180%
ft53.2	54	1,856	9,059		80%	632	3,555	8,648		59%	272	91.5%	4.8%		35.0%	132%
ft53.3	54	2,493	12,598		80%	477	4,852	11,095		56%	390	94.6%	13.5%		42.6%	22%
ft53.4	54	3,619	14,867		76%	1,240	7,560	14,611		48%	797	108.9%	1.8%		56.8%	56%
ft70.1	71	25,507	41,686		39%	373	31,122	41,235		25%	108	22.0%	1.1%		58.3%	245%
ft70.2	71	25,261	42,901		41%	297	31,630	42,182		25%	123	25.2%	1.7%		64.4%	141%
ft70.3	71	25,891	47,806		46%	377	32,539	46,488		30%	151	25.7%	2.8%		52.8%	150%
ft70.4	71	31,186	56,366		45%	958	37,984	56,366		33%	356	21.8%	0.0%		37.0%	169%
kro124p.1	101	10,683	48,866		78%	152	19,224	45,643		58%	43	79.9%	7.1%		35.0%	253%
kro124p.2	101	10,706	52,038		79%	125	19,299	48,102		60%	43	80.3%	8.2%		32.6%	191%
kro124p.3	101	12,078	58,562		79%	64	20,145	57,358		65%	45	66.8%	2.1%		22.3%	42%
kro124p.4	101	14,511	82,672		82%	281	25,002	82,364		70%	102	72.3%	0.4%		18.4%	175%
p43.1	44	610	29,460		98%	1,033	27,255	28,635		5%	146	4368%	2.9%		1932%	608%
p43.2	44	460	29,020		98%	547	27,455	29,020		5%	391	5868%	0.0%		1725%	40%
p43.3	44	750	29,530		97%	1,016	27,780	29,530		6%	764	3604%	0.0%		1545%	33%
p43.4	44	1,425	83,880		98%	1,365	28,195	83,435		66%	1,380	1879%	0.5%		48.5%	-1%
prob.42	42	90	289		69%	1,166	103	275		63%	617	14.4%	5.1%		10.1%	89%
prob.100	100	157	1,886		92%	113	178	1,721		90%	45	13.4%	9.6%		2.3%	151%
rbg048a	50	80	389		79%	794	80	373		79%	534	0.0%	4.3%		1.1%	49%
rbg050c	52	62	583		89%	810	175	503		65%	442	182.3%	15.9%		37.0%	83%
rbg109a	111	89	1,181		92%	394	406	1,106		63%	204	356.2%	6.8%		46.1%	93%
rbg150a	152	115	1,845		94%	406	571	1,845		69%	100	396.5%	0.0%		35.8%	306%
rbg174a	176	362	2,172		83%	337	646	2,171		70%	57	78.5%	0.0%		18.6%	491%
rbg253a	255	359	3,177		89%	139	727	3,176		77%	22	102.5%	0.0%		15.0%	532%
rbg323a	325	99	3,476		97%	114	346	3,480		90%	14	249.5%	-0.1%		7.9%	714%
rbg341a	343	84	3,016		97%	120	340	3,016		89%	7	304.8%	0.0%		9.6%	1614%
rbg358a	360	88	3,280		97%	92	88	3,382		97%	5	0.0%	-3.0%		-0.1%	1740%
rbg378a	380	44	3,385		99%	35	53	3,385		98%	6	20.5%	0.0%		0.3%	483%
ry48p.1	49	5,470	17,464		69%	897	9,432	17,071		45%	377	72.4%	2.3%		53.5%	138%
ry48p.2	49	5,606	18,060		69%	834	6,615	17,627		62%	383	18.0%	2.5%		10.4%	118%
ry48p.3	49	6,558	21,142		69%	859	8,723	20,850		58%	513	33.0%	1.4%		18.6%	67%
ry48p.4	49	17,359	34,074		49%	1,557	17,322	33,773		49%	990	-0.2%	0.9%		0.7%	57%

Table 6 Comparison of PnB at 2048 over PnB at 256: RB = Relaxed Bound, BS = Best Solution, OG = Optimality Gap.

Problem Info		PnB: width 256			PnB: width 2048			Percent Improvements		
Name	n	RB	BS	OG	RB	BS	OG	RB	BS	OG
ESC47	49	658	1,339	51%	882	1,304	32%	34.0%	2.7%	57.2%
ESC63	65	44	62	29%	44	62	29%	0.0%	0.0%	0%
ESC78	80	5,600	20,135	72%	6,025	20,505	71%	7.6%	-1.8%	2.2%
ft53.1	54	4,603	8,244	44%	5,167	8,237	37%	12.3%	0.1%	18.5%
ft53.2	54	3,555	8,648	59%	4,910	8,598	43%	38.1%	0.6%	37.3%
ft53.3	54	4,852	11,095	56%	7,722	11,092	30%	59.2%	0.0%	85.2%
ft53.4	54	7,560	14,611	48%	7,466	14,618	49%	-1.2%	0.0%	-1.4%
ft70.1	71	31,122	41,235	25%	33,382	41,476	20%	7.3%	-0.6%	25.7%
ft70.2	71	31,630	42,182	25%	32,964	41,833	21%	4.2%	0.8%	18.0%
ft70.3	71	32,539	46,488	30%	34,366	46,001	25%	5.6%	1.1%	18.6%
ft70.4	71	37,984	56,366	33%	40,919	56,310	27%	7.7%	0.1%	19.3%
kro124p.1	101	19,224	45,643	58%	21,954	47,425	54%	14.2%	-3.8%	7.8%
kro124p.2	101	19,299	48,102	60%	22,746	49,571	54%	17.9%	-3.0%	10.7%
kro124p.3	101	20,145	57,358	65%	25,566	54,633	53%	26.9%	5.0%	21.9%
kro124p.4	101	25,002	82,364	70%	29,377	81,050	64%	17.5%	1.6%	9.2%
p43.1	44	27,255	28,635	5%	27,755	28,960	4%	1.8%	-1.1%	16%
p43.2	44	27,455	29,020	5%	27,725	29,000	4%	1.0%	0.1%	23%
p43.3	44	27,780	29,530	6%	27,755	29,530	6%	-0.1%	0.0%	-1%
p43.4	44	28,195	83,435	66%	28,680	83,020	65%	1.7%	0.5%	1.2%
prob.42	42	103	275	63%	152	261	42%	47.6%	5.4%	49.8%
prob.100	100	178	1,721	90%	220	1,735	87%	23.6%	-0.8%	2.7%
rbg048a	50	80	373	79%	93	367	75%	16.3%	1.6%	5.2%
rbg050c	52	175	503	65%	184	501	63%	5.1%	0.4%	3.1%
rbg109a	111	406	1,106	63%	453	1,126	60%	11.6%	-1.8%	5.9%
rbg150a	152	571	1,845	69%	672	1,841	63%	17.7%	0.2%	8.7%
rbg174a	176	646	2,171	70%	1,104	2,121	48%	70.9%	2.4%	46.5%
rbg253a	255	727	3,176	77%	1,186	3,101	62%	63.1%	2.4%	24.9%
rbg323a	325	346	3,480	90%	421	3,449	88%	21.7%	0.9%	2.6%
rbg341a	343	340	3,016	89%	329	2,965	89%	-3.2%	1.7%	-0.2%
rbg358a	360	88	3,382	97%	107	3,131	97%	21.6%	8.0%	0.8%
rbg378a	380	53	3,385	98%	74	3,338	98%	39.6%	1.4%	0.7%
ry48p.1	49	9,432	17,071	45%	10,386	17,124	39%	10.1%	-0.3%	13.7%
ry48p.2	49	6,615	17,627	62%	7,896	17,461	55%	19.4%	1.0%	14.0%
ry48p.3	49	8,723	20,850	58%	10,558	20,686	49%	21.0%	0.8%	18.8%
ry48p.4	49	17,322	33,773	49%	24,248	32,953	26%	40.0%	2.5%	84.4%

On Quantitative Testing of Samplers

Mate Soos
National University of Singapore, Singapore

Priyanka Golia
Indian Institute of Technology Kanpur, India
National University of Singapore, Singapore

Sourav Chakraborty
Indian Statistical Institute Kolkata, India

Kuldeep S. Meel
National University of Singapore, Singapore

Abstract

The problem of uniform sampling is, given a formula F, sample solutions of F uniformly at random from the solution space of F. Uniform sampling is a fundamental problem with widespread applications, including configuration testing, bug synthesis, function synthesis, and many more. State-of-the-art approaches for uniform sampling have a trade-off between scalability and theoretical guarantees. Many state of the art uniform samplers do not provide any theoretical guarantees on the distribution of samples generated, however, empirically they have shown promising results. In such cases, the main challenge is to test whether the distribution according to which samples are generated is indeed uniform or not.

Recently, Chakraborty and Meel (2019) designed the first scalable sampling tester, Barbarik, based on a grey-box sampling technique for testing if the distribution, according to which the given sampler is sampling, is close to the uniform or far from uniform. They were able to show that many off-the-self samplers are far from a uniform sampler. The availability of Barbarik increased the test-driven development of samplers. More recently, Golia, Soos, Chakraborty and Meel (2021), designed a uniform like sampler, CMSGen, which was shown to be accepted by Barbarik on all the instances. However, CMSGen does not provide any theoretical analysis of the sampling quality.

CMSGen leads us to observe the need for a tester to provide a quantitative answer to determine the quality of underlying samplers instead of merely a qualitative answer of Accept or Reject. Towards this goal, we design a computational hardness-based tester ScalBarbarik that provides a more nuanced analysis of the quality of a sampler. ScalBarbarik allows more expressive measurement of the quality of the underlying samplers. We empirically show that the state-of-the-art sampler, CMSGen is not accepted as a uniform-like sampler by ScalBarbarik. Furthermore, we show that ScalBarbarik can be used to design a sampler that can achieve balance between scalability and uniformity.

2012 ACM Subject Classification Theory of computation; Computing methodologies → Artificial intelligence

Keywords and phrases SAT Sampling, Testing of Samplers, SAT Solvers

Digital Object Identifier 10.4230/LIPIcs.CP.2022.36

Supplementary Material *Software (Source Code)*: https://github.com/meelgroup/scalbarbarik

Funding This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004], Ministry of Education Singapore Tier 2 grant [MOE-T2EP20121-0011], NUS ODPRT grant [R-252-000-685-13], and Amazon Research Award.

Acknowledgements We are grateful to the anonymous reviewers for constructive comments that significantly improved the final version of the paper. We are thankful to Yash Pote for his detailed feedback on the early drafts of the paper. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore: https://www.nscc.sg.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 36; pp. 36:1–36:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Given a formula F over the set of variables X, the problem of Boolean satisfiability (SAT) is to determine whether there exists an assignment σ to X such that F evaluates true under σ. The past two decades have witnessed a dramatic improvement in the runtime of SAT solvers owing to the Conflict Driven Clause Learning (CDCL) paradigm, and as a result, SAT solvers find applications in diverse areas ranging from constrained-random verification [19], computational biology [10], and artificial intelligence. The progress in SAT solving has led to development of algorithmic and practical implementations for problems in complexity classes beyond NP. One such problem that has seen a sustained interest over the past decade is that of uniform sampling. The problem of uniform sampling is to sample satisfying assignments of a formula uniformly at random from the space of satisfying assignments of the formula. Like SAT solver, uniform sampling also has wide variety of applications, like in configuration testing [7, 15], constrained-random simulation [19], bug synthesis [21], and function synthesis [12].

The last decade has seen several algorithmic proposals for efficient uniform sampling owing to its diverse applications. The different techniques for uniform sampling can be divided into two categories: (1) techniques that provide theoretical guarantees on the distribution from which the samples are generated, and (2) techniques that do not provide any theoretical guarantees on the samples produced. The hashing-based sampler UniGen, UniGen3 [6, 5, 23], and the knowledge compilation-based sampler KUS [22] fall in the first category, however, experimental evaluation shows that these samplers could not always achieve scalability for real world instances. At the same time, there exist many other sampling techniques, such as the mutation-based QuickSampler [8] and BDD-based techniques [16], or randomized CDCL SAT solvers [13] that can provide empirical scalability, however do not provide guarantees on the distribution of samples generated.

Algorithmic proposals that cannot provide theoretical grantees on the distribution of samples generated often rely on statistical test such as KL-divergence [17] to showcase the quality of the samples generated. These statistical tests are only able to show that samples produced by the samplers for a small set of benchmarks are close to samples produced from a uniform distribution. However, such tests do not generalize over entire benchmark sets. Recently, Chakraborty and Meel proposed the first scalable sampling test framework, Barbarik [3], to test whether a sampler under test (SUT[1]) is close to uniform or not. The tester Barbarik takes an (1) SUT, a (2) base uniform sampler, a (3) tolerance parameter ε, an (4) intolerance parameter η, a (4) confidence parameter δ, and a (5) formula φ and returns Accept if SUT is close to a uniform sampler. Barbarik returns Reject only if the SUT is far from a uniform sampler under *subquery-consistency* assumption, which is to assume that the SUT does not change its sampling behavior during the test, that is, off the shelf samplers would be sub-query consistent[2].

The main idea behind Barbarik is to reduce the input formula φ to $\hat{\varphi}$ using two satisfying assignments of φ chosen uniformly at random from the solution space of φ. One assignment, say σ_1 is drawn using the SUT, and another assignment, say σ_2 is drawn according to uniform distribution using the base sampler. The analysis for Barbarik shows that if the distribution from which the SUT is sampling the assignments is close to uniform distribution, the conditional distribution over $\{\sigma_1, \sigma_2\}$ is also close to uniform. Similarly, if the distribution

[1] The term SUT is from software testing literature, where it is shorthand for System Under Test.

[2] We rename the notion of *non-adversarial* assumption introduced in [3] to subquery-consistency to better capture its intended properties. We formally define the subquery-consistent assumption in Section 2.

from which the SUT is sampling the assignments is far from uniform, the conditional distribution over $\{\sigma_1, \sigma_2\}$ is also far from uniform. It is easy to estimate the distance of conditional distribution over $\{\sigma_1, \sigma_2\}$ to uniform distribution using random samples from $\hat{\varphi}$. Empirically, it was shown that Barbarik accepts UniGen3, which is a sampler with theoretical grantees, however, it rejects the state of the art uniform-like samplers, that is, samplers without theoretical guarantees, such as QuickSampler [8] and STS [9]. Recently, Meel et al. generalize the idea of Barbarik to handle any arbitrary weight function, that is, to test whether a SUT generates samples according to a given distribution [18].

Recently, Golia, et al. used Barbarik in a test-driven development fashion to create the uniform-like sampler CMSGen [13] from the state-of-art SAT solver CryptoMiniSat [24]. CMSGen is based on randomization of the conflict-driven-clause-learning (CDCL) framework inside CryptoMiniSat and most modern SAT solvers. Based on the feedback from Barbarik, the authors iteratively changed the hyper-parameters of CryptoMiniSat such as restart intervals, restart types, polarity picking heuristics and the like, until they arrived at a point where it was able to pass all tests. Analyzing the CDCL itself is a hard problem, and so the resulting uniform-like sampler, CMSGen, could not provide theoretical guarantees on the distribution of samples produced. However, it was shown in [13] that Barbarik returns Accept for CMSGen.

The development of samplers such as CMSGen poses an interesting question regarding test frameworks such as Barbarik: is it possible that uniform-like samplers such as CMSGen pass the test, but they are not uniform? If so, how can one demonstrate that they are not? These questions point towards revisiting the design of sampler test frameworks such as Barbarik. We need a tester that provides a quantitative analysis instead of qualitative answer of Accept or Reject to measure the quality of samplers.

The above stated goal to improve sampling testers requires new insights about the workings of samplers. The improvement of Barbarik that we are envisioning is to generate input formulas that are specifically crafted to highlight non-uniformity in the samples produced by the samplers. Towards this goal, we propose the framework ScalBarbarik.

Contributions

The success of CMSGen and the current lack of theoretical analysis leads us to hypothesize that CMSGen may not be uniform for all the formulas but is not necessarily far from uniform for a large class of formulas. The current framework of Barbarik provides too coarse grained analysis to allow users to determine the quality of distributions generated by a sampler such as CMSGen. To achieve such a fine-grained analysis, we need a parameterized generation of $\hat{\varphi}$. To this end, we design an improved algorithm, Shakuni, for construction of $\hat{\varphi}$ such that $\hat{\varphi}$ is composed of two sub-formulas with varying computational hardness.

We augment Barbarik with Shakuni to obtain ScalBarbarik that can provide fine-grained analysis with respect to hardness dial provided by Shakuni. ScalBarbarik allows us to view that the distribution quality of CMSGen is better than samplers such as QuickSampler but falls short of samplers with rigorous guarantees such as UniGen. ScalBarbarik can then be used to fine-tune a heuristic-based uniform-like sampler such as CMSGen to achieve a different balance between scalability and uniformity. Towards this, we empirically analyze the distribution of samples generated by CMSGen with different restart intervals. We then show that CMSGen could generate samples from a close to uniform distribution with increased restart intervals, sacrificing speed for better uniformity.

It is worth remarking that an important strength of ScalBarbarik is its simplicity. Based on our empirical analysis, ScalBarbarik with varying computational hardness is able to show that CMSGen is not a uniform sampler. The availability of ScalBarbarik has the potential to spur a virtuous cycle of development of samplers and testing techniques: the developers can design sampling methods that can be accepted by testers such as Barbarik/ScalBarbarik and consequently improve testers so that such samplers are rejected in the following version of it. With the help of ScalBarbarik, we can tune a sampler to achieve the balance between scalability and uniformly. Our experimental evaluation demonstrates that as we increase the restart intervals of CMSGen, we need to increase the computational hardness of ScalBarbarik to reject CMSGen, that is, with increased restart intervals CMSGen is able to generate samples from a close to uniform distribution; however, it takes longer time to generate the samples. The availability of ScalBarbarik allows us to improve to samplers such as CMSGen.

The rest of the paper is organized as follows: In Section 2, we present the formal definitions and also present a brief description of state-of-the-art tester Barbarik. In Section 3, we present the improved test framework ScalBarbarik based on a cryptographically hard function. We provide a detailed algorithmic description in Section 4, and we present the experimental evaluation in Section 5. Finally, we conclude in Section 6.

2 Notation and Background

A literal is a Boolean variable or its negation. A formula is considered to be in conjunctive normal form (CNF) if the formula is conjunction of clauses. A clause is a disjunction of literals. Let φ be the formula in CNF, and let $\mathit{Supp}(\varphi)$ represent the set of variables in φ. A satisfying assignment to φ is an assignment of truth values to $\mathit{Supp}(\varphi)$ under which the formula φ evaluates to True. Let σ be a satisfying assignment of φ, and let $S \subseteq \mathit{Supp}(\varphi)$, $\sigma_{\downarrow S}$ represents the projection of σ over S. Let R_φ be the set of all satisfying assignments of formula φ. We used $L[n : m]$ to represent the substring of L, starting with position n to m.

Chain Formulas. Chain formulas were introduced in [4]. Given positive integers k and m, chain formulas are Boolean formulas with exactly k satisfying solutions with $\lceil \log(k) \rceil \leq m$ variables.

▶ **Definition 1** ([4]). *Let $c_1, c_2, \ldots, c_m$ be the m-bit binary representation of k, where c_m is the least significant bit. A chain formula $\varphi_{k,m}(.)$ on m Boolean variables $v_1, v_2, \ldots, v_m$ is as follows:*
For every j in $\{1, \ldots, m-1\}$, let C_j be the connector "$\vee$" if $c_j = 1$, and the connector "$\wedge$" if $c_j = 0$, and the formula $\varphi_{k,m}(v_1, v_2, \ldots, v_m) = v_1 C_1 (v_2 C_2 (\ldots (v_{m-1} C_{m-1} v_m)))$

A Sampler. A CNF sampler or simply a sampler takes a formula φ, a number of required satisfying assignments N, $S \subseteq \mathit{Supp}(\varphi)$, and returns satisfying assignments $\sigma_{1_{\downarrow S}}, \sigma_{2_{\downarrow S}}, \ldots, \sigma_{N_{\downarrow S}}$. A uniform sampler, say $\mathcal{G}$ takes $\varphi, N, S \subseteq \mathit{Supp}(\varphi)$ that generates a satisfying assignment σ_i for all $i \in \{1, N\}$ with probability $\dfrac{1}{|R_\varphi|}$. Similarly, a sampler is considered to be an additive almost-uniform sampler, if the following holds with $0 \leq \varepsilon \leq 1$:

$$\forall \sigma \in R_\varphi, \frac{1-\varepsilon}{|R_\varphi|} \leq Pr[\mathcal{G}(\varphi, N) = \sigma] \leq \frac{1+\varepsilon}{|R_\varphi|}$$

We use a sampler $\mathcal{G}(.,.,.)$ or $\mathcal{G}(.,.)$ when S is $\mathit{Supp}(\varphi)$, or simply $\mathcal{G}$ when N and S are clear from context. We use $p_{\mathcal{G}(.,.,.)}$ to denote the probability with that $\mathcal{G}$ samples a satisfying assignment σ, and $D_{\mathcal{G}(\varphi,.,.)}$ to denote the distribution induced by sampler G over solution space of φ.

Given a formula φ, and an intolerance parameter η, a sampler $\mathcal{G}$ is considered to be η-far from a uniform sampler if ℓ_1 distance between the distribution induced by $\mathcal{G}$ over solution space of φ to the uniform distribution is at least η, that is,

$$\sum_{x \in sol(\varphi)} \left| p_{\mathcal{G}(\varphi,x)} - \frac{1}{|sol(\varphi)|} \right| \geq \eta$$

A Sampler Tester. Given a uniform sampler, a sampler tester tests if the sampler is sampling an assignment from the solution space R_φ, and the samples are generated from a close to uniform distribution. A sampler test framework is defined as follows:

▶ **Definition 2.** *Given a Boolean formula φ, a sampler $\mathcal{G}$, a tolerance parameter ε, an intolerance parameter η, a confidence parameter δ, a sampler tester $\mathcal{T}(\cdot,\cdot,\cdot,\cdot,\cdot)$ returns* Accept *or* Reject *(with a witness) with the following guarantees:*

1. *If the sampler $\mathcal{G}(\varphi,.,)$ is an additive almost-uniform generator, then $\mathcal{T}(G,\varphi,\varepsilon,\eta,\delta)$ returns* Accept *with probability at least $1-\delta$*
2. *If the sampler $\mathcal{G}(\varphi,.,)$ is η-far from uniform generator, then $\mathcal{T}(G,\varphi,\varepsilon,\eta,\delta)$ returns* Reject *with probability at least $1-\delta$*

Barbarik

Chakraborty and Meel [3] designed the tester Barbarik that takes a base uniform sampler $\mathcal{U}$, a Sampler Under Test (SUT) $\mathcal{G}$, a tolerance parameter ϵ, an non-tolerance parameter η, and a confidence parameter δ. ϵ, η, δ take values between 0 to 1. The problem under consideration is to distinguish between the case where $\mathcal{G}$ is close to $\mathcal{U}$, and the case when $\mathcal{G}$ is far from $\mathcal{U}$. We know the probability of each assignment in the support for uniform sampler $\mathcal{U}$, that is, $Pr[\mathcal{U}(.,.,.)] = \dfrac{1}{|R_\varphi|}$. However, distribution for $\mathcal{G}$ is unknown, we only have access to samples from $\mathcal{G}$. Given access to a uniform sampler $\mathcal{U}$, Barbarik provides guarantees described in Definition 2. Furthermore, in case Barbarik rejects the SUT, it also provides a CNF formula $\hat{\varphi}$ as a witness. The formula φ is reduced to $\hat{\varphi}$ such that $\hat{\varphi}$ has exactly two assignments for the variables in the support S, and the distribution $D_{\mathcal{G}(\hat{\varphi})}$ from which samples are generated for $\hat{\varphi}$ is η far from uniform.

To achieve the aforementioned guarantees, Barbarik uses the idea of conditional sampling. Barbarik samples a satisfying assignment σ_1 from the SUT $\mathcal{G}$, and another satisfying assignment σ_2 from the base uniform sampler $\mathcal{U}$. Let T be $\{\sigma_1, \sigma_2\}$. If the distribution $D_{\mathcal{G}(\varphi)}$ from which SUT is sampling is close to uniform distribution, then the conditional distribution $D_{\mathcal{G}(\varphi)|T}$ is also close to uniform distribution. Similarly, if the distribution $D_{\mathcal{G}(\varphi)}$ is far from uniform distribution, then the conditional distribution $D_{\mathcal{G}(\varphi)|T}$ is also far from uniform distribution. Therefore, instead of focusing on the distribution $D_{\mathcal{G}(\varphi)}$, Barbarik considers the distribution $D_{\mathcal{G}(\varphi)|T}$ as it is easier to test.

In order to consider the distribution $D_{\mathcal{G}(\varphi)|T}$, Barbarik constructs a formula $\hat{\varphi}$ from φ with the help of the subroutine Kernel. The subroutine Kernel takes a formula φ, two satisfying assignments σ_1 and σ_2, and an integer N which represents the number of assignments $\hat{\varphi}$ and returns a formula $\hat{\varphi}$. The subroutine Kernel ensures that $\hat{\varphi}$ and φ have the similar structure, and $Supp(\varphi) \subset Supp(\hat{\varphi})$. Furthermore, $|\forall \sigma \in R_{\hat{\varphi}} | \sigma_{\downarrow S} = \sigma_1| = |\forall \sigma \in R_{\hat{\varphi}} | \sigma_{\downarrow S} = \sigma_2|$.

The formula $\hat{\varphi}$ should satisfy the two conditions: (i) If the SUT $\mathcal{G}(\varphi)$ is ϵ-additive almost-uniform generator, the distribution from which sampler is generating samples, say $D_{\mathcal{G}(\hat{\varphi},S)}$ is *close* to uniform distribution over the set $\{\sigma_1, \sigma_2\}$, and (ii) If the SUT $\mathcal{G}(\varphi)$ is η-far from uniform sampler, then the distribution $\mathcal{U}$, the distribution $D_{\mathcal{G}(\hat{\varphi},S)}$ is *far* from uniform distribution over the set $\{\sigma_1, \sigma_2\}$.

If the sampler $\mathcal{G}$ is an additive almost-uniform generator on any input formula φ, the first condition would be satisfied. However, to satisfy the second condition, we need *subquery-consistent assumption* as per [3]:

▶ **Definition 3.** *The* ***subquery-consistent sampler assumption*** *states that if* $(\hat{\varphi}, \hat{S})$ *is the output obtained from* Kernel$(\varphi, S, \sigma_1, \sigma_2, N)$ *then*

- $S \subseteq \hat{S}$
- *the output of* $\mathcal{G}(\hat{\varphi}, S, N)$ *is* N *independent samples from the conditional distribution* $\mathcal{D}_{\mathcal{G}(\varphi,S)}|_T$*, where* $T = \{\sigma_1, \sigma_2\}$.

Thus, if for any formula φ the sampler $\mathcal{G}(\varphi)$ is η-far from the uniform sampler in the ℓ_1 distance and the sampler satisfies the **subquery-consistent sampler assumption** then Barbarik will Reject with probability $(1 - \delta)$.

3 A Quantitative Tester

The behavior of Barbarik shows that while Barbarik is able to return Reject for samplers without guarantees such as STS or QuickSampler, it returns Accept for CMSGen. It is important to note that the theoretical analysis of soundness of Barbarik is unconditional but the analysis of completeness is conditional, i.e., when Barbarik returns Reject, then the sampler is non-uniform, but the output Accept from Barbarik needs to be interpreted through the lens of *subquery-consistent* assumption.

It is worth emphasizing that the existence of strong lower bounds on the black-box approach necessitates introduction of a grey-box approach, and in turn subroutines such as Kernel along with *subquery-consistent* assumption are likely unavoidable. Therefore, in order to improve Barbarik, we focus on extending Kernel via parameterization to allow a nuanced analysis of the quality of distributions. To this end, we first focus on identifying properties of formulas that may make it hard for algorithms without rigorous guarantees to sample well.

3.1 Computational Hardness

As discussed in Section 1, there are a number of decisions taken by CMSGen, as in all samplers and solvers, for increasing efficiency. Many of these decisions/heuristics are inherited from CryptoMiniSat. One of the crucial components of CDCL-based SAT solvers is the usage of restarts [2]. While theoretical understanding of the power and need for restarts in CDCL SAT solvers is limited, a predominant view among practitioners is that frequent restarts help the solver avoid being stuck in a part of assignment space.

The usage of heuristics that seek to avoid a sampler being stuck in a part of assignment space may have implications on its ability to sample uniformly. In particular, one can argue that usage of frequent restarts may lead CMSGen to not sample uniformly for a certain class of formulas, where the solution space of the formula can be categorized into *easy* and *hard* – such that solutions belong to the *easy* set are easier to find without the need for excessively large number of conflicts while the solutions belonging to the *hard* set require significantly more conflicts. In such a scenario, CMSGen may find it harder to sample uniformly as the restarts will push CMSGen towards the easier side while it may almost never end up finding an assignment from the harder side. At this point, one may ask if this observation can be used to inform the design of the sampler tester.

To design a test framework to Reject a sampler such as CMSGen, we need to formalize our observation. To this end, we seek to define the notion of *computational hardness* for our case formally. At the onset, it is worth accepting that our limited understanding of the workings of CDCL solvers in the context of classical complexity-theoretic notions imply that we need to use constructs based on practical aspects of SAT solvers. Roughly speaking, the computational hardness of a CNF-formula should indicate how hard it is for a SAT solver to find a satisfying assignment. It is well known that while modern SAT solvers are extremely efficient at solving many problems, there are entire classes of problems that pose significant challenges. One such class of problem is cryptographic challenges, which are designed to be hard to be solved by any tool. The consumption of resources such as a memory by an algorithm varies with time, and we seek to capture the peak resource consumption as follows:

▶ **Definition 4.** *Given an algorithm $\mathcal{A}$, input $\mathcal{I}$, and time t, for a particular run of the algorithm $\mathcal{A}$ on input $\mathcal{I}$ the* $\mathsf{PeakCost}(\mathcal{A}, \mathcal{I}, t)$ *measures the maximum resource consumption by $\mathcal{A}$ at execution step t on that particular run. This function is a non-decreasing function in t and stops to increase from the moment the run of the algorithm stops.*

Given a set of solvers/samplers $\mathcal{G}$, a CNF-formula φ is said to have computational hardness κ with respect to $\mathcal{G}$ if for $\mathcal{A} \in \mathcal{G}$

$$\Pr[\lim_{t\to\infty}\{\mathsf{PeakCost}(\mathcal{A}, \varphi, t)\} \geq \kappa] \geq 1 - o(1),$$

where the probability is taken over the internal randomness of $\mathcal{A}$, and $o(1)$ refers to "little-o" notation.

To capture the behavior of samplers that employ cutoff parameters, we define the notion of intractable formulas for cutoff κ as the set of formulas whose computational hardness is at least κ with respect to $\mathcal{G}$, i.e.,

▶ **Definition 5.** $\mathsf{Intractable}(\kappa, \mathcal{G}) = \{\varphi \mid \varphi \textit{ has computational hardness} \geq \kappa \textit{ w. r. t. } \mathcal{G}\}$

In the next section, we seek to use the notion of $\mathsf{Intractable}(\kappa, \mathcal{G})$ to improve Barbarik.

3.2 From Kernel to Shakuni

In this section, we turn to the design of an improved version of Barbarik, called ScalBarbarik, that can employ the set of formulas belonging to Intractable so as to distinguish samplers that were beyond the reach of Barbarik. ScalBarbarik takes as input an SUT $\mathcal{G}$, a uniform sampler $\mathcal{U}$, tolerance parameter ε, intolerance parameter η, accuracy parameter δ, a CNF-formula φ, a set $S \subseteq \mathit{Supp}(\varphi)$ and a computational hardness parameter κ. It outputs Accept or Reject depending on whether the SUT is ε-additive close to a uniform sampler or whether it is η-far from the uniform sampler. It is supposed to output the correct answer with probability at least $(1 - \delta)$. The computational hardness parameter is passed onto the subroutine Shakuni.

Shakuni takes in a CNF-formula φ, a set $S \subseteq \mathit{Supp}(\varphi)$, two assignments σ_1 and σ_2 from $\mathit{sol}(\varphi)_{\downarrow S}$ and a positive integer N and returns a new formula $\hat{\varphi}$ such that the following conditions are satisfied:

- $\hat{\varphi}$ has at least N satisfying assignments
- Every satisfying assignment of $\hat{\varphi}$ restricted to the set S is either σ_1 or σ_2
- If R_{σ_1} and R_{σ_2} are the set of assignments of $\hat{\varphi}$ that when restricted to the set S is σ_1 and σ_2 respectively, then $|R_{\sigma_1}| = |R_{\sigma_2}|$

In contrast to Kernel [3], Shakuni constructs $\hat{\varphi}$ such that the set R_{σ_1} is significantly different from the set R_{σ_2} in a structure such that finding assignments from one is easier than finding assignments from the other. More precisely, Shakuni assumes access to a subroutine GenHard that takes in the computational hardness parameter κ and estimated count parameter τ as inputs and returns $(\psi, \hat{\tau})$ such that $\hat{\tau} = |\mathit{sol}(\psi)|$ and $\psi \in \mathsf{Intractable}(\kappa, \mathsf{C}_{\mathsf{CDCL}})$ where $\mathsf{C}_{\mathsf{CDCL}}$ refers to the set of all the *efficient* CDCL solvers. As discussed above, given our lack of understanding of CDCL solvers, we do not seek to define $\mathsf{C}_{\mathsf{CDCL}}$ formally, but we discuss the approach to construct formulas that seem to exhibit desired properties in practice in Section 3.3.

Assuming existence to GenHard, Shakuni starts by first finding a formula ψ with computational hardness parameter κ. Then, it uses ψ to construct the CNF-formula $\hat{\varphi}$ such that the assignments in R_{σ_1} correspond to solutions of ψ while the assignments of R_{σ_2} corresponds to solutions of a Chain Formula obtained according to [4] and having a much smaller computational hardness measure.

3.3 Formulas with Computational Hardness Measure

As discussed above, Shakuni (and in turn, ScalBarbarik) assumes access to a subroutine GenHard that takes in a counting parameter τ and hardness parameter κ and returns a formula $(\psi, \hat{\tau})$ such that (1) $|sol(\psi)| = \hat{\tau}$, where $\hat{\tau} \approx \tau$, and (2) the hardness of finding a solution of ψ using a CDCL-based SAT solver is proportional to κ.

To this end, we employ the construct of cryptographic hash functions, widely studied in cryptography. A cryptographic hash family, $\mathcal{H}_{crypto} := \{h : \{0,1\}^* \to \{0,1\}^m\}$ is a family of hash functions that compute a fixed-length hash value, also known as *fingerprint*, for arbitrarily long message msg. In the context of this work, we are interested in a collection of such families, $\{\mathcal{H}^1_{crypto}, \mathcal{H}^2_{crypto}, \ldots, \mathcal{H}^\kappa_{crypto} \ldots\}$ that satisfy the following two properties:

Pre-Image Resistance. For all $h \in \mathcal{H}^\kappa_{crypto}$, given y, the computational hardness of the task of finding msg such that $h(\mathsf{msg}) = y$ is a monotonically non-decreasing function of the hardness[3] parameter κ [11]. In our context, we are interested in the hardness measured as runtime of a CDCL SAT solver to find msg such that $h(\mathsf{msg}) = y$.

(Weak) Collision Resistance. For $x, y \in \{0,1\}^*$ we have $\Pr[h(x) = h(y)] \approx \frac{1}{2^m}$, where probability is defined over random choice of x and y.

The understanding in the cryptographic community is that most of the widely used hash families satisfy the above properties. In this work, we work with one of the widely studied hash families, SHA-1, whose hardness parameter can be varied by changing the number of so-called *rounds* of the algorithm [14]. We exploit the above properties of SHA-1 to be able to generate formulas that are similar but have tunable complexity and number of solutions. We use the SHA-1 preimage CNF instance generator[4] by Nossum [20], which generates the function $\mathcal{H}_{\text{SHA-1}} := \{h : \{0,1\}^{512} \mapsto \{0,1\}^{160}\}$. The generator allows us to set any number of randomly fixed input bits, any number of output bits, and to vary the number of rounds κ. For example, using 10 rounds, fixing 0 bits of input and 160 bits of output, the generator takes a random 512 bits input msg, runs SHA-1 on msg to obtain y, then generates a formula to encode the problem $h^{-1}(y)$, where $h \in \mathcal{H}^{10}_{\text{SHA-1}}$.

We need to construct a formula ψ with predefined number of satisfying assignments. Therefore, in order to be able to decide the number of satisfying assignments of the generated formula, and to have adjustable complexity, we change the problem slightly. We consider a random 512 bits input, msg, and we calculate $y = h_\kappa(\mathsf{msg})$, where κ is the number of rounds. We generate the formula ψ using the generator as above, encoding the function $y = h_\kappa(\mathsf{msg})$. We then fix the first e bits of msg and the first f bits of y in ψ. Hence, our formula has the following parameters: κ, e, f. We use these parameters to allow us to generate any number of problems of approximate complexity and of approximate number of solutions.

[3] A formal characterization from complexity theoretic viewpoint along with the standard cryptographic assumptions is beyond the scope of this work.

[4] Available at `https://github.com/vegard/sha1-sat`

Generating hard problems with multiple solutions

Due to the collision resistance effect, with $\kappa = 80, e = 500, f = 160$, it is most likely that there is only one solution to the generated formula: there are only 12 bits to vary for msg and there is at least one solution given the way the problem is generated. Checking the actual number of solution is easy given an optimized SHA-1 implementation, as it only needs 2^{12} executions of SHA-1. Now, to create a formula with multiple solutions, let us consider the parameters $\kappa = 80, e = 500, f = 0$. Here, there are almost certainly 2^{12} solutions, as any lower than 2^{12} would mean a collision on SHA-1, which is extremely unlikely. However, this formula is very easy to solve, as any of the 12 bits can be varied and a solution obtained.

Putting the above two cases together, one might use the parameters $\kappa = 80, e = 500, f = 5$ to get the number of solutions to be approximately $s = 2^{512-e-f} = 2^7$. There are 12 bits that are unset in the input and there are 5 bits set in the output, leading to a difference of 7 bits combined with the weak collision effect, leads to approximate 2^7 solutions. If we generate with the same parameters but $f = 6$ the number of solutions halves, and the complexity of finding a solution approximately doubles, as now there is one more fingerprint bit that must match. To change the complexity with a finer grain than doubling or halving it, one can also change the number of rounds, κ. Therefore, we can vary κ, e and f to generate a formula ψ with varying complexity that can have solution $\hat{\tau}$, where $\hat{\tau}$ approximate the τ.

4 Algorithmic Description

We augment Barbarik with Shakuni to obtain ScalBarbarik. We now provide the detailed algorithm description of Shakuni.

Algorithm 1 presents the pseudocode of the Shakuni subroutine. Shakuni takes a formula φ, two satisfying assignments of φ, σ_1 and σ_2, the desired number of samples τ, and the hardness parameter κ. Shakuni assumes access to following two subroutines:

- GenHard: Takes a counting parameter τ and hardness parameter κ and returns a formula $(\psi, \hat{\tau})$.
- ConstructChain: Takes $\hat{\tau}$ and variables of ψ as input and constructs a chain formula $\hat{\psi}$ as discussed in Section 2.

Shakuni first finds a lit that is the first literal that appears in σ_1, but not in σ_2. On line 2, Shakuni conditions the formula φ over σ_1 and σ_2, and considers the new formula as φ'. Then, on line 3, Shakuni calls GenHard subroutine with τ and κ. GenHard returns a formula ψ and $\hat{\tau}$. On lines 4 and 5 Shakuni constructs the formula $\hat{\varphi}$. $\hat{\varphi}$ is the formula φ' conjuncted with positive literal lit implies ψ, and literal $\neg$lit implies the formula returned by ConstructChain. Finally, Shakuni adds the variables of $\hat{\varphi}$ in S, and stores them as $\hat{S}$ on line 6. Finally, Shakuni returns the formula $\hat{\varphi}$ and $\hat{S}$.

As discussed, Shakuni assumes access to the subroutine GenHard. Algorithm 2 presents GenHard. GenHard takes a integer τ, and a hardness parameter κ as inputs. GenHard further assumes access to following two subroutines:

- Compute: Takes an integer τ and returns two positive integers m and f such that $m - f$ is equal to $\lceil \log \tau \rceil$.
- NossumFormulaGen: Takes the SHA-1 number of rounds κ, integers m and f, and strings over $\{0, 1\}$ M and F. It considers a random 512 bits msg and fixes the first m bits of msg to M. It runs SHA-1 with κ rounds on msg to obtain y, whose first f bits are fixed to F. NossumFormulaGen returns a formula ψ which encodes the problem $h_{\kappa}^{-1}(y)$.

Algorithm 1 Shakuni($\varphi, S, \sigma_1, \sigma_2, \tau, \kappa$).

```
lit ← $(\sigma_1 \setminus \sigma_2)[0]$    /* Choose first literal lit s.t. lit $\in \sigma_1$, and lit $\notin \sigma_2$ */
$\varphi' = \varphi \wedge (\sigma_1 \vee \sigma_2)$
$(\psi, \hat{\tau}) \leftarrow$ GenHard$(\tau, \kappa)$
$\hat{\varphi} \leftarrow \varphi' \wedge$ (lit $\rightarrow \psi$)
$\hat{\varphi} \leftarrow \hat{\varphi} \wedge$ ($\neg$lit $\rightarrow$ ConstructChain$(\hat{\tau}, Supp(\psi))$)
$\hat{S} \leftarrow S \cup Supp(\hat{\varphi})$
return $(\hat{\varphi}, \hat{S})$.
```

Algorithm 2 GenHard(τ, κ).

```
1  $(m, f) \leftarrow$ Compute$(\tau)$  /* Compute $m$, $f$ such that $m, f \geq 0, m - f = \lceil \log \tau \rceil$ */
2  $M \leftarrow_r \{0,1\}^{512-m}$
3  $F \leftarrow_r \{0,1\}^f$
4  $\psi \leftarrow$ NossumFormulaGen$(\kappa, m, M, f, F)$
5  $\hat{\tau} \leftarrow 0$
6  for $value \in \{0,1\}^m$ do
7      if $h_\kappa(M + value)[1:f] = F$        /* $h_\kappa$ is a hash-function, $h_\kappa \in \mathcal{H}^{\kappa}_{\texttt{SHA-1}}$ */
8      then
9          $\hat{\tau} \leftarrow \hat{\tau} + 1$
10 return $\psi, \hat{\tau}$
```

GenHard first computes the value of m and f by calling subroutine Compute. On line 2 GenHard generates a random string M of length $512 - m$ over $\{0,1\}$ from all possible sets of such strings. Similarly, on line 3, GenHard generates a string F of length f over $\{0,1\}$ randomly from all possible such strings. On line 4, GenHard calls NossumFormulaGen subroutine that returns a formula ψ. Finally, to calculate the exact number of satisfying assignments of ψ, on lines 6-9, GenHard iterates over all possible strings, denoted as *value*, of $\{0,1\}$ of size m. If first f bits of $h_\kappa(M + value)$ matches with F, then the count of $\hat{\tau}$ is increased by 1. At the end, GenHard returns the formula ψ and $\hat{\tau}$.

The algorithmic description of ScalBarbarik is almost identical to the Barbarik except for a notable difference of replacement of Kernel subroutine with Shakuni and the argument of hardRange. For completeness, we now provide the detailed algorithmic description of ScalBarbarik in Algorithm 3. Note that the expressions for t_j, β_j, N_j in Algorithm 3 have been revised after fixing minor errors in [3].

Algorithm 3 represents ScalBarbarik. ScalBarbarik has three loops, the outermost loop, lines 2-19 varies the computational hardness parameter κ as per the given range. The second loop, lines 3-19 makes $\log(\dfrac{4}{2\varepsilon + \eta})$ many rounds. And, in each round, first ScalBarbarik on line 9 computes the number of satisfying assignments, called N_j to be sampled from SUT. The inner loop of ScalBarbarik iterates t_j many times, which is computed in each round on line 4. In the inner loop, ScalBarbarik first samples a satisfying assignment σ_1 from the ideal distribution using the base sampler on line 12, and then it samples a satisfying assignment σ_2 from the SUT on line 13. Then, on line 14 ScalBarbarik calls subroutine Shakuni with formula φ, sampling set S, σ_1, and σ_2,N_j and κ. Subroutine Shakuni returns a new formula $\hat{\varphi}$ and a sampling set $\hat{S}$. On line 15, ScalBarbarik asks the SUT to sample N_j many satisfying assignments of $\hat{\varphi}$, which is stored in list L_3. On line 16, ScalBarbarik calls subroutine *Bias*.

Algorithm 3 ScalBarbarik($\mathcal{G}$, $\mathcal{U}$, S, ε, η, δ, φ, hardRange).

```
1  $S \leftarrow Supp(\varphi)$
2  for $\kappa \in$ hardRange do
3      for $j \leftarrow 1$ to $\lceil \log(\frac{4}{2\varepsilon+\eta}) \rceil$ do
           /* constants required to compute the # of samples */
4          $t_j \leftarrow \lceil 2^j \frac{(\eta+2\varepsilon)}{(\eta-2\varepsilon)^2} \log(4(2\varepsilon+\eta)^{-1})(\frac{4e}{(e-1)}) \ln(\delta^{-1}) \rceil$
5          $\beta_j \leftarrow \frac{(2^{j-1}+1)(2\varepsilon+\eta)}{4+(2\varepsilon+\eta)(2^{j-1}-1)}$
6          BoundFactor $\leftarrow \log\left(\frac{2^4 e}{e-1} \frac{\delta^{-1}}{(\eta-2\varepsilon)^2} \log(\frac{4}{2\varepsilon+\eta}) \ln(\frac{1}{\delta})\right)\rceil$
7          $\gamma \leftarrow \frac{(\beta_j - 2 \times \varepsilon)}{4}$
8          ConstantFactor $\leftarrow \left\lceil \frac{1}{(8.79 \times \gamma \times \gamma)} \right\rceil$
9          $N_j \leftarrow \lceil$(ConstantFactor $\times$ BoundFactor )$\rceil$
10         for $i \leftarrow 1$ to $t_j$ do
11             while $L_1 = L_2$ do
12                 $L_1 \leftarrow \mathcal{G}(\varphi, S, 1)$; $\sigma_1 \leftarrow L_1[0]$     /* $\mathcal{G}$ samples $\sigma_1$ from Sol($\varphi$) */
13                 $L_2 \leftarrow \mathcal{U}(\varphi, S, 1)$; $\sigma_2 \leftarrow L_2[0]$     /* $\mathcal{U}$ samples $\sigma_2$ from Sol($\varphi$) */
14             $(\hat{\varphi}, \hat{S}) \leftarrow$ Shakuni$(\varphi, S, \sigma_1, \sigma_2, N_j, \kappa)$
15             $L_3 \leftarrow \mathcal{G}(\hat{\varphi}, S, N_j)$     /* $\mathcal{G}$ samples $N_j$ solutions from Sol($\hat{\varphi}$) */
16             $b \leftarrow Bias(\sigma_1, L_3, S)$
17             if $b < \frac{1}{2}(1 - c_j)$ or $b > \frac{1}{2}(1 + c_j)$ then
18                 return REJECT
19         return ACCEPT
```

The subroutine *Bias* takes σ_1, L_3, and S as input and returns the cardinality of intersection of the σ_1 and L_3 over the sampling set S. The returned cardinality from *Bias* is stored in b. Finally, ScalBarbarik checks if the value of b is either lower than the low threshold or higher than the high threshold on line 17. If that is the case, ScalBarbarik rejects the SUT on line 18, otherwise, it continues with the inner loop on line 10.

4.1 Theoretical Analysis

First we need to prove the correctness of GenHard. From the code of GenHard (also, refer to Section 3.3) the following theorem follows:

▶ **Theorem 6.** *If* GenHard(τ, κ) *returns* $\psi, \hat{\tau}$ *then* $|R_\psi| = \hat{\tau}$*, where* $\hat{\tau} \geq \tau$*.*

Now, the correctness of Shakuni is almost identical to that of Kernel from [3]. We can prove the following theorem:

▶ **Theorem 7.** *If* $\hat{\varphi}$ *is the output of* Shakuni$(\varphi, S, \sigma_1, \sigma_2, \tau, \kappa)$ *then* $R_{\hat{\varphi}}$ *can be written as a disjoint union of two sets* Z_1 *and* Z_2 *such that for* $|Z_1| = |Z_2|$ *and for all* $\sigma \in Z_1$*,* $\sigma|_S = \sigma_1$ *and for all* $\sigma \in Z_2$*,* $\sigma|_S = \sigma_2$*.*

Proof. On line 2 it is ensured that φ' has only two satisfying assignments – namely σ_1 and σ_2. From Theorem 6 we see that GenHard (on line 3) returns a formula $(\psi, \hat{\tau})$ where $R_\psi = \hat{\tau}$ and at the same time ConstructChain (on line 5) returns a formula $\hat{\psi}$ with $R_{\hat{\psi}} = \hat{\tau}$ and

$Supp(\psi) = Supp(\hat{\psi})$. Thus by the construction of $\hat{\varphi}$ on lines 4 and 5, if σ is a satisfying assignment of $\hat{\varphi}$ then firstly $\sigma|_S$ is either σ_1 or σ_2. Also if $\sigma|_S$ is σ_1 then $\sigma|_{Supp(\hat{\varphi}\setminus S}$ is a satisfying assignment of ψ. Moreover, there is a one-to-one correspondence between the satisfying assignments of $\hat{\varphi}$, that satisfy $\sigma|_S = \sigma_1$, with R_ψ. Similarly, if $\sigma|_S$ is σ_2 then $\sigma|_{Supp(\hat{\varphi}\setminus S}$ is a satisfying assignment of $\hat{\psi}$. and there is a one-to-one correspondence between the satisfying assignments of $\hat{\varphi}$, that satisfy $\sigma|_S = \sigma_2$, with $R_{\psi'}$. Thus we have the theorem. ◀

Given the correctness of Shakuni, we observe that the theoretical analysis and query complexity of ScalBarbarik are almost identical to that of Barbarik from [3]. That is, if SUT $\mathcal{G}$ is ε-additive close to the uniform sampler then with probability $(1-\delta)$, ScalBarbarik outputs Accept. If the SUT is η far from uniform and it abides by the *subquery-consistent assumption*, ScalBarbarik outputs Reject with probability $(1-\delta)$. In case ScalBarbarik outputs Reject for sampler $\mathcal{G}$ on input φ, the assignments σ_1 and σ_2 can be seen as a certificate because the sampler $\mathcal{G}$ samples them with significantly different probabilities. Therefore, the output of ScalBarbarik is a list of tuples of the values of κ and the corresponding output.

5 Experimental Evaluation

To analyze the behavior of ScalBarbarik, we built a prototype implementation in Python and performed empirical evaluation on the 50 benchmarks that were used for the evaluation of Barbarik so as to situate our results with prior context [3]. For our evaluation, we used SPUR [1] as a base uniform sampler.

Test Hardware. All our experiments were conducted on a high-performance computing cluster with each node consisting of a E5-2690 v3 CPU with 24 cores and 96GB of RAM, with a memory limit of 4GB/core.

Test Parameters. We considered tolerance parameter ϵ, intolerance parameter η, and confidence δ to be 0.2, 1.6, and 0.1, respectively for experimentation evaluation using ScalBarbarik. For our chosen parameters, the number of samples required to return Accept for a given SUT is 2.173×10^3. We considered the following hardness parameters for ScalBarbarik: $\kappa = 10$, 11, 12, and 13. In the implementation of GenHard, we used $m = 14, f = 4$.

Samplers Tested. We performed empirical evaluation with four state-of-the-art samplers, QuickSampler [8], STS [9] CMSGen [13], and UniGen3 [23]. Of these, STS, QuickSampler, and CMSGen cannot provide theoretical guarantees on the distribution of samples generated, whereas UniGen provides guarantees. Furthermore, we experimented with different restart intervals for CMSGen. We set the parameter restart intervals to 300 and 500, that is, restarts at every 300 or 500 conflicts. We used CMSGen$_{300}$ and CMSGen$_{500}$ to refer to our prototype of CMSGen, respectively. The default version of CMSGen restarts at 100 conflicts.

The objective of our experimental evaluation is to analyze the impact of different computational hardness levels on the ability of ScalBarbarik to distinguish between state-of-the-art samplers. Furthermore, we seek to use ScalBarbarik to establish the balance between scalability and uniformity in order to tune the sampler to the application at hand. Towards this, we analyses the impact of different restart intervals of CMSGen on the quality of samples generated through ScalBarbarik. In particular, we seek to answer the following questions:

(1) Can ScalBarbarik distinguish between distributions generated by the various state-of-the-samplers? (2) Can we use ScalBarbarik to design a sampler that can balance scalability and uniformity?

Summary of results. In summary, we observe that ScalBarbarik Rejects STS and QuickSampler, and returns Accept for UniGen for all the benchmarks. Moreover, as we increase the hardness parameter for ScalBarbarik, it could Reject CMSGen. These experiments show that the quality of distribution for the samples generated by CMSGen is between the distribution generated by samplers without guarantees such as STS and QuickSampler, and by samplers with guarantees, such as UniGen3. Furthermore, with the help of ScalBarbarik, we are able to show that the quality of distribution generated improves with increased restart intervals for CMSGen, however, it takes more time to generate the samples.

5.1 Performance of ScalBarbarik

The first column of Table 1 shows the value of κ and rest of the table consists of two columns for each of the samplers. The columns with Accept/Reject represent the number of instances for which ScalBarbarik outputs Accept or Reject, respectively.

Table 1 Analysis of different samplers with ScalBarbarik over 50 benchmarks. Parameters used were $\epsilon : 0.2, \eta : 1.6, \delta : 0.1$, and samples required to output Accept: 2.173×10^3.

ScalBarbarik	QuickSampler		STS		CMSGen		UniGen3	
(κ)	Accept	Reject	Accept	Reject	Accept	Reject	Accept	Reject
10	0	50	0	50	50	0	50	0
11	0	50	0	50	41	9	50	0
12	0	50	0	50	19	31	50	0
13	0	50	0	50	0	50	50	0

Note that for $\kappa = 10$, ScalBarbarik outputs Accept on all instances for CMSGen, whereas it Rejects QuickSampler and STS. Upon increasing the value of κ to 11 and 12, ScalBarbarik outputs Reject on 9 and 31 instances, respectively. Finally, ScalBarbarik outputs Reject for CMSGen on all 50 instances with $\kappa = 13$. On the other hand, ScalBarbarik outputs Accept for all values of κ on all instances for UniGen3.

It is worth emphasizing that in comparison to Barbarik, ScalBarbarik returns a fine-grained analysis of the quality of distributions generated by the given sampler. Such a fine-grained analysis allows one to observe that the quality of distributions generated by CMSGen lie between QuickSampler, STS and UniGen3.

5.2 Achieving Balance between Scalability and Uniformity

Based on the discussion in Section 3.1, we can hypothesize that the quality of samples produced increase with an increase in restart interval for SAT solver based sampler such as CMSGen. To put our hypothesis to test, and to understand the behavior of CMSGen with different restart intervals, we performed evaluation using ScalBarbarik on CMSGen, CMSGen_{300}, and CMSGen_{500}. To provide a prospective, we also considered a sampler with theoretical guarantees, UniGen. We set the computation hardness parameter $\kappa = 11$, 15, 18, and 22. In Table 2, we list the number of instances for which ScalBarbarik returned Accept and Reject corresponding to the aforementioned samplers.

Table 2 # of benchmarks for which CMSGen, CMSGen$_{300}$, CMSGen$_{500}$, and UniGen3 are Accepted or Rejected by ScalBarbarik. Total of 50 benchmarks. Parameters $\epsilon : 0.2, \eta : 1.6, \delta : 0.1$, and samples required to return Accept 2.173×10^3. The default version of CMSGen used restart at 100 conflicts.

ScalBarbarik	CMSGen		CMSGen$_{300}$		CMSGen$_{500}$		UniGen3	
(κ)	Accept	Reject	Accept	Reject	Accept	Reject	Accept	Reject
11	41	9	47	3	47	3	50	0
15	0	50	37	13	42	8	50	0
18	0	50	0	50	36	14	50	0
22	0	50	0	50	0	50	50	0

Table 3 CMSGen, CMSGen$_{300}$, CMSGen$_{500}$ with ScalBarbarik with different hardness parameters.

(κ)	Benchmarks	CMSGen		CMSGen$_{300}$		CMSGen$_{500}$	
		Result	Samples	Result	Samples	Result	Samples
15	GuidanceService	Reject	742	Accept	2.173×10^3	Accept	2.173×10^3
	70.sk-310	Reject	265	Accept	2.173×10^3	Accept	2.173×10^3
	BlastedSpring24	Reject	318	Accept	2.173×10^3	Accept	2.173×10^3
	ActivityService	Reject	106	Reject	848	Accept	2.173×10^3
	IterationService	Reject	265	Reject	742	Reject	1.802×10^3
18	GuidanceService	Reject	159	Reject	265	Accept	2.173×10^3
	70.sk-310	Reject	53	Reject	848	Accept	2.173×10^3
	BlastedSpring24	Reject	159	Reject	742	Reject	849
	ActivityService	Reject	106	Reject	689	Accept	2.173×10^3
	IterationService	Reject	53	Reject	265	Reject	1.961×10^3

We observe that ScalBarbarik needs to increase the computation hardness in order to Reject CMSGen$_{500}$ for all the benchmarks – it Rejects CMSGen, CMSGen$_{300}$, and CMSGen$_{500}$ at κ values 13, 18, and 22 respectively.

Table 3 presents the result of ScalBarbarik with κ set to 15 and 18 over a subset of representative benchmarks. The first column in Table 3 presents the hardness parameter κ used with ScalBarbarik. The second column has the benchmarks details and the following columns indicate the outcome of ScalBarbarik for samplers CMSGen, CMSGen$_{300}$ and CMSGen$_{500}$. There are two columns for each of the samplers: (i) the first column shows whether the sampler is accepted by ScalBarbarik as a uniform sampler, and (ii) the second column shows the number of samples required by ScalBarbarik to decide Accept/Reject. Table 3 shows that ScalBarbarik needs less samples to reject CMSGen as compared to CMSGen$_{300}$ and CMSGen$_{500}$. Furthermore, as the hardness parameter κ is increased, ScalBarbarik rejects more instances with less number of samples for all three SUTs.

The results in Table 2 and Table 3 strongly support that as we increase the restart intervals, the distribution of samples generated are more likely to be uniform.

At this point, one may wonder whether there are costs associated with the improved quality of sampling in terms of runtime efficiency. To this end, we conducted a study of runtimes over 70 benchmarks used in prior studies [13]. We present the runtime comparison of CMSGen, CMSGen$_{300}$, and CMSGen$_{500}$ to generate 1000 samples in Figure 1. To put the runtimes in perspective, we also plot the curve corresponding to UniGen3. Figure 1 represents a cactus plot – a point $\langle x, y \rangle$ represents that a sampler took less than or equal to y seconds to sample 1000 satisfying assignments for x many benchmarks. With a timeout of 7200 seconds, CMSGen, CMSGen$_{300}$, CMSGen$_{500}$, were all able to generate 1000 samples for 52 benchmarks, and we see a significant increase in the runtime for those instances with CMSGen$_{500}$ and CMSGen$_{300}$ as compared to CMSGen.

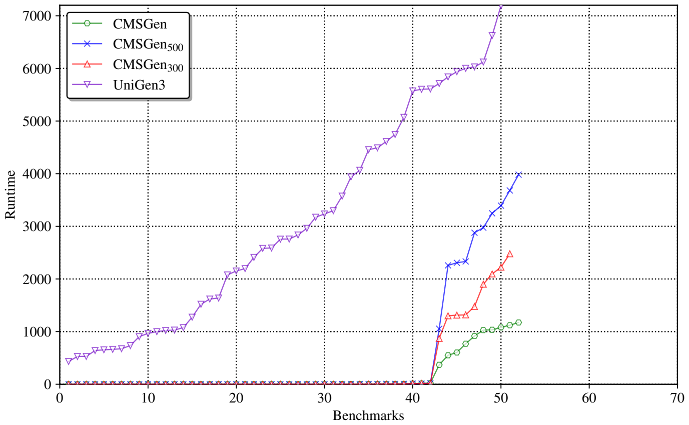


Figure 1 Cactusplot showing runtime performance of CMSGen, CMSGen$_{300}$, CMSGen$_{500}$, and UniGen3 to generate 1000 samples within a timeout of 7200s.

The gain of uniformity at the loss of runtime efficiency in the case of CMSGen$_{500}$ illustrates the trade-off between uniformity and runtime performance, and highlights opportunities for design of large number of samplers based on the needs of the underlying applications. While ideally, one would perform in-depth theoretical analysis to characterize the distribution generated by different samplers, modern CDCL solvers have not been shown to be amenable to such analysis. In this regard, having access to test frameworks such as ScalBarbarik to test uniformity is crucial.

6 Conclusion

Uniform sampling is a fundamental problem in computer science with widespread applications. This variety of applications has led to the design of many samplers with varying theoretical guarantees. There exists many uniform-like samplers that do not provide any guarantees on the distribution from which the samples are generated. The existence of such samplers led to the design of the first tester, Barbarik to test whether the distribution generated is ε-close or η-far from the uniform distribution. Barbarik was used in a test-driven development manner to create a uniform-like sampler CMSGen that cannot provide theoretical guarantees on the sampling distribution but is accepted as a ε-close uniform sampler by Barbarik.

The development of such a sampler led us to improve the testing framework Barbarik. In this work, we propose the sampler tester ScalBarbarik that provides quantitative answers to measure the quality of samplers, that is, it provides a hardness dial to achieve a fine-grained analysis of quality of samples. We showed that that the quality of samples generated by CMSGen are better than the other state-of-the-art samples such as STS and QuickSampler that do not provide theoretical guarantee; however, it is not as good as the samplers that provide guarantees on the distribution generated, such as UniGen3. Furthermore, the availability of ScalBarbarik can be used to achieve a balance between scalability and uniformity of samplers. We hope the demonstration of virtuosity of the cycle between testing and design will encourage other developers to design their own samplers while using ScalBarbarik as the underlying testing engine.

References

1 Dimitris Achlioptas, Zayd S Hammoudeh, and Panos Theodoropoulos. Fast sampling of perfectly uniform satisfying assignments. In *Proc. of SAT*, 2018.

2 Armin Biere and Andreas Fröhlich. Evaluating CDCL restart schemes. In *Proc. of Pragmatics of SAT 2015*, 2018.

3 Sourav Chakraborty and Kuldeep S. Meel. On testing of uniform samplers. In *Proc. of AAAI*, 2019.

4 Supratik Chakraborty, Dror Fried, Kuldeep S Meel, and Moshe Y Vardi. From weighted to unweighted model counting. In *Proc. of AAAI*, 2015.

5 Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. A scalable and nearly uniform generator of sat witnesses. In *Proc. of CAV*, 2013.

6 Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. Balancing scalability and uniformity in SAT witness generator. In *Proc. of DAC*, 2014.

7 Lori A Clarke. A program testing system. In *Proc. of ACM*, 1976.

8 Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. Efficient sampling of SAT solutions for testing. In *Proc. of ICSE*, 2018.

9 Stefano Ermon, Carla P Gomes, Ashish Sabharwal, and Bart Selman. Uniform solution sampling using a constraint solver as an oracle. In *Proc. of UAI*, 2012.

10 Vijay Ganesh, Charles W O'donnell, Mate Soos, Srinivas Devadas, Martin C Rinard, and Armando Solar-Lezama. Lynx: A programmatic SAT solver for the RNA-folding problem. In *Proc. of SAT*, 2012.

11 Oded Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

12 Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. Manthan: A data-driven approach for Boolean function synthesis. In *Proc. of CAV*, 2020.

13 Priyanka Golia, Mate Soos, Sourav Chakraborty, and Kuldeep S. Meel. Designing samplers is easy: The boon of testers. In *Proc. of FMCAD*, 2021.

14 Evgeny A. Grechnikov. Collisions for 72-step and 73-step SHA-1: improvements in the method of characteristics. *Proc. of IACR*, 2010.

15 James C King. Symbolic execution and program testing. *Comm. ACM*, 1976.

16 James H Kukula and Thomas R Shiple. Building circuits from relations. In *Proc. of CAV*, 2000.

17 S. Kullback and R. A. Leibler. On information and sufficiency. *Proc. of Ann. Math. Statist.*, 1951.

18 Kuldeep S. Meel, Yash Pote, and Sourav Chakraborty. On testing of samplers. In *Proc. of NeurIPS*, 2020.

19 Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan s Marcu, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. *Proc. of AI magazine*, 2007.

20 Vegard Nossum. SAT-based preimage attacks on SHA-1. Master's thesis, University of Oslo, 2012. URL: `https://www.duo.uio.no/handle/10852/34912`.

21 Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. Bug synthesis: Challenging bug-finding tools with deep faults. In *Proc. of ESEC/FSE*, 2018.

22 Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S Meel. Knowledge compilation meets uniform sampling. In *Proc. of LPAR*, 2018.

23 Mate Soos, Stephan Gocht, and Kuldeep S. Meel. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *Proc. of CAV*, 2020.

24 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Proc. of SAT*, 2009.

Structured Set Variable Domains in Bayesian Network Structure Learning

Fulya Trösser ✉
Université Fédérale de Toulouse, ANITI, INRAE, UR 875, 31326 Toulouse, France

Simon de Givry ✉
Université Fédérale de Toulouse, ANITI, INRAE, UR 875, 31326 Toulouse, France

George Katsirelos ✉
Université Fédérale de Toulouse, ANITI, INRAE, MIA Paris, AgroParisTech, 75231 Paris, France

Abstract

Constraint programming is a state of the art technique for learning the structure of Bayesian Networks from data (Bayesian Network Structure Learning – BNSL). However, scalability both for CP and other combinatorial optimization techniques for this problem is limited by the fact that the basic decision variables are set variables with domain sizes that may grow super polynomially with the number of random variables. Usual techniques for handling set variables in CP are not useful, as they lead to poor bounds. In this paper, we propose using decision trees as a data structure for storing sets of sets to represent set variable domains. We show that relatively simple operations are sufficient to implement all propagation and bounding algorithms, and that the use of these data structures improves scalability of a state of the art CP-based solver for BNSL.

2012 ACM Subject Classification Computing methodologies → Learning in probabilistic graphical models; Theory of computation → Discrete optimization

Keywords and phrases Combinatorial Optimization, Bayesian Networks, Decision Trees

Digital Object Identifier 10.4230/LIPIcs.CP.2022.37

Supplementary Material *Software (Source Code)*: `https://gkatsi.github.io/elsa-cp22.tar.gz`

Funding This work has been partly funded by the "Agence nationale de la Recherche" (ANR-16-CE40-0028 Demograph project and ANR-19-PIA3-0004 ANITI-DIL chair of Thomas Schiex).

Acknowledgements Thanks to the GenoToul (Toulouse, France) Bioinformatics platform for computational support.

1 Introduction

Bayesian Networks (BNs) are directed probabilistic graphical models, which can describe a normalized joint probability distribution over a potentially large set of random variables, by exploiting conditional independence to decompose the function. Learning the structure of BNs from data (the Bayesian Network Structure Learning problem, BNSL) is a challenging combinatorial optimization problem. There exist constraint-based approaches to learn BNs, which use local conditional independence tests, and score-based approaches, which use a decomposable score function to score each potential structure and aim to find the structure that minimizes this score. The former are known to be efficient, but have trouble with noisy data. The latter yield a known to be NP-hard problem [4], which additionally has proved very challenging in practice.

There exist complete methods for score-based BNSL based on dynamic programming [20], heuristic search [24, 8], maximum satisfiability [2], branch-and-cut [1] and constraint programming [22, 21]. Branch-and-cut and constraint programming have proven to be the most successful of these methods. However, scaling them up remains challenging. One challenge

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 37; pp. 37:1–37:9
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

has to do with the decomposition of the scoring functions: these assign a score to each potential set of parents of each vertex and the score of a specific structure is the sum of the scores of each parent set. This means that the objective function must have a term for each potential parent set, a potentially exponential number of terms. There are various methods by which this number is made manageable, but it is still among the greatest obstacles to scalability. Moreover, the best solvers, ILP-based GOBNILP [1], and CP-based ELSA [21] also explicitly have this set of parent sets in other parts of the model as well, in the case of ELSA as domains of variables.

Here, we propose exploiting the fact that these domains are structured, i.e., that each value is a set. Specifically, we show that we can represent potential parent sets as paths on decision trees and that using these decision trees we can answer queries more efficiently than by traversing a list of domain values. This feature has not been exploited in BNSL in the past and allows us to solve large instances more efficiently.

2 Background

2.1 Bayesian Networks

A Bayesian Network is a directed graphical model $B = \langle G, P \rangle$ where $G = \langle V, E \rangle$ is a directed acyclic graph (DAG) called the structure of B and P are its parameters. A BN describes a normalized joint probability distribution. Each vertex of the graph corresponds to a random variable and presence of an edge between two vertices denotes direct conditional dependence. Each vertex v_i is also associated with a Conditional Probability Distribution $P(v_i \mid parents(v_i))$. The CPDs are the parameters of B.

Learning a BN from a set of multivariate discrete data using the score based method uses a decomposable scoring function (such as BIC [19, 14] or BDeu [3, 12]) which assigns, based on the data, a score to each potential parent set of each vertex. The BNSL problem is the problem of finding the structure G which minimizes this scoring function.

The number of candidate parent sets can in principle be exponentially large, but it is typically kept in check. For one, the BIC scoring function [19, 14] guarantees that the number of candidate parent sets grows only logarithmically with the size of the data set. Second, there exist dedicated pruning rules [7, 6] which reduce the set further. As a last resort, an upper bound can be placed on the cardinality of parent sets. This is necessary especially in larger instances, where it is necessary to limit cardinality to as low as 3 in some cases.

2.2 CP-based BNSL

ELSA [21] is a CP-based solver for the BNSL, based on the CPBayes solver [22]. The constraint model used in ELSA has several features that we do not discuss here. Instead, we focus on the part that is relevant to our contribution. For each random variable X, there exists a corresponding CSP variable P_X whose domain is the set of candidate parent sets of X. These are unsurprisingly called parent set variables. There exists an acyclicity constraint over these which requires that their instantiation yields an acyclic graph. ELSA enforces GAC on this constraint. The central part of the GAC algorithm is algorithm 1, `acycChecker`. `acycChecker` determines in time $O(n^2d)$ whether the current set of domains admits an acyclic solution, based on the property that in any acyclic graph, for any subset of vertices C, at least one of the vertices $v \in C$ has a parent set that does not intersect C.

In addition, ELSA computes lower bounds by approximately solving the linear relaxation of the ILP (1), which was proposed by Bartlett and Cussens [1] for the GOBNILP solver.

Algorithm 1 Acyclicity checker.

```
1  acycChecker (P, D)
2  order ← {}
3  changes ← true
4  while changes do
5      changes ← false
6      foreach $v \in \mathbf{P} \setminus order$ do
8          if $\exists S \in D(v)$ s.t. $S \subseteq order$ then
10             order ← order + v
11             changes ← true
12 return order
```

$$\min \sum_{v \in \mathbf{P}, S \subseteq V \setminus \{v\}} \sigma^v(S) x_{v,S} \tag{1a}$$

$$s.t. \sum_{S \in PS(v)} x_{v,S} = 1 \qquad \forall v \in \mathbf{P} \tag{1b}$$

$$\sum_{v \in C, S \in PS^{-C}(v)} x_{v,S} \geq 1 \qquad \forall C \subseteq \mathbf{P} \tag{1c}$$

$$x_{v,S} \in \{0,1\} \qquad \forall v \in \mathbf{P}, S \in PS(v) \tag{1d}$$

This is an exponentially large ILP, but on the flip side, the constraints (1c), called *cluster constraints* are facets of the polytope [5]. Hence, following GOBNILP, ELSA starts with none of the cluster constraints in the linear relaxation and then adds only those that can improve the dual bound. This is an NP-hard problem. GOBNILP solves this NP-hard problem to find violated cluster constraints, while ELSA uses a polynomial time algorithm which can identify a strict subset of all improving cluster constraints. The central element of the algorithm used in ELSA to find cluster constraints uses algorithm 1 on the domains restricted only to values which have reduced cost 0 in the current dual solution of the linear relaxation.

Both in finding improving cluster constraints and in enforcing GAC on the acyclicity constraint, the main bottleneck is line 8 of algorithm 1, which tests whether there exists in $D(v)$ a value which is a subset of a given set. As domain sizes grow drastically faster than the number of random variables, it is crucial to optimize this step. In practical terms, even given the mitigations mentioned earlier, the average domain size can be in the thousands for larger instances.

3 Related work

A typical approach to dealing with large domain sizes in constraint programming is to enclose the set of domain values with an underestimation and an overestimation and reason with those instead. Sometimes, this can even be achieved without any loss in strength of inference. This is the case, for example, when representing only the bounds of a variables that are only used in linear inequalities. In the case where the values of a domain are sets, the variable is called a set variable. Its domain can be represented with the subset bound scheme [9], which underestimates by a set indicating all elements which appear in all remaining domain values and overestimates by a set indicating all elements which appear in any remaining domain value. The length-lex scheme uses lexicographic and cardinality information to get a tighter under- and over-estimation [10]. However, detecting infeasibility of the acyclicity constraint is crucial for the performance of CPBayes and even more for ELSA. Hence, over-estimating the actual domain in our case would lead to poor performance.

Hawkins et al. [11] followed an approach which is closer to our own, by using ROBDDs (reduced ordered binary decision diagrams) to represent domains. ROBDDs are diagrams like decision trees, but they require the same variable ordering in each branch and isomorphic subgraphs are merged, so that the underlying graph is a DAG rather than a tree. They can be significantly more compact than decision trees. However, Hawkins et al. used them in a setting where all constraints can be expressed as operations on ROBDDs. They do not deal with costs of the domain values, and in particular with reduced cost filtering.

4 Decision Trees as domain store

The set of sets that are in a domain can be seen as the set of solutions of a propositional formula, in which we have a propositional variable for each element of the universe. Therefore, knowledge compilation languages such as ROBDDs can be used to represent a domain.

There exist several queries and operations performed on the domains in ELSA, but not all are critical to optimize, as they are not performed often enough to dominate the runtime. In particular, we want to address the test in line 8, which asks whether the domain contains a set which is a subset of another given set. Therefore, the main queries that need to be supported efficiently by a domain store for our purposes are:

1. Does there exist a domain value S such that $S \subseteq T$ for some T?
2. Does there exist a domain value S with reduced cost 0 such that $S \subseteq T$ for some T?

And the main operations, which also have to support backtracking, are:

1. Pruning a single value S
2. Updating the reduced cost of a value

The main issue that disqualifies ROBDDs and other reduced representations for us is that operation 1, reduced cost filtering, may remove arbitrary values, shattering the shared suffixes that an ROBDD exploits, which means that pruning may result in increasing the size of the representation and is not even guaranteed to be in linear time. Instead, we use decision trees here, in particular binary decision trees with implied literals, inspired by a similar technique in BDDs [13]. The main use of decision trees is in machine learning for classification, but their use as a data structure for representing sets of sets (or, equivalently, a knowledge compilation language) is straightforward.

We give below definitions for the specific case of binary decision trees and binary classification, as that is all we need.

▶ **Definition 1** (Binary decision tree). *Let A be a set of features $\{a_1, \dots, a_n\}$ with Boolean domains and C_1, C_2 be two classes. A binary decision tree $\mathcal{T}$ over the features A is a directed rooted binary tree. Each internal node n of $\mathcal{T}$ is labeled with a feature $l(n) \in A$ and each arc e (of the at most two outgoing arcs) from n is labeled with $l(e) \in \{true, false\}$ and are mutually exclusive. Each leaf node t is labeled with $l(t) \in \{C_1, C_2\}$. Given an instantiation I of the features, there is a unique path from the root to a leaf t so that for each arc $e = (n, c)$ along that path, it holds that $I(l(n)) = l(e)$. We say that $\mathcal{T}$* classifies *I as $l(n)$ and that I and the path from the root to n are* consistent *with each other, or simply that I and n are consistent with each other.*

To see how we can use binary decision trees as a data structure for a set of sets, observe that we can set the features to be the variables of the indicator function of the sets in the domain and the classes as *in-set* and *not-in-set*.

This allows us to further optimize the representation. Since we only care about the *in-set* class, from now on we assume that all nodes and arcs that do not appear on a path from the root to a leaf n with $l(n) = \textit{in-set}$ are removed from the decision tree.

Additionally, we can eliminate some nodes by adding *implied literals* in each node of the tree.

▶ **Definition 2** (Binary decision tree with implied literals). *A binary decision tree with implied literals is a decision tree in which each node n (internal or leaf) is additionally labeled with a set of literals $lit(n, C_i) \subseteq \{a = v \mid a \in A, v \in \{true, false\}\}$ for $i \in \{1, 2\}$. An instantiation I is consistent with a path to a leaf t with $l(t) = C_i$ if it is consistent with all the arcs it follows and all implied literal labels $lit(n, C_i)$ for each node n on the path from the root to t.*

In our case, we abbreviate $lit(n, \textit{in-set})$ to $lit(n)$, as we ignore the class *not-in-set*. Decision trees with implied literals allow us to collapse chains, i.e., paths along which every node has outdegree 1, into a single node. Hence, they are not more compact than those without implied literals by more than a linear factor, but they have almost no overhead and, in preliminary experiments, we found them to provide some performance improvement.

In machine learning, the objective is not only to construct models that perform well on the training set, but that also generalize. Hence, it is not only acceptable, but also desirable to misclassify some samples in training sets, if that means a smaller and hence more general decision tree. In our setting, however, where we use decision trees to model a Boolean function, we accept no error. So no two sets that belong to different classes, i.e., one in *in-set* and one in *not-in-set*, are allowed to both be consistent with the same leaf node.

We place an additional constraint on the decision trees we construct, which is that each leaf node must be consistent with exactly one positive instantiation. This ensures that there exists a bijection between leaves of the tree and values in the domain. This is not as significant a constraint as it might seem at first. A leaf node n that is consistent only with positive instantiations but more than one of them is expanded into a full binary tree of depth k, where k is the number of variables (features) which have not appeared on the path from the root to n. However, for the queries that we care about, this means only that the corresponding algorithm will have to traverse an additional k nodes before answering, and, crucially, will only arrive at this point when it is guaranteed that it will give a positive answer. Even that overhead can be eliminated with some care. Indeed, while traversing the decision tree, we can determine that we have reached such a node n if the number of possible instantiations that are consistent with n is equal to the number of leaves reachable from n. The former is 2^{n-lvl}, where lvl is the distance from the root to n. The latter can be computed on construction and updated as values are removed. If these are equal, we know that the subtree contains all possible subsets and we can answer our query without more search. We give more detail later.

Constructing decision trees

Constructing a minimum decision tree is NP-hard with respect to several metrics [15]. We use the *information gain* heuristic [17] to choose which variable to branch on in each node. It is a natural side effect of computing the information gain that we learn how many of the sets that are consistent with a node n contain the literals $a = true$ and $a = false$ for all $a \in A$. If either of these is 0, then its negation is added to the implied literal label for n and a is not considered as a candidate for branching. We also experimented with optimizing the in-memory layout for better cache behavior. Compared to the van-Emde Boas layout [23], a depth-first, false-child first layout performed better.

Maintaining a decision tree during search

It is fairly straightforward to update a decision tree for a pruning. In order to prune a value, we remove the unique leaf node that corresponds to it. Once we remove a leaf, its parent may no longer be able to reach any more leaves, hence we propagate this removal upwards. We associate each removed node with the decision level in which it was removed, so on backtracking we add them back to restore the tree to its correct state.

This guarantees that the tree representation of a domain only shrinks down a branch of the branch and bound tree. Hence, the tree can remain static and we only mask nodes that do not lead to any leaves that correspond to unpruned values, which is simple to implement.

Reduced costs

ELSA solves the linear relaxation (1) from scratch at every node, and then strengthens it by discovering new violated cluster inequalities using the acyclicity checker (algorithm 1). Both these algorithms require an efficient implementation of the subset query on the subset of values which have reduced cost 0. In contrast to the domain itself, however, this set is reset to the empty set at the beginning of every node and grows monotonically until it admits an acyclic solution. Here again, the fact that there exists a bijection between values and leaves of the tree allows us to represent the set of 0-cost values as a subset of the full decision tree. Every time the reduced cost of a value reaches 0, the unique leaf it corresponds to, as well as all its parents, are added to the set of visible nodes for these queries. This is implemented as an additional mask on top of that which hides pruned values.

Subset queries

To answer the query "does the domain contain a value S such that $S \subseteq T$?", we perform a depth first traversal of the tree. At each node n, we check $l(n)$. If $l(n) \notin T$, we only allow DFS to follow the outgoing arc labeled with false. If $l(n) \in T$, we allow DFS to follow both outgoing arcs. If the label $lit(n)$ contains a literal $p \notin T$, we backtrack. If we reach a leaf, we stop and report success. If we exhaust the search without reaching a leaf, we report failure.

When this procedure reaches a node which is the root of a complete subtree of depth k, with no additional implied literal labels, it is guaranteed to terminate after visiting exactly k nodes and report success. Indeed, since this is a complete subtree, one of the outgoing arcs is always available to the depth first search, and it will reach a leaf after k more steps.

This procedure can be used to answer subset queries either on the entire domain, masking away only pruned values, or on those values which have reduced cost 0, masking away both pruned values and those whose reduced cost is greater than 0.

5 Experimental Results

We implemented decision trees as the domain representation on top of ELSA. The default implementation of a subset query in ELSA iterates over all domain values and returns if it finds one that is a subset of T. We replaced this by the depth-first traversal described in section 4 and denote this solver [1] ELSA^{IG}. We compare against the previous version of ELSA[2], GOBNILP[3], and CPBayes[4].

[1] Available at `https://gkatsi.github.io/elsa-cp22.tar.gz`

[2] Available at `https://gkatsi.github.io/elsa-ijcai21.tar.gz`

[3] Version 1.6.3 with CPLEX 12.7.1

[4] Retrieved from `http://cs.uwaterloo.ca/~vanbeek/Publications/CPBayes.zip`

The datasets come from the UCI Machine Learning Repository[5], the Bayesian Network Repository[6], and the Bayesian Network Learning and Inference Package[7]. We have 51 medium datasets with $|V| < 64$, and 18 large datasets with $64 \leq |V| < 128$.

Local scores were computed from the datasets using B. Malone's code[8]. BDeu and BIC scores were used for medium datasets (less than 64 variables) and only BIC score for large datasets (above 64 variables). The maximum number of parents was limited to 5 for large datasets (except for `accidents.test` with maximum of 8), a high value that allows learning even complex structures [18]. For example, `jester.test` has 100 random variables, a sample size of $4,116$ and $770,950$ parent set values. For medium datasets, no restriction was applied except for some BDeu scores, where we limit sets to 6 or 8 to complete the computation of the local scores within 24 hours of CPU-time [16].

For the experiments, we modified the C++ source of CPBayes v1.1 just to allow us to run it with datasets having more than 64 variables. All computations were performed on a single core of Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz and 1 TB of RAM with a 1-hour CPU time limit for the 51 medium datasets, as well as 3 of the large datasets: `kdd.ts`, `kdd.test`, and `kdd.valid`. For the remaining 15 large datasets, we had a 10-hour CPU time limit. For the preprocessing phase, we used two different settings depending on problem size $n = |V|$: $l_{min} = 20, l_{max} = 26, r_{min} = 50, r_{max} = 500$ if $n \leq 64$, else $l_{min} = 20, l_{max} = 20, r_{min} = 15, r_{max} = 30$, where l_{min}, l_{max} are partition lower bound sizes and r_{min}, r_{max} are the number of restarts for the local search.

In Table 1, we show the time needed to find the optimal solution and prove optimality for all these solvers. We see that, while the use of decision trees has little effect, either positive or negative, for the smaller instances, it makes a great difference in the larger instances. In particular, ELSA^{IG} is the only solver that can prove optimality for the `baudio.test` and `jester.valid` datasets. For the only instances where ELSA is significantly worse than CPBayes, `bnetflix.ts`, `bnetflix.test`, and `bnetflix.valid`, ELSA^{IG} either closes the gap back down (`bnetflix.valid`) or is faster yet than CPBayes (`bnetflix.ts` and `bnetflix.test`). However, ELSA^{IG} regresses with respect to ELSA in the `accidents` dataset and in `plants.test`. Part of the reason for this is that the benefit of the decision trees in terms of the reduction of the cost in answering the subset queries is comparatively reduced, hence the other overheads of decision trees dominate. For example, in `bnetflix.ts`, where ELSA^{IG} significantly outperforms ELSA, ELSA looks at an average of 3315 values to answer each subset test, while ELSA^{IG} visits just 90 nodes of the decision tree. On the other hand, in `accidents.test`, ELSA looks at an average of 80 values to answer each subset test, while ELSA^{IG} visits 20 nodes of the decision tree. This difference is not enough to overcome other overheads.

With respect to GOBNILP, ELSA^{IG} mostly outperforms it, but there are some instances where neither ELSA nor ELSA^{IG} can match it. It seems, however, that ELSA^{IG} is overall the best performer.

[5] `http://archive.ics.uci.edu/ml`
[6] `http://www.bnlearn.com/bnrepository`
[7] `https://ipg.idsia.ch/software.php?id=132`
[8] `http://urlearning.org`

Table 1 Comparison of GOBNILP, CPBayes, ELSA, and ELSA^{IG} in terms of total running (and search) time in seconds. Time limit for the datasets above the line is 1 hour, and for the rest it is 10 hours. Datasets are sorted by increasing total domain size for each time limit category. For CPBayes as well as all variants of ELSA we report in parentheses time spent in search, after preprocessing finishes. † indicates a timeout.

	n	sum |D|	GOBNILP	CPBayes	ELSA	ELSA^{IG}
carpo100_BIC	60	423	**0.5**	76.7 (27.5)	52.6 (0.1)	52.5 (0.0)
insurance1000_BIC	27	506	**0.6**	31.6 (0.0)	32.8 (0.0)	37.2 (0.0)
spectf_BIC	45	610	1.4	4.2 (3.5)	**0.8 (0.0)**	1.0 (0.1)
sponge_BIC	45	618	**1.6**	5.1 (3.3)	1.8 (0.0)	2.1 (0.0)
insurance1000_BDe	27	792	**0.6**	34.8 (0.0)	34.3 (0.0)	39.2 (0.0)
alarm1000_BIC	37	1002	**1.3**	191.1 (159.1)	34.4 (1.0)	37.9 (1.9)
flag_BDe	29	1324	4.0	16.6 (15.6)	**1.0 (0.2)**	1.3 (0.2)
autos_BIC	26	2391	**11.9**	18.4 (0.0)	19.2 (0.0)	19.9 (0.1)
soybean_BIC	36	5926	**48.9**	51.9 (1.7)	50.8 (3)	49.6 (0.0)
wdbc_BIC	31	14613	86.3	459.4 (398.0)	**56.0 (2.4)**	61.7 (1.7)
autos_BDe	26	25238	1005.2	239.5 (0.1)	**145.8 (0.8)**	177.1 (0.3)
kdd.ts	64	43584	**508.8**	†	1452.3 (274.6)	1355.2 (141.3)
steel_BIC	28	93026	†	1265.6 (1196.1)	124.2 (71.8)	**100.6 (45.7)**
kdd.test	64	152873	3178.0	†	1594.3 (224.4)	**1519.6 (48.9)**
mushroom_BDe	23	438185	†	167.0 (4.9)	182.6 (58.9)	**150.1 (16.7)**
bnetflix.ts	100	446406	†	1086.9 (876.3)	2103.1 (1900.9)	**557.9 (358.4)**
plants.test	111	520148	†	†	**28049.6 (26312.9)**	35961.7 (33712.7)
jester.ts	100	531961	†	†	21550.5 (21003.7)	**7951.4 (7301.6)**
accidents.ts	100	568160	**1932.2**	†	2302.2 (930.0)	†
plants.valid	111	684141	†	†	**17801.6 (14080.2)**	19819.2 (14547.9)
jester.test	100	770950	†	†	30186.8 (29455.0)	**9644.5 (8742.8)**
baudio.test	100	1016403	†	†	†	**31077.1 (29028.1)**
bnetflix.test	100	1103968	†	5794.5 (5486.2)	10333.1 (10096.5)	**1448.8 (1137.7)**
bnetflix.valid	111	1325818	†	**998.1 (451.0)**	10871.7 (10527.7)	1476.5 (1041.5)
accidents.test	100	1425966	14453.1	†	**3641.7 (680.7)**	8434.1 (4723.0)
jester.valid	100	1463335	†	†	†	**31949.5 (30624.2)**
accidents.valid	100	1617862	**27730.5**	†	†	†

6 Conclusion

We have shown that, in the BNSL problem, we can exploit the structure of domains to get a significant speedup in learning the structure of BNs of larger datasets. Specifically, we have shown that by treating domains as sets of sets instead of sets of values, and using decision trees to represent these sets, we can answer subset queries significantly faster. This is unlike the typical approach to handling large domains in CP, which uses over- and under-approximations. Although the current implementation shows some significant improvements, answering subset queries is still the most time consuming operation performed by the solver. Moreover, the fact remains that decision trees as a knowledge compilation language are fairly weak in terms of conciseness. It remains an open question whether we can overcome the issues with ROBDDs or even DNNFs to achieve even more significant speedups.

References

1 Mark Bartlett and James Cussens. Integer linear programming for the bayesian network structure learning problem. *Artificial Intelligence*, pages 258–271, 2017.

2 Jeremias Berg, Matti Järvisalo, and Brandon Malone. Learning optimal bounded treewidth bayesian networks via maximum satisfiability. In *Artificial Intelligence and Statistics*, pages 86–95. PMLR, 2014.

3 Wray Buntine. Theory refinement on bayesian networks. In *Proc. of UAI*, pages 52–60. Elsevier, 1991.

4 David Maxwell Chickering. Learning bayesian networks is NP-Complete. In *Proc. of Fifth Int. Workshop on Artificial Intelligence and Statistics (AISTATS)*, pages 121–130, Key West, Florida, USA, 1995. doi:10.1007/978-1-4612-2404-4_12.

5 James Cussens, Matti Järvisalo, Janne H Korhonen, and Mark Bartlett. Bayesian network structure learning with integer programming: Polytopes, facets and complexity. *Journal of Artificial Intelligence Research*, 58:185–229, 2017.

6 Cassio P de Campos, Mauro Scanagatta, Giorgio Corani, and Marco Zaffalon. Entropy-based pruning for learning bayesian networks using BIC. *Artificial Intelligence*, 260:42–50, 2018.

7 Cassio Polpo de Campos and Qiang Ji. Properties of bayesian dirichlet scores to learn bayesian network structures. In *Proc. of AAAI-10*, Atlanta, Georgia, USA, 2010.

8 Xiannian Fan and Changhe Yuan. An improved lower bound for bayesian network structure learning. In *Proc. of AAAI-15*, Austin, Texas, 2015.

9 Carmen Gervet. Conjunto: Constraint logic programming with finite set domains. In Maurice Bruynooghe, editor, *Logic Programming, Proceedings of the 1994 International Symposium, Ithaca, New York, USA, November 13-17, 1994*, pages 339–358. MIT Press, 1994.

10 Carmen Gervet and Pascal Van Hentenryck. Length-lex ordering for set CSPs. In *Proceedings of AAAI*, 2006.

11 Peter Hawkins, Vitaly L. Lagoon, and Peter J. Stuckey. Solving set constraint satisfaction problems using ROBDDs. *Journal of Artificial Intelligence Research*, 24:109–156, 2005.

12 David Heckerman, Dan Geiger, and David M Chickering. Learning bayesian networks: The combination of knowledge and statistical data. *Machine learning*, 20(3):197–243, 1995.

13 Y. Lai, D. Liu, and S. Wang. Reduced ordered binary decision diagram with implied literals: A new knowledge compilation approach. *Knowledge and Information Systems*, 35(3):665–712, 2013.

14 Wai Lam and Fahiem Bacchus. Using new data to refine a bayesian network. In *Proc. of UAI*, pages 383–390, 1994.

15 Hyafil Laurent and Ronald L Rivest. Constructing optimal binary decision trees is NP-complete. *Information processing letters*, 5(1):15–17, 1976.

16 Colin Lee and Peter van Beek. An experimental analysis of anytime algorithms for bayesian network structure learning. In *Advanced Methodologies for Bayesian Networks*, pages 69–80, 2017.

17 J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–107, 1986.

18 Mauro Scanagatta, Cassio P de Campos, Giorgio Corani, and Marco Zaffalon. Learning bayesian networks with thousands of variables. *Proc. of NeurIPS*, 28:1864–1872, 2015.

19 Gideon Schwarz. Estimating the dimension of a model. *The Annals of Statistics*, 6(2):461–464, 1978.

20 Tomi Silander and Petri Myllymäki. A simple approach for finding the globally optimal bayesian network structure. In *Proc. of UAI'06*, Cambridge, MA, USA, 2006.

21 Fulya Trösser, Simon de Givry, and George Katsirelos. Improved acyclicity reasoning for bayesian network structure learning with constraint programming. In *Proceedings of IJCAI*, pages 4250–4257, 2021.

22 Peter van Beek and Hella-Franziska Hoffmann. Machine learning of bayesian networks using constraint programming. In *Proc. of International Conference on Principles and Practice of Constraint Programming*, pages 429–445, Cork, Ireland, 2015.

23 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84, 1975.

24 Changhe Yuan and Brandon Malone. Learning optimal bayesian networks: A shortest path perspective. *J. of Artificial Intelligence Research*, 48:23–65, 2013.

Selecting SAT Encodings for Pseudo-Boolean and Linear Integer Constraints

Felix Ulrich-Oltean
Department of Computer Science, University of York, UK

Peter Nightingale
Department of Computer Science, University of York, UK

James Alfred Walker
Department of Computer Science, University of York, UK

Abstract

Many constraint satisfaction and optimisation problems can be solved effectively by encoding them as instances of the Boolean Satisfiability problem (SAT). However, even the simplest types of constraints have many encodings in the literature with widely varying performance, and the problem of selecting suitable encodings for a given problem instance is not trivial. We explore the problem of selecting encodings for pseudo-Boolean and linear constraints using a supervised machine learning approach. We show that it is possible to select encodings effectively using a standard set of features for constraint problems; however we obtain better performance with a new set of features specifically designed for the pseudo-Boolean and linear constraints. In fact, we achieve good results when selecting encodings for unseen problem classes. Our results compare favourably to AutoFolio when using the same feature set. We discuss the relative importance of instance features to the task of selecting the best encodings, and compare several variations of the machine learning method.

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

Keywords and phrases Constraint programming, SAT encodings, machine learning, global constraints, pseudo-Boolean constraints, linear constraints

Digital Object Identifier 10.4230/LIPIcs.CP.2022.38

Supplementary Material *Dataset (Repository with models and results)*: `https://github.com/felixvuo/cp2022-pbli-sat`

Funding *Felix Ulrich-Oltean*: Supported by grant EP/R513386/1 from the UK Engineering and Physical Sciences Research Council.

Acknowledgements We are very grateful to Nguyen Dang for helpful conversations about portfolio approaches. The experiments were undertaken on the Viking Cluster, which is a high performance compute facility provided by the University of York. We are grateful for support from the University of York High Performance Computing service, Viking and the Research Computing team.

1 Introduction

Many constraint satisfaction and optimisation problems can be solved effectively by encoding them as instances of the Boolean Satisfiability problem (SAT). Modern SAT solvers are remarkably effective even with large formulas, and have proven to be competitive with (and often faster than) CP solvers (including those with conflict learning). However, even the simplest types of constraints have many encodings in the literature with widely varying performance, and the problem of predicting suitable encodings is not trivial.

We explore the problem of selecting encodings for constraints of the form $\sum_{i=1}^{n} q_i x_i \diamond k$ where $\diamond \in \{<, \leq, =, \neq, \geq, >\}$, $q_1 \dots q_n$ are integer coefficients, k is an integer constant and x_i are decision variables. We separate these constraints into two classes: *pseudo-Boolean* (PB) when all x_i are Boolean variables or integer variables with two values; and *linear integer* (LI) when there exists an x_i variable with more than two possible values. We treat these two classes separately, selecting one encoding for each class when encoding an instance.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 38; pp. 38:1–38:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We select from a set of state-of-the-art encodings, including all four encodings of Bofill et al. [8, 9, 7] which are extensions of the Generalized Totalizer [16], Binary Decision Diagram [1], Global Polynomial Watchdog [6], and Sequential Weight Counter [14]. All four of these encodings are for pseudo-Boolean constraints with at-most-one (AMO) sets of terms (where at most one of the corresponding x_i variables are true). The AMO sets come from an integer variable or are detected automatically [5] as described in Section 2.1. We also use Savile Row's *Tree* encoding, which we describe briefly in this paper.

The context for this work is Savile Row [21], a constraint modelling tool that takes the modelling language Essence Prime and can produce output for various types of solver, including CP, SAT, and recently SMT [12]. When encoding a constraint to SAT, two different approaches may be taken depending on the type of constraint. Some constraint types are decomposed into simpler constraints prior to encoding (e.g. allDifferent is decomposed into a set of at-most-one constraints, stating that each relevant domain value appears at most once). Other constraint types are encoded to SAT directly, in which case Savile Row will apply the encoding chosen on the command-line (or the default if no choice is made).

We use a supervised machine learning approach, trained with a corpus of 615 instances from 49 problem classes (constraint models). We show that it is possible to select encodings effectively, approaching the performance of the virtual best encoding (i.e. the best possible choice for each instance), using an existing set of features for constraint problem instances. Also we obtain better performance by adding a new set of features specifically designed for the pseudo-Boolean and linear integer constraints, especially when selecting encodings for unseen problem classes.

1.1 Contributions

In summary, our contributions are as follows:

- We address the problem of selecting SAT encodings for instances of *unseen* problem classes, which we argue is a realistic version of the encoding selection problem. To our knowledge, all previous approaches (such as [15, 24]) train and test their machine learning models on instances drawn from the same set of problem classes.
- We describe a machine learning approach that produces very good results, and that performs much better than the mature, self-tuning algorithm selection tool AutoFolio [19].
- We present a new set of features for pseudo-Boolean and linear integer constraints, and show improved overall performance and robustness when using them.
- We evaluate our machine learning method thoroughly, and present an analysis of feature importance.

1.2 Preliminaries

A *constraint satisfaction problem* (CSP) is defined as a set of variables X, a function that maps each variable to its domain, $D : X \to 2^{\mathbb{Z}}$ where each domain is a finite set, and a set of constraints C. A *constraint* $c \in C$ is a relation over a subset of the variables X. The *scope* of a constraint c, named scope(c), is the set of variables that c constrains. A *constraint optimisation problem* (COP) also minimises or maximises the value of one variable. A *solution* is an assignment to all variables that satisfies all constraints $c \in C$. Boolean Satisfiability (SAT) is a subset of CSP with only Boolean variables and only constraints (*clauses*) of the form $(l_1 \vee \cdots \vee l_k)$ where each l_i is a literal x_j or $\neg x_j$. A *SAT encoding* of a CSP variable x is a set of SAT variables and clauses with exactly one solution for each value in $D(x)$. A SAT encoding of a constraint c is a set of clauses and additional Boolean variables A, where the clauses contain only literals of A and of the encodings of variables in scope(c). An encoding of c has *at least* one solution corresponding to each solution of c. *Generalised arc consistency* (GAC) for a constraint c means that for a given partial assignment, all values are removed from the domain of each variable in scope(c) if they cannot appear in any extended assignment satisfying c. A SAT encoding of c has the property *UP maintains GAC* iff unit propagation of the SAT encoding of c results in the following correspondence: for each variable $x_i \in \text{scope}(c)$, the set of solutions of the encoding of x_i corresponds to the set of values in $D(x_i)$ after GAC has been enforced on c.

2 Learning to Choose SAT Encodings

First we describe the palette of encodings for PB and LI constraints, then our approach to selecting encodings using instance features and machine learning.

2.1 SAT Encodings

Recall that we are considering constraints of the form $\sum_{i=1}^{n} q_i x_i \diamond k$ where $\diamond \in \{<, \leq, =, \neq, \geq, >\}$, $q_1 \ldots q_n$ are integer coefficients, k is an integer constant and x_i are decision variables (of type integer or Boolean). We use five encodings and each can be applied to either PB or LI constraints, giving 25 configurations in total.

2.1.1 PB(AMO) Encodings

The first four are encodings of PB(AMO) constraints [8, 9, 7], which are pseudo-Boolean constraints with non-intersecting at-most-one (AMO) groups of terms (where at most one of the corresponding x_i variables are true in any solution). Encodings of PB(AMO) constraints can be substantially smaller and more efficient to solve than the corresponding PB constraints [8, 9, 5, 7]. For the four PB(AMO) encodings the constraints must be placed in a normal form where all coefficients are positive, only $\leq$ is allowed, and each x_i must be Boolean. We exactly follow the normalisation rules of Bofill et al. [7].

When encoding LI constraints, each integer CSP variable x_i in the constraint (where $|D(x_i)| > 2$) is required to have a direct encoding. When an integer variable (with $|D(x_i)| > 2$) appears in an LI constraint it is replaced with an AMO group of $|D(x_i)|-1$ terms representing each value except the smallest (which is cancelled out). In the case where x_i is an integer variable with two values, Savile Row encodes x_i with a single Boolean variable that is true iff x_i takes its larger value. Also, automatic AMO detection [5] (which applies constraint propagation to find AMO groups among the Boolean terms of the original constraint) is enabled in our experiments. Automatic AMO detection has been shown to substantially improve solving time in some cases [5].

Equality constraints are decomposed into two inequalities ($\leq$) prior to encoding. Disequality constraints and any constraints that are not top-level (i.e. are nested in another expression such as a disjunction) are encoded with the *Tree* encoding, described in Section 2.1.2. Full details of the conversion of integer terms and normalisation for PB(AMO) encodings are given elsewhere [5]. The PB(AMO) encodings are as follows:

MDD The Multi-valued Decision Diagram encoding [8] (a generalisation of the BDD encoding for PB constraints [1]) uses an MDD to encode the PB(AMO) constraint. Each layer of the MDD corresponds to one AMO group. BDDs and MDDs are a popular choice for encoding sums to SAT since they can compress equivalent states in each layer.

GGPW The Generalized Global Polynomial Watchdog encoding [9] (generalising GPW [6]) is based on bit arithmetic and is polynomial in size.

GGT The Generalized Generalized Totalizer [9] encodes the PB(AMO) constraint with a binary tree, where the leaves represent the AMO groups and each internal node represents the sum of all leaves beneath it. GGT compresses equivalent states at its internal nodes. It extends the Generalized Totalizer [16].

GSWC The Generalized Sequential Weight Counter [9] (based on the Sequential Weight Counter [14]) encodes the sum of each prefix sub-sequence of the AMO groups.

Unit propagation on the MDD, GGT, and GSWC encodings enforces GAC on the original constraint c when c is a PB(AMO) $\leq$ constraint [9] but not when c contains integer terms or is an equality. GGPW does not have this property.

2.1.2 Tree Encoding

The *Tree* encoding is related to GGT, however it is not a PB(AMO) encoding. *Tree* does not require the constraint to be an inequality, nor to have positive coefficients.

Tree Given a constraint c, each term is shifted such that its smallest value becomes 0, and k is adjusted accordingly. A binary tree is constructed from the sum, with each term (integer or Boolean) attached to a leaf. Internal nodes represent the sum of the leaves beneath them. The order encoding is required for integer leaf nodes (Savile Row generates the *direct* or *order* encoding for variables as required [20]) and is also used for internal nodes. Each internal node is connected to its two children using the order encoding of linear constraints [25]. The root node represents the entire sum, and its set of values is restricted to those satisfying the constraint c. *Tree* can directly encode constraints with integer terms, equality and disequality, but does not benefit from automatic AMO detection.

Unit propagation on Tree enforces GAC on the original constraint c when c is not an equality or disequality.

The set of 5 encodings is diverse but not exhaustive. Abío et al proposed a BDD-based encoding for linear constraints [3], however it has been directly related to the MDD encoding [10]. In addition to MDD-based encodings, Abío et al propose two further encodings for linear constraints [2]: one based on sorting networks (SN), which is related to the GPW encoding, and another log-based encoding BDD-Dec. Other log encodings such as the one used by Picat-SAT [27] may also be more effective in some cases.

For our experiments we use an extended version of Savile Row 1.9.1 [20]. All constraints other than PB and LI use the default encoding as described in the Savile Row manual.

2.2 Instance Features

f2f We use the `fzn2feat` tool [4] to extract 95 static instance features relating to the number and types of variables and their domains, the types and sizes of constraints and features of the objective in optimisation problems. The full list of features can be found at `https://github.com/CP-Unibo/mzn2feat`; some features were not applicable, e.g. there are no float variables in Essence Prime and Savile Row does not produce all the same annotations.

f2fsr We also re-implement these features as closely as possible in Savile Row, applied to the model directly before encoding to SAT.

lipb We introduce a new set of 45 features describing the PB constraints in a problem instance. We also extract these for LI constraints, giving 90 new features in total. These features are listed in Table 1.

combi We combine the *f2fsr* and *lipb* features.

Table 1 New features for pseudo-Boolean and linear integer constraints. For each aspect of a constraint listed in the left column, we calculate the aggregates in the right column. In the aggregation functions, *IQR* means inter-quartile range, *skew* refers to the non-parametric skew and *ent* is Shannon's entropy. The identifier for each aspect is given in brackets.

Aspect of constraint	Aggregate
Number of (PB or LI) constraints	count
Number of terms (`n`)	min, max, mean, median, IQR, skew, ent, sum
Sum of coefficients (`wsum`)	sum, skew, IQR
Minimum coefficient (`q0`)	min, mean
Maximum coefficient (`q4`)	max, median, mean
Median coefficient (`q2`)	median, skew, ent
IQR of coefficients (`iqr`)	median, skew
Coefficients' quartile skew (`skew`)	mean, min, max, ent
Distinct coefficient values (`sep`)	mean, max
Ratio of distinct coeff. values to number of coeffs. (`sepr`)	mean, max
Number of At-Most-One groups (AMOGs) (`amogs`)	mean
Mean size of AMO group (`asize_mn`)	mean
Mean AMOG size ÷ number of terms (`asize_r2n`)	mean
Mean maximum coeff. size in AMOGs (`amaxw_mn`)	mean
Skew of maximum coeff. in AMOGs (`amaxw_skew`)	mean, ent
Upper limit (k) (`k`)	mean, median, max, IQR, ent, skew
k × number of AMOGs (`k_amo_prod`)	mean, IQR, ent

2.3 Problem Corpus

We use the 65 constraint models with a total of 757 instances from a recent paper [12] in order to work with a wide variety of problem classes. An added advantage is that the models are written in Essence Prime, Savile Row's input language. Unfortunately this collection has a very skewed distribution of instances per problem class, ranging from just 1 to 100. We address this by limiting the number of instances per class to 50 (a random sample) and by adding instances to existing classes where it is easy, such as when instance parameters are just a few integers. We also add two problem classes from recent XCSP3 competitions [18]. More details of the corpus after cleaning are given in Section 3.2 and Table 2.

2.4 Training

We evaluated several classifier models from the `scikit-learn` library [22], including decision trees and forests of extremely randomised trees. We also investigated training various regressors to predict runtime. We find that random forest classifier performs best for our purposes. The `scikit-learn` implementation is based on Breiman's random forests [11], but uses an average of predicted probabilities from its decision trees rather than a simple vote.

We follow the method of Probst et al. [23] who investigated hyperparameter tuning for random forests and concluded that the number of estimators should be set sufficiently high (we use 200) and that it is worth tuning the *number of features*, *maximum tree depth*, and *sample size*. We use randomised search with 50 iterations and 5-fold cross-validation to tune the hyperparameters. We experimented with more tuning iterations but it did not lead to improved prediction quality.

If a classifier makes a poor prediction, the consequences vary. It is possible that the chosen encodings lead to a running time which is very close to that of the ideal choice; the opposite is also true and misclassification can be very expensive. To address this issue, we follow a similar approach to the *pairwise classification* used in AutoFolio [19]: we train a random forest model for each of the possible pairs of encoding configurations. When making predictions, each model chooses between its two candidates. The configuration with most votes is chosen; if two or more configurations have equal votes, we select the one which produced the shortest total running time over the training set. This approach effectively creates a predicted ranking of configurations from the features and leads to better prediction performance than using a single random forest classifier.

To facilitate the pairwise training and prediction approach, we reduce our selection of encoding combinations from 25 (5 PB encodings × 5 LI encodings) to a portfolio of 5, thus needing to train just 10 models (rather than 300 if we had used all 25 choices). We seek to retain performance complementarity as described in [17] from a much reduced portfolio size. The portfolio is built from the timings in the training set using a greedy approach as follows. We begin with a single encoding configuration in the portfolio and then successively add the remaining configuration which would lower the virtual best PAR2 time (PAR2 is defined in Section 3.2) by the biggest margin. We do this until we have a portfolio of 5. We repeat the process using each of the 25 configurations as the starting element and finally select the best-performing portfolio from these 25. Figure 1 shows that this portfolio reduction has a small impact on the virtual best performance across our corpus – the virtual best time for a portfolio of size 5 is within 15% of the time achievable with all configurations.

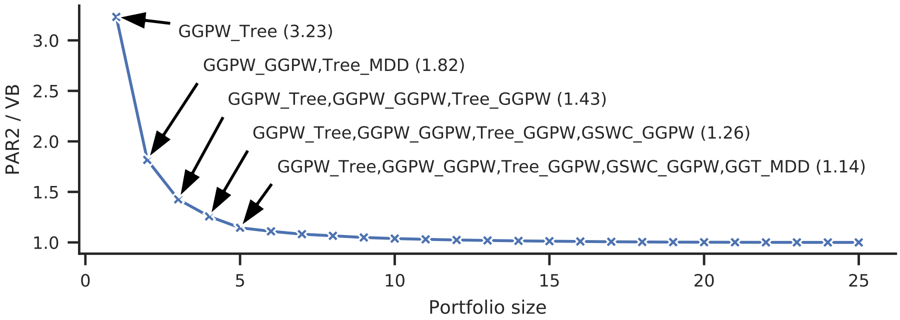


Figure 1 The virtual best PAR2 run-time on our corpus for all portfolio sizes as a multiple of the overall virtual best; the resulting portfolios (of *li_pb* configurations) are shown for sizes 1 to 5.

In addition to the pairwise voting scheme, we implement two further customisations when training the classifiers. Firstly we apply *sample weights* to give more importance to harder instances (those with a longer virtual best runtime) during training. Each instance is given a positive integer weight w according to $w = \lfloor \log_{10}(10(1 + t_{vb})) \rfloor$, where t_{vb} is the virtual best running time for the instance. Secondly, we provide a *custom loss function* for the cross-validation used during hyperparameter tuning. The loss function simply returns the difference in time between the runtime of the chosen encoding configuration and the virtual best for that instance.

To conduct a more complete comparison we also implement two additional alternative setups. Firstly we use a single classifier with a portfolio of 5 configurations (combining PB and LI encodings) and allow it the same number of hyperparameter tuning iterations as the

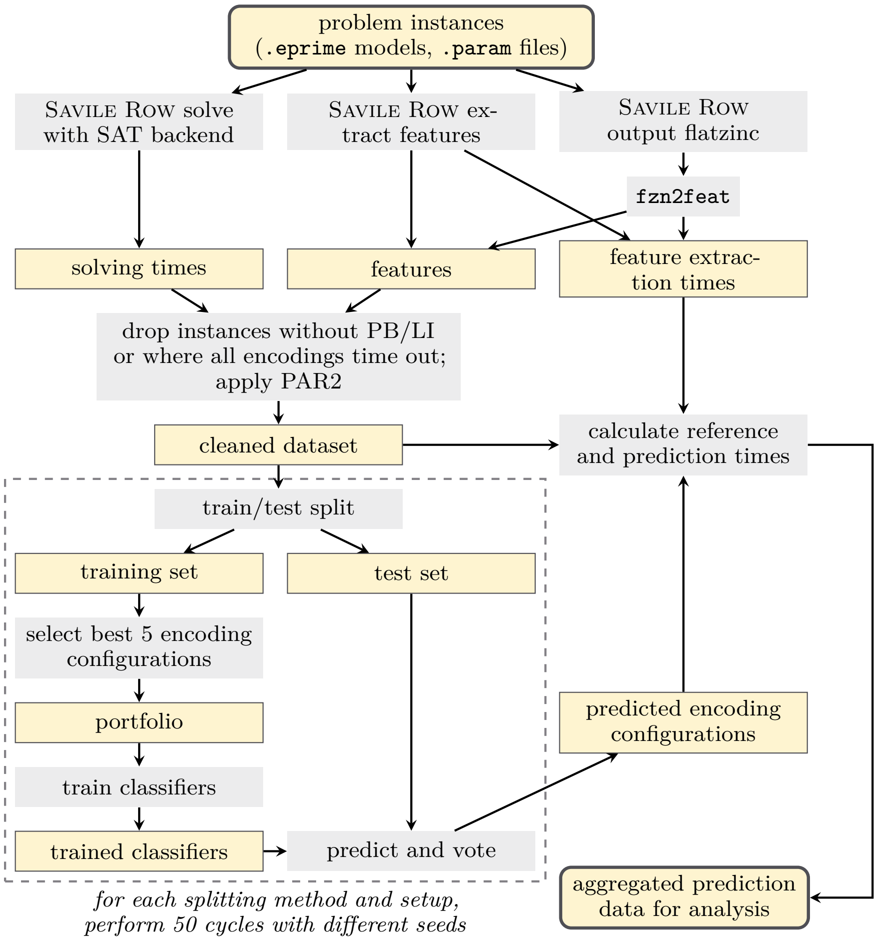

problem instances
(.eprime models, .param files)
SAVILE ROW solve with SAT backend
SAVILE ROW extract features
SAVILE ROW output flatzinc
fzn2feat
solving times
features
feature extraction times
drop instances without PB/LI or where all encodings time out; apply PAR2
cleaned dataset
calculate reference and prediction times
train/test split
training set
test set
select best 5 encoding configurations
portfolio
train classifiers
trained classifiers
predict and vote
predicted encoding configurations
aggregated prediction data for analysis
for each splitting method and setup, perform 50 cycles with different seeds

Figure 2 An overview of the steps involved in our experimental investigation. The boxes with solid borders represent data; the grey boxes represent processes.

total used by the pairwise classifiers (i.e. 50 × 10). Secondly we modify the pairwise setup to make a separate prediction for PB and LI constraints – this approach has its difficulties because when labelling the dataset with the best encoding for one type of constraint, the other constraint must be chosen somehow. We address this by setting the other constraint to the single best for the training set; however this process is not easily scalable in the same way as the other setups we present. In both of these setups we use sample weighting and the custom loss function for cross-validation.

3 Empirical Investigation

We provide an overview of our experimental process in Figure 2.

3.1 Solving Problem Instances and Extracting Features

We run SAVILE ROW on each instance in the corpus with each of the 25 encoding configurations. The CNF clause limit is set to 5 million and the SAVILE ROW time-out to 1 hour. We switch on automatic detection of At-Most-One constraints [5]. We choose Kissat as our SAT solver as it formed the basis of the top three performers in the 2021 SAT competition [13]. We

Table 2 Number of instances *(#)*, mean number of PB constaints *(PBs)* and mean number of linear integer constaints *(LIs)* per instance for each problem class in the eventual corpus.

Problem Class	#	PBs	LIs	Problem Class	#	PBs	LIs
killerSudoku2	50	1811.2	129.9	carSequencing	49	435.7	0.0
knights	44	170.5	336.9	langford	39	146.2	0.0
opd	38	21.7	74.8	knapsack	30	1.0	1.0
sonet2	24	10.0	1.0	immigration	23	0.0	1.0
bibd-implied	22	410.6	0.0	handball7	20	705.0	1206.0
mrcpsp-pb	20	90.0	45.7	n_queens	20	1593.0	0.0
efpa	20	156.6	0.0	bibd	19	338.7	0.0
n_queens2	16	309.0	0.0	briansBrain	16	0.0	1.0
life	16	0.0	438.9	molnars	15	0.0	4.0
bpmp	14	14.0	0.0	blackHole	11	202.2	0.0
pegSolitaireTable	8	59.9	0.0	pegSolitaireState	8	59.9	0.0
pegSolitaireAction	8	59.9	0.0	peaceArmyQueens1	7	0.0	1008.0
peaceArmyQueens3	6	0.0	4.0	golomb	6	59.2	38.7
quasiGrp5Idem	6	586.7	0.0	magicSquare	6	118.3	34.0
quasiGrp7	6	410.7	0.0	quasiGrp6	6	410.7	0.0
quasiGrp4NonIdem	4	1067.5	208.0	quasiGrp3NonIdem	4	1067.5	208.0
quasiGrp5NonIdem	4	389.0	0.0	quasiGrp4Idem	4	416.0	208.0
bacp	4	0.0	25.0	quasiGrp3Idem	4	458.0	208.0
waterBucket	4	0.0	46.0	discreteTomography	2	240.0	0.0
solitaire_battleship	2	72.0	16.0	plotting	1	1.0	28.0
nurse	1	27.0	42.0	grocery	1	0.0	2.0
farm_puzzle1	1	0.0	2.0	diet	1	0.0	6.0
sokoban	1	0.0	24.0	sonet	1	3.0	1.0
contrived	1	0.0	4.0	sportsScheduling	1	166.0	64.0
tickTackToe	1	6.0	14.0				

use the latest release available at the time, `sc2021-sweep` [26], with default settings and separate time limit of 1 hour. The experiment is run on a research cluster [name removed for anonymity] with Intel Xeon 6138 20-core 2.0 GHz processors; we set the memory limit for each job to 8 GB. We carry out 5 runs (with distinct random seeds) for each configuration to average out stochastic behaviour of the solver.

To extract the features we run each problem instance once with the Savile Row feature extractor and once to generate standard FlatZinc (using the `-flatzinc` flag) followed by `fzn2feat` [4]. We record the time taken to extract the features.

3.2 Cleaning the Dataset

We calculate the median runtime over 5 runs for each instance and encoding configuration, and filter the corpus as follows. We mark a result as timed out if the total runtime (Savile Row + Kissat) exceeds 1 hour. We use PAR2 times, i.e. assigning 2 hours to any result which takes longer than our time-out limit. We choose PAR2 rather than PAR10 as used in some literature [15, 19, 24] because when choosing between our encodings the worst encoding for an instance tends to be around 2 times slower – the median *worst:best* runtime ratio is 1.85 for instances which don't time out. We drop instances if they contain no PB or LI constraints. We exclude instances for which all configurations time out, as well as instances which end up requiring no SAT solving – Savile Row can sometimes solve a problem in pre-processing through its automatic re-formulation and domain filtering. At this point, 615 instances of 49 problem classes remain in the corpus; Table 2 shows the number of instances for each problem class and the mean number of PB and LI constraints per instance.

3.3 Splitting the Corpus, Training and Predicting

For each of our classifier setups and our four featuresets, we run a *split, train, predict* cycle 50 times. We use seeds 1 to 50 to co-ordinate the splits so that we compare the prediction power of the different feature sets and setups using the same training and test sets.

For each cycle, we obtain an 80:20 train to test split using two approaches. The *split-by-instance* approach simply selects instances at random with uniform probability – with this approach, instances of a problem class are usually found in both the training and test sets. The *split-by-class* approach also splits problems randomly but ensures that all instances relating to a problem class end up either in the training or the test set, ensuring that predictions are being made on unseen problem classes. This second method can lead to the test set being slightly larger than 20%.

Prior to training the classifiers, the portfolio of available configurations is built based on the runtimes of the training set. The training instances are labelled for each pairwise classifier with the configuration which has the fastest runtime. For each pairwise classifier, we search the hyperparameter space and fit the model to the training set. Finally, we make predictions using the test set ready for evaluation.

3.4 Evaluating the Performance of Predicted Encodings

To evaluate the impact of using the learnt encoding choices, we calculate two benchmarks commonly used in algorithm selection [17]: the *Virtual Best (VB)* time is the total time taken to solve the instances in a test set if we always made the best choice from our portfolio of configurations; the *Single Best (SB)* time is how long it would take using the one configuration from the portfolio which performs fastest on our training set. In addition we refer to: the time taken using Savile Row's *default (Def)* configuration, which is the *Tree* encoding for both PB and LI constraints, and finally the *Virtual Worst (VW)* time to indicate the overall variation in performance of the encoding configurations in the portfolio.

In Table 3 we report the total PAR2 runtime across all 50 test sets for the predicted encoding configurations from each of the six classifier setups, four feature sets and two splitting methods. The predicted runtimes include the time taken to extract the features.[1] For ease of comparison, we report the runtime as a multiple of the virtual best time. For example, a figure of 2 in Table 3 means that the predictions across the 50 test sets led to a total runtime which was twice as long as the runtime achieved if we always chose the best available encoding combination from each portfolio (as determined from the training set in each cycle).

3.5 Results and Discussion

We found that the machine learning predictors work well, clearly outperforming the *SB* and *Def* configurations. These performance improvements can be achieved with predictions based on the generic CSP feature sets *f2f* and *f2fsr*, but are even better when using the new specialised features (*lipb*). Sometimes the best results are obtained by the combined featureset *combi*. This seems particularly true when predicting for unseen problem classes.

[1] For features extracted directly from Savile Row (*f2fsr, lipb, combi*), the feature extraction time added a median of 9% (mean 23%) to the overall running time. The features extracted via `fzn2feat` added 68% (median), 73% (mean).

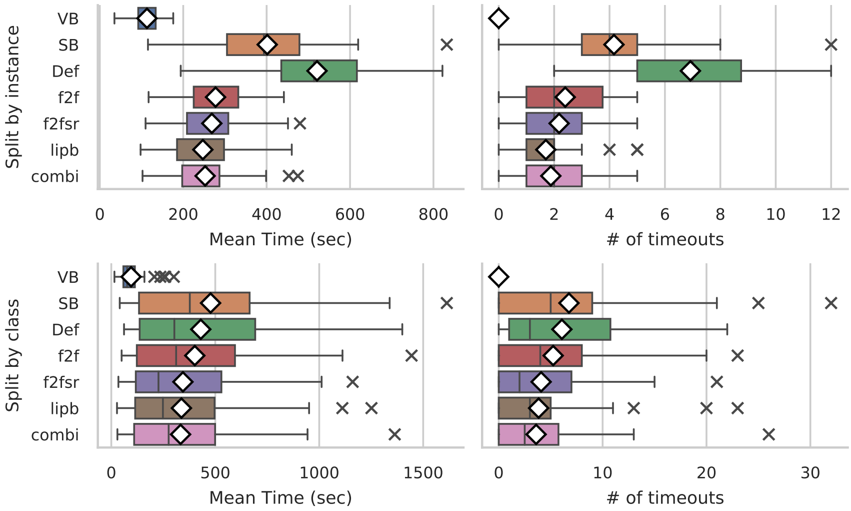


Figure 3 Prediction performance using different featuresets against reference times. We show mean runtime (left) and number of timeouts (right) per test set, when using our preferred setup (*pairwise combined + sample weights + custom loss*).

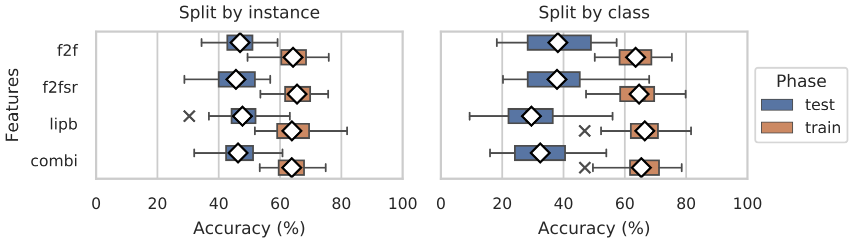


Figure 4 Prediction accuracy per cycle using our preferred setup.

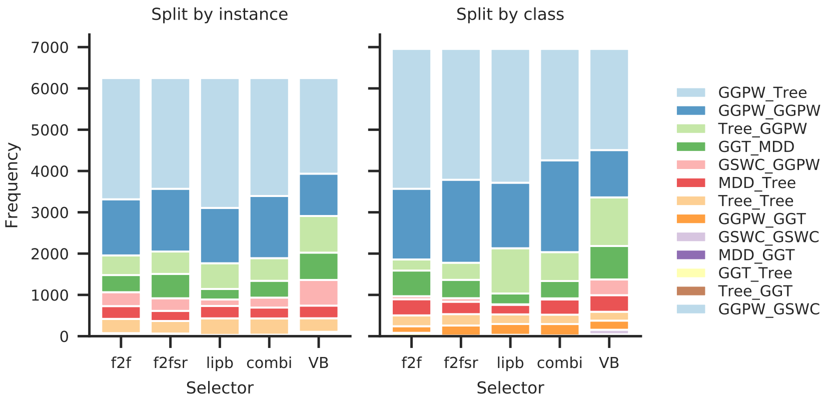


Figure 5 Frequency of each configuration (*li_pb*) selected across the 50 test sets when using each feature set with our preferred setup. We also show the virtual best (VB) configuration for comparison.

Table 3 Total PAR2 times over the 50 test sets as a multiple of the virtual best configuration time. We show the reference times for virtual best (VB), single best (SB), default (Def), and virtual worst (VW) configurations followed by the timings using predictions made using our four feature sets, our six machine learning setups and two splitting strategies. In the setups, *sw* means sample weighting is used and *cl* is when custom loss is used in cross-validation. The best time for each combination of setup, features and split is shown in **bold**. The predicted runtimes include feature extraction time.

Reference Times

Split	VB	SB	Def	VW
by instance	1.00	3.55	4.61	9.75
by class	1.00	5.06	4.53	9.49

Predicted Times

	Split by instance				Split by class			
Setup	f2f	f2fsr	lipb	combi	f2f	f2fsr	lipb	combi
pairwise combined	2.62	2.57	**2.41**	2.51	3.88	3.92	**3.75**	3.90
pairwise combined + sw	2.49	2.46	**2.28**	2.37	3.70	4.12	3.86	**3.52**
pairwise combined + cl	2.62	2.43	**2.36**	2.41	3.97	3.98	**3.58**	3.66
pairwise combined + sw + cl	2.45	2.37	**2.18**	2.23	4.24	3.66	3.56	**3.53**
single combined + sw + cl	2.43	2.43	**2.33**	2.36	4.23	4.43	3.89	**3.74**
pairwise separate + sw + cl	2.35	2.26	2.24	**2.18**	4.01	**3.90**	4.36	3.95

We argue that the split-by-class approach is both a more difficult challenge and closer to a real-world deployment, where a new instance to solve may belong to an unseen problem class. However, both approaches are realistic, so we choose *pairwise combined +sw +cl* as our *preferred setup* for the rest of this paper. For split-by-class it is very close to the best, and for split-by-instance is has a sizeable advantage.

In a recent survey, Kerschke et al. state that "State-of-the-art per-instance algorithm selectors for combinatorial problems have demonstrated to close between 25% and 96% of the VBS-SBS gap" [17]. In these terms, our preferred setup using *combi* features closes 38% of the VB-SB gap for unseen classes and 52% for seen classes using the *combi* features (rising to 54% if we use just *lipb* features). Because the distribution of runtimes in the split-by-class trials is skewed, we use a non-parametric statistical test to report on the significance of the improvement achieved by using our classifier. We apply the Wilcoxon Signed-Rank test for paired samples on the mean times from the SB selector choices and our preferred selector using the *combi* features. We obtain a p value of 4.3×10^{-5} (well below even a 1% significance level) and an effect size of -0.67 using the rank-biserial correlation method, or -0.58 using $\frac{z}{\sqrt{n}}$ which would usually be interpreted as a medium to large effect.

Figure 3 summarises the performance for our preferred setup, showing the distribution of mean predicted times per test. The mean values are marked with diamonds and correspond to the numbers reported in Table 3, albeit not scaled. We note that when splitting by instance, the performance across test sets is fairly symmetrical, with very similar means and medians. However, when it comes to splitting by class, the distribution shows positive skew – this likely comes from test sets where there are many instances from an unseen problem class for which the classifier struggles to make the best choices. It is interesting to consider that as we move from the *f2fsr* features to *lipb* and finally to *combi*, the mean remains roughly the

Table 4 Comparison of our system's performance with AutoFolio. As before, the overall runtimes for all test instances are reported as multiples of the virtual best (VB) and the best result for each setup is shown in **bold**. The first entry is our system with the preferred setup which includes sample weighting (*sw*) and custom loss (*cl*); to match AutoFolio we use a 10% test split and PAR10 timings. The final two entries show the timings for AutoFolio's predictions after 1 and 2 hours of training. All predicted timings include feature extraction time.

Reference Times

Split	VB	SB	Def	VW
by instance	1.00	10.14	18.60	41.41
by class	1.00	21.91	18.99	43.65

Predicted Times

	Split by instance				Split by class			
Setup	f2f	f2fsr	lipb	combi	f2f	f2fsr	lipb	combi
pairwise combined + sw + cl	5.68	5.95	**5.18**	5.41	14.39	14.58	13.75	**12.45**
AutoFolio (1hr)	20.33	19.90	**19.28**	21.21	21.82	**20.01**	**20.01**	21.87
AutoFolio (2hrs)	20.01	18.79	19.48	**18.33**	22.99	25.19	**17.17**	21.57

same, whereas the median time is increasing. This suggests that the generic *f2fsr* features perform better in the "easier" 50% of test sets, but that good work is undone by costly misclassification in harder test sets. This observation is also echoed by the distribution of timeouts, also shown in Figure 3. Again, *f2fsr* seems to avoid more timeouts in the easier half of tests; however, when considering the entire distribution, the *lipb* features lead to the fewest timeouts, offering more robust protection against a bad choice of encoding.

A further insight is provided by Figure 4 which shows the accuracy of predictions across the 50 training and test sets – in this figure we see how often the pairwise classifier ends up making exactly the "right" decision. In the split-by-instance scenario the prediction accuracy is fairly consistent across feature sets; however, for unseen classes we see once again that the generic feature sets can spot the very best encoding on more occasions (they have a higher average accuracy on the test sets) but they lead to more costly misclassification as shown by the evaluation of overall runtimes above.

In Figure 5 we show the frequency with which different encoding configurations are predicted. Recall that although we use a portfolio of 5 encodings, this is generated from the training set; consequently the portfolios are different across the 50 sets. One notable finding is that the *GGPW_Tree* and *GGPW_GGPW* appear to be low-risk choices that often perform well, and consequently are favoured by our classifiers. For both split-by-class and split-by-instance, all four selectors choose *GGPW_Tree* or *GGPW_GGPW* more frequently than the VB. Other choices such as *Tree_GGPW* are chosen less often than the VB. Another case in point is the *GSWC_GGPW* encoding: in the split-by-class trials the oracle (VB) uses this configuration in a few hundred cases, but our most successful feature sets (*lipb* and *combi*) eschew it almost entirely. This is likely due to the fact that the GSWC encoding can grow very large and perform badly in some cases; so our predictors seem to choose safer options, being encouraged by the PAR2 penalty to avoid timeouts.

3.6 Comparison with AutoFolio

To further assess the value of our approach, we compare with AutoFolio [19], a sophisticated algorithm selection approach which automatically configures algorithm selectors and "can be applied out-of-the-box to previously unseen algorithm selection scenarios." We use the latest version of AutoFolio (the 2020-03-12 commit which adds a CSV API to the 2.1.2 release) with its default settings. To compare as fairly as possible, we run our system with a similar setup to AutoFolio, namely a 10% test sample and PAR10 times. Our system takes less than 5 minutes to train using 8 cores on the cluster, so we allow AutoFolio 1 hour on one core. We also run it with a more generous budget of 2 hours to see if its performance improves. The results of these runs are shown in Table 4.

Our system's predictions lead to better runtimes than AutoFolio's. AutoFolio is designed to be a good general algorithm selection and configuration system able to make good predictions when choosing between different solvers. It is likely that AutoFolio's sophisticated decision-making is better suited to problems that run much longer or to algorithms for which the likelihood of timeouts or non-termination is more of an issue. It is interesting to note that AutoFolio performs better with the *lipb* features than the generic instance features. Allowing AutoFolio more time for tuning led to marginal improvement with some feature sets, but in some cases actually led to worse performance, for example with *split-by-class* and the *f2fsr* features.

3.7 Feature Importance

We investigate the relative importance of instance features by computing the permutation feature importance. Breiman [11] calculates "variable importance" in random forests by recording the percentage increase in misclassification when each variable (feature) has its values randomly permuted compared to when all features are used. Permuting the values means that the distribution is preserved but the feature effectively becomes noise. This method is applied at prediction time to the test set, unlike the Gini (entropy) feature importance measure which is calculated during training. We implement this analysis but record the mean increase in PAR2 time when each feature is permuted, effectively giving us the extra runtime cost when the feature is lost. Each feature is randomly permuted 5 times and the mean PAR2 time increase recorded. The distribution of feature importance thus calculated is shown in Figure 6. We report on the *lipb* features and on the *combi* feature set which additionally contains the generic features from *f2fsr*.

We can see for both feature sets that the median feature importance in the majority of cases is close to zero, but the mean importance is substantial. This suggests that there are no features which are dominant on their own – most of the time a missing feature incurs no loss of prediction performance. Indeed sometimes removing a feature can improve performance, as shown by some negative costs in most box plots. However, the means of the distributions show that there are cases where each of the features shown is able to prevent a costly wrong choice.

Notice that in the top 20 *combi* features we find a roughly equal mix of generic features and features specific to PB/LI constraints (the names of these features have prefixes `pb_` and `li_`). This is in keeping with the similar performance of the *f2fsr* and *libp* feature sets as shown previously in Table 3.

We suspect that when splitting by instance the system is, to a large extent, recognising problem classes rather than picking out traits of PB/LI constraints. Even when we predict for unseen problem classes, the proportion of PB/LI to generic features in the top 20 is

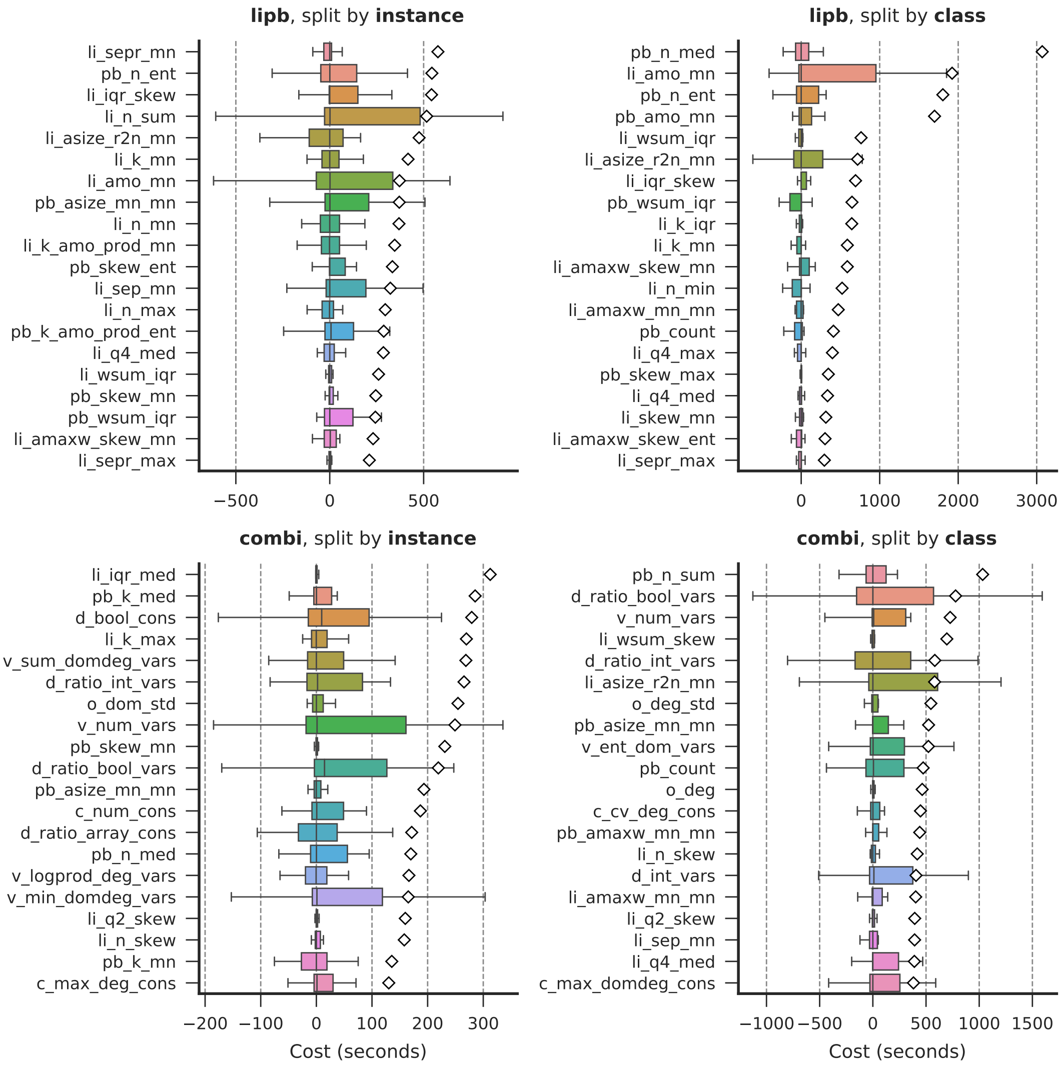


Figure 6 Permutation feature importance: increase in PAR2 time (mean from 5 trials) over 50 *split, train, predict* cycles. We show the top 20 features ordered by mean importance and we do not plot outliers (beyond 1.5 × IQR away from the box). The mean is shown by a diamond. Features beginning `li_` or `pb_` refer to our LI/PB features as introduced in Table 1; the other feature names refer to the generic instance features from the *combi* feature set.

roughly equal. Although we are keeping problem classes apart in this second case, there may be similarities in the constraint models between some problem classes. These similarities might extend beyond the characteristics of PB/LI constraints so the classifier can make a choice of PB/LI encoding on the basis of a choice which worked well in a problem class with similar generic features. This interpretation is also supported by the fact that the *lipb* feature set is sometimes matched or even outperformed by *combi* in split-by-class predictions as shown in Table 3.

Of the PB/LI-specific features, the ones extracted from LI constraints feature more strongly – this may reflect the fact that in our corpus the average number of LI constraints per instance is considerably higher than the number of PB constraints, so getting the LI choice right is more important.

There are limitations to how much we can read into the permutation feature importance, especially when we have quite a substantial number of features; a feature may be discriminating but masked by another feature with which it is highly correlated. We have shown that the features in *libp* and *f2fsr* can give comparable prediction performance even though they consider different aspects of a CSP.

4 Related Work

In recent work, new or improved SAT encodings of linear constraints [2] and pseudo-Boolean constraints (combined with AMO constraints) [9, 7] have been devised and their performance compared on several benchmark problems. The scaling properties of encodings are studied, and it is suggested that smaller encodings should be used when coefficients or values of integer variables are large. However, to the best of our knowledge the problem of selecting an encoding (particularly for a previously-unseen problem class) has not been systematically addressed for LI or PB constraints. We use the full set of encodings from one recent paper [9] combined with automatic AMO detection [5].

MeSAT [24] and Proteus [15] both select SAT encodings using machine learning. MeSAT has two encodings of LI constraints: the order encoding [25]; and an encoding based on enumeration of allowed tuples of values (which uses a direct encoding of the CSP variables). It is not clear whether high-arity sums are broken up before encoding. MeSAT selects from three configurations using a k-nearest neighbour classifier using 70 CSP instance features. They report high accuracy (within 4% of the virtual best configuration), however the single best configuration is only 18% slower than the virtual best. Proteus makes a sequence of decisions: whether to use CSP or SAT; the SAT encoding; and the SAT solver to use. The portfolio contains three SAT encodings: direct, support, and a hybrid direct-order, however the encoding of LI constraints is not specified [15]. Proteus generates each candidate SAT encoding and extracts features of the SAT formula to inform its selection – scaling this approach would be difficult when several constraint types are involved, each with many encoding choices. Results show that the choice of encoding (combined with the choice of SAT solver) is important and that machine learning methods can be effective in their context.

5 Conclusions and Future Work

We have shown that it is possible to close much of the performance gap between the single best and virtual best SAT encodings by using machine learning to select encoding configurations based on instance features. We have studied the problem of selecting encodings for instances of previously-unseen classes, a problem that is more challenging and arguably more realistic than the usual setting where training and test instances are drawn from the same set of problem classes. General instance features such as those provided by `fzn2feat` [4] perform well; however the introduction of features specific to linear integer and pseudo-Boolean constraints has enabled us to improve the quality of predictions. We present a machine learning method that performs well, and investigate several variations of it. We have also presented a thorough experimental analysis of the method, a comparison with AutoFolio, and an analysis of feature importance.

We intend to build on these results by considering other constraint types for which multiple SAT encodings exist. It may also be beneficial to expand the problem corpus to have a more even distribution of problem instances per class and to broaden the range of constraint models represented.

References

1 I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and V. Mayer-Eichberger. A New Look at BDDs for Pseudo-Boolean Constraints. *Journal of Artificial Intelligence Research*, 45:443–480, November 2012. doi:10.1613/jair.3653.

2 Ignasi Abío, Valentin Mayer-Eichberger, and Peter Stuckey. Encoding Linear Constraints into SAT. *arXiv:2005.02073 [cs]*, May 2020. arXiv:2005.02073.

3 Ignasi Abío, Valentin Mayer-Eichberger, and Peter J Stuckey. Encoding linear constraints with implication chains to CNF. In *International Conference on Principles and Practice of Constraint Programming*, pages 3–11. Springer, 2015. doi:10.1007/978-3-319-23219-5_1.

4 Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An enhanced features extractor for a portfolio of constraint solvers. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1357–1359, New York, NY, USA, March 2014. Association for Computing Machinery. doi:10.1145/2554850.2555114.

5 Carlos Ansótegui, Miquel Bofill, Jordi Coll, Nguyen Dang, Juan Luis Esteban, Ian Miguel, Peter Nightingale, András Z Salamon, Josep Suy, and Mateu Villaret. Automatic detection of at-most-one and exactly-one relations for improved SAT encodings of pseudo-boolean constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 20–36. Springer, 2019. doi:10.1007/978-3-030-30048-7.

6 Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New Encodings of Pseudo-Boolean Constraints into CNF. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, Lecture Notes in Computer Science, pages 181–194, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-02777-2_19.

7 Miquel Bofill, Jordi Coll, Peter Nightingale, Josep Suy, Felix Ulrich-Oltean, and Mateu Villaret. SAT encodings for Pseudo-Boolean constraints together with at-most-one constraints. *Artificial Intelligence*, 302:103604, January 2022. doi:10.1016/j.artint.2021.103604.

8 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*, pages 555–562, Melbourne, Australia, August 2017. International Joint Conferences on Artificial Intelligence Organization. doi:10.24963/ijcai.2017/78.

9 Miquel Bofill, Jordi Coll, Josep Suy, and Mateu Villaret. SAT encodings of pseudo-boolean constraints with at-most-one relations. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 112–128. Springer, 2019. doi:10.1007/978-3-030-19212-9.

10 Miquel Bofill, Jordi Coll, Josep Suy, and Mateu Villaret. An MDD-based SAT encoding for pseudo-Boolean constraints with at-most-one relations. *Artificial Intelligence Review*, 53(7):5157–5188, 2020. doi:10.1007/s10462-020-09817-6.

11 Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, October 2001. doi:10.1023/A:1010933404324.

12 Ewan Davidson, Özgür Akgün, Joan Espasa, and Peter Nightingale. Effective Encodings of Constraint Programming Models to SMT. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 143–159, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-58475-7_9.

13 Marijn Heule, Matti Jarvisalo, Martin Suda, Markus Iser, Tomáš Balyo, and Nils Froleyks. SAT competitions. URL: https://satcompetition.github.io/ [cited 22.02.2022].

14 Steffen Hölldobler, Norbert Manthey, and Peter Steinke. A Compact Encoding of Pseudo-Boolean Constraints into SAT. In Birte Glimm and Antonio Krüger, editors, *KI 2012: Advances in Artificial Intelligence*, Lecture Notes in Computer Science, pages 107–118, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-33347-7_10.

15 Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A Hierarchical Portfolio of Solvers and Transformations. In Helmut Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming*, Lecture Notes in Computer Science, pages 301–317, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-07046-9.

16 Saurabh Joshi, Ruben Martins, and Vasco Manquinho. Generalized Totalizer Encoding for Pseudo-Boolean Constraints. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 200–209, Cham, 2015. Springer International Publishing. `doi:10.1007/978-3-319-23219-5_15`.

17 Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. Automated Algorithm Selection: Survey and Perspectives. *Evolutionary Computation*, 27(1):3–45, March 2019. `doi:10.1162/evco_a_00242`.

18 Christophe Lecoutre and Olivier Roussel. XCSP3 Competition, 2019. URL: `http://www.cril.univ-artois.fr/XCSP19/` [cited 22.02.2022].

19 Marius Lindauer, Holger H. Hoos, Frank Hutter, and Torsten Schaub. AutoFolio: An Automatically Configured Algorithm Selector. *Journal of Artificial Intelligence Research*, 53:745–778, August 2015. `doi:10.1613/jair.4726`.

20 Peter Nightingale. Savile Row 1.9.0 Manual. URL: `https://savilerow.cs.st-andrews.ac.uk/index.html` [cited 22.02.2022].

21 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, October 2017. `doi:10.1016/j.artint.2017.07.001`.

22 F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

23 Philipp Probst, Marvin N. Wright, and Anne-Laure Boulesteix. Hyperparameters and tuning strategies for random forest. *WIREs Data Mining and Knowledge Discovery*, 9(3):e1301, 2019. `doi:10.1002/widm.1301`.

24 Mirko Stojadinović and Filip Marić. meSAT: Multiple encodings of CSP to SAT. *Constraints*, 19(4):380–403, October 2014. `doi:10.1007/s10601-014-9165-7`.

25 Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, June 2009. `doi:10.1007/s10601-008-9061-0`.

26 Helsinki Institute for Information Technology University of Helsinki, Tomáš Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Proceedings of SAT Competition 2021 : Solver and Benchmark Descriptions. In *Proceedings of SAT Competition 2021*. Department of Computer Science, University of Helsinki, 2021.

27 Neng-Fa Zhou and Håkan Kjellerstrand. Optimizing SAT encodings for arithmetic constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 671–686. Springer, 2017. `doi:10.1007/978-3-319-66158-2_43`.

Completeness Matters: Towards Efficient Caching in Tree-Based Synchronous Backtracking Search for DCOPs

Jie Wang[1]
College of Computer Science, Chongqing University, China

Dingding Chen[2]
College of Computer Science, Chongqing University, China

Ziyu Chen[3]
College of Computer Science, Chongqing University, China

Xiangshuang Liu
College of Computer Science, Chongqing University, China

Junsong Gao
College of Computer Science, Chongqing University, China

Abstract

Tree-based backtracking search is an important technique to solve Distributed Constraint optimization Problems (DCOPs), where agents cooperatively exhaust the search space by branching on each variable to divide subproblems and reporting the results to their parent after solving each subproblem. Therefore, effectively reusing the historical search results can avoid unnecessary resolutions and substantially reduce the overall overhead. However, the existing caching schemes for asynchronous algorithms cannot be applied directly to synchronous ones, in the sense that child agent reports the lower and upper bound rather than the precise cost of exploration. In addition, the existing caching scheme for synchronous algorithms has the shortcomings of incompleteness and low cache utilization. Therefore, we propose a new caching scheme for tree-based synchronous backtracking search, named Retention Scheme (RS). It utilizes the upper bounds of subproblems which avoid the reuse of suboptimal solutions to ensure the completeness, and deploys a fine-grained cache information unit targeted at each child agent to improve the cache-hit rate. Furthermore, we introduce two new cache replacement schemes to further improve performance when the memory is limited. Finally, we theoretically prove the completeness of our method and empirically show its superiority.

2012 ACM Subject Classification Computing methodologies → Cooperation and coordination

Keywords and phrases DCOP, Cache, Any-space Algorithms, Complete Search Algorithms

Digital Object Identifier 10.4230/LIPIcs.CP.2022.39

Supplementary Material *Software (Source Code)*: `https://github.com/czy920/RS`

1 Introduction

Distributed Constraint Optimization Problems (DCOPs) [15, 11] are a popular framework for multi-agent systems (MAS) where agents need to cooperate with each other to optimize a global objective. Owing to their excellent modeling ability, DCOPs have been widely applied in many real-world problems such as sensor network [9], task scheduling [18, 27], smart grid [12] and so on.

[1] Equal contribution.
[2] Equal contribution.
[3] Corresponding author.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 39; pp. 39:1–39:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Incomplete algorithms for DCOPs [31, 17, 10, 23, 22] aim to rapidly find an acceptable solution at the expense of sacrificing optimality, while complete algorithms ensure the optimal solution and can be generally divided into inference-based and search-based algorithms. DPOP [24] and Action_GDL [28] are typical inference-based complete algorithms which employ dynamic programming to solve DCOPs. However, they require a linear number of messages of exponential size which bring an excessive network load. Whereupon, ODPOP [25], MB-DPOP [26] and RMB-DPOP [6] were proposed to trade the number of messages for smaller memory consumption by propagating the dimension-limited utilities with the corresponding contexts iteratively. Besides, DPOP with function filtering [2] proposed to exploit utility bounds to reduce the size of messages.

Search-based complete algorithms perform distributed backtracking search to exhaust the search space and have a linear size of messages. Among them, tree-based complete search algorithms have attracted wider attention due to their high concurrency and can further be classified into synchronous and asynchronous. Given a context, tree-based complete synchronous search algorithms like NCBB [3], PT-FB [16] and HS-CAI [5] require each agent to report its search result after it has thoroughly explored its subproblem, while asynchronous ones like ADOPT [20] and BnB-ADOPT [29] allow each agent to report its search results solely based on its local view of its subproblem. To accelerate the search process, these algorithms utilize the upper and lower bounds to prune the search space. Accordingly, many pruning techniques like soft arc consistency [14], forward bounding procedure [16] and inference-based estimation [8, 1, 5] have arisen to tighten the lower bounds. On the other side, tree-based backtracking search requires agents to report their search results regarding their subproblems given a context. When receiving the same context, agents can effectively reuse the historical search results to avoid unnecessary resolutions and substantially reduce the overall overhead. Accordingly, any-space algorithms using historical exploration results to speed up the search process have been proposed. Matsui et al [19] proposed a caching scheme for ADOPT, named any-space ADOPT, where each agent caches the upper and lower bounds of its subproblem given a context. Yeoh et al [30] further improved the scheme by introducing three cache replacement schemes to boost the cache utilization. However, these caching schemes for asynchronous algorithms are not suitable for synchronous algorithms since synchronous algorithms need to report the optimal cost for each subproblem rather than the bounds. Although tree-based complete synchronous search algorithms have been widely studied, there are few studies on cache schemes for them. Any-space NCBB [4], the only caching scheme for synchronous algorithms now, presented a caching scheme for NCBB where a caching unit is introduced to store the search result regarding a given context for each agent. Unfortunately, the scheme fails to consider the impact of pruning on cached results and leads to the incompleteness. And also, the scheme matches by comparing all the separators of local agent, which brings the low cache-hit rate. Therefore, the cache scheme in any-space NCBB cannot be applied to existing tree-based complete synchronous search algorithms. To this end, we present a complete and effective caching scheme for tree-based complete synchronous search algorithms and our main contributions are listed as follows:

- We systematically analyze the cause for the incompleteness of the existing synchronous caching scheme, and provide a solution which compares the historical upper bound with the current upper bound to determine whether the historical results can be reused.
- To improve the cache-hit rate, we introduce a fine-grained cache information unit targeted at each child agent which allows each agent to independently reuse the historical results of subproblem rooted at its child. Along with the solution to the incompleteness, we propose our Retention Scheme (RS) accordingly, which is suitable for all tree-based complete synchronous search algorithms.

- In addition, to improve the performance of our RS when the memory is limited, we propose two heuristic cache replacement schemes which consider the characteristics of the cached information units and synchronous algorithms, respectively.
- We theoretically show the completeness of RS, and the experimental results demonstrate it improves state-of-the-art tree-based complete synchronous search algorithms on all the metrics in most cases.

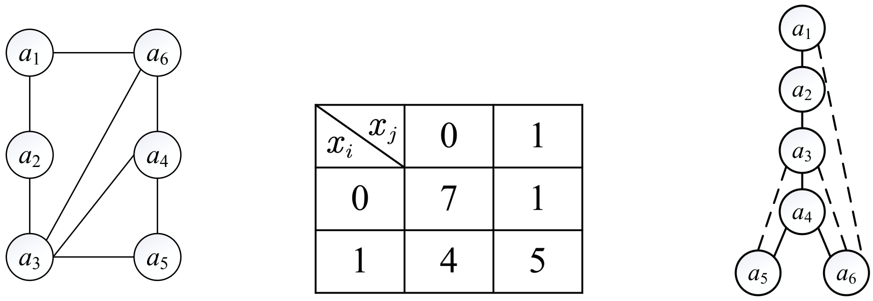
2 Background

In this section, we first introduce the preliminaries including DCOPs, pseudo tree, tree-based complete synchronous search algorithms and the caching scheme in any-space NCBB.

2.1 Distributed Constraint optimization Problems

A distributed constraint optimization problem [20] can be defined by a tuple $\langle A, X, D, F\rangle$ where

- $A = \{a_1, a_2, \cdots, a_n\}$ is a set of agents.
- $X = \{x_1, x_2, \cdots, x_m\}$ is a set of variables.
- $D = \{D_1, D_2, \cdots, D_m\}$ is a set of finite variable domains, each variable x_i taking a value in D_i.
- $F = \{f_1, f_2, \cdots, f_q\}$ is a set of constraints, each of which $f_i : D_{i_1} \times D_{i_2} \times \cdots D_{i_k} \to \mathbb{R}_{\geqslant 0}$ denotes the non-negative cost for each assignment combination of $x_{i_1}, x_{i_2}, \cdots x_{i_k}$.



x_i \ x_j	0	1
0	7	1
1	4	5

(a) Constraint graph. **(b)** Constraint matrix. **(c)** Pseudo tree.

Figure 1 An example of a DCOP and its pseudo tree.

For the sake of simplicity, the paper assumes that each agent holds exactly one variable (i.e., $n = m$) and all constraints are binary relations (i.e., $f_{ij} : D_i \times D_j \to \mathbb{R}_{\geqslant 0}$). Thus, the term “agent” and “variable” could be used interchangeably. Without loss of generality, a DCOP seeks an assignment to all the variables that minimizes the total cost. Formally,

$$X^* = \underset{d_i \in D_i, d_j \in D_j}{\arg\min} \sum_{f_{ij} \in F} f_{ij}\left(x_i = d_i, x_j = d_j\right)$$

In general, a DCOP can be visualized by a constraint graph where vertexes represent the agents and edges represent the constraints, respectively. Fig.1(a) presents a DCOP with six variables and eight constraints. For simplicity, the domain size of each variable is two and all constraints are identical as shown in Fig.1(b) where $i < j$.

2.2 Pseudo Tree

A pseudo tree [13, 7] is a partial ordering among agents, which can be generated by depth-first search (DFS) traversal to a constraint graph. It has the property that different branches are independent, and categorizes its constraints into tree edges and pseudo edges (i.e., non-tree edges). Fig.1(c) presents a possible pseudo tree deriving from Fig.1(a) where tree edges and pseudo edges are denoted by solid and dashed lines, respectively. For an agent a_i, its neighbors can be categorized into its parent $P(a_i)$, children $C(a_i)$ and pseudo parents $PP(a_i)$, according to their positions in the pseudo tree and the type of edges they connect through. Each agent directs its message passing and traverses the solution space through such a (pseudo) parent-child relationship. More precisely, they can be formally defined as follows:

- $P(a_i)$ is the ancestor connecting with a_i through a tree edge (e.g., $P(a_4) = a_3$ in Fig.1(c)).
- $C(a_i)$ is the set of descendants connecting with a_i through tree edges (e.g., $C(a_4) = \{a_5, a_6\}$ in Fig.1(c)).
- $PP(a_i)$ contains the ancestors that connect with a_i through pseudo edges (e.g., $PP(a_5) = \{a_3\}$ in Fig.1(c)).

For succinctness, we also adopt the following notations.

- $AP(a_i)$ is the set of all (pseudo) parents of a_i. i.e., $AP(a_i) = PP(a_i) \cup \{P(a_i)\}$ (e.g., $AP(a_5) = \{a_3, a_4\}$ in Fig.1(c)).
- $Anc(a_i)$ is the set of ancestors of a_i (e.g., $Anc(a_4) = \{a_1, a_2, a_3\}$ in Fig.1(c)).
- $Desc(a_i)$ is the set of descendants of a_i (e.g., $Desc(a_3) = \{a_4, a_5, a_6\}$ in Fig.1(c)).
- $Sep(a_i)$ [11] is the separator set of a_i, comprising the ancestors that are constrained with agents in $\{a_i\} \cup Desc(a_i)$ (e.g., $Sep(a_4) = \{a_1, a_3\}$ in Fig.1(c)), i.e.,

$$Sep(a_i) = \{a_j \in Anc(a_i) \,|\, \exists a_k \in \{a_i\} \cup Desc(a_i), s.t., a_j \in AP(a_k)\}$$

2.3 Tree-based Complete Synchronous Search Algorithms

Tree-based complete synchronous search algorithms perform a branch-and-bound search on a pseudo tree to exhaust the search space. Specifically, each agent in the algorithms obtains the optimal cost of subproblem rooted at itself under the current partial assignment, and prunes to avoid unnecessary exploration by exploiting the bounds including the lower and upper bounds of its subproblem. In fact, most of the existing synchronous algorithms (e.g., NCBB, PT-FB and HS-CAI etc.) can be regarded as variants of SBB [15] on a pseudo tree, which utilize different techniques to obtain a tighter lower bound to improve pruning. Algorithm 1 presents the sketch of the implementation of SBB on a pseudo tree (named TreeBB) to describe the general framework of tree-based complete synchronous search algorithms.

In TreeBB, each agent a_i needs to maintain the following data structures.

- Cpa_i refers to the current partial assignment which contains all the assignments to $Anc(a_i)$.
- ub_i is the upper bound of its subproblem under Cpa_i.
- $opt_i^c(d_i)$ is the optimal cost of its child $a_c \in C(a_i)$ for $d_i \in D_i$, which is set to infinite if $Cpa_i \cup \{x_i, d_i\}$ is infeasible under ub_i.
- $sendub_c(d_i)$ is the upper bound sent to its child a_c for $d_i \in D_i$, i.e.,

$$sendub_c(d_i) = ub_i - \sum_{a_j \in AP(a_i)} f_{ij}(d_i, Cpa_i(a_j)) - \sum_{a_j \in C(a_i) \wedge opt_i^j \neq null} opt_i^j(d_i) \qquad (1)$$

Algorithm 1 TreeBB for a_i.

```
    When Initialization ():
 1      if $a_i$ is the root agent then
 2          InitializeVariables()
 3          $d_i \leftarrow$ the first element in $\tilde{D}_i^c, Cpa_i \leftarrow \{(x_i, d_i)\}$
 4          send CPA($Cpa_i, \infty$) to $\forall a_c \in C(a_i)$
    When received CPA ($Cpa_i, ub_i$) from $P(a_i)$:
 5      store $\{Cpa_i, ub_i\}$
 6      if $a_i$ is a leaf agent then
 7          SendBacktrack ()
 8      else
 9          InitializeVariables()
10          ExploreValue ($a_c$), $\forall a_c \in C(a_i)$
    When received BACKTRACK ($opt^*$) from $a_c \in C(a_i)$:
11      $d_i \leftarrow SrchVal_i^c, \tilde{D}_i^c \leftarrow \tilde{D}_i^c \backslash \{d_i\}$ ,$opt_i^c(d_i) \leftarrow opt^*$
12      if $a_i$ has received all BACKTRACK messages from $C(a_i)$ for $d_i$ then
13          $ub_i \leftarrow min(ub_i, lb_i(d_i))$
14      ExploreValue ($a_c$)
    Function InitializeVariables ():
15      $\tilde{D}_i^c \leftarrow D_i, \forall a_c \in C(a_i)$
16      $opt_i^c(d_i) \leftarrow null, \forall a_c \in C(a_i), d_i \in D_i$
    Function ExploreValue ($a_c$):
17      $d_i \leftarrow$ NextFeasibleAssignment ($a_c$)
18      if $\tilde{D}_i^c = \emptyset, \forall a_c \in C(a_i)$ then
19          if $a_i$ is the root agent then
20              Algorithm terminates.
21          else
22              SendBacktrack ()
23      else
24          $SrchVal_i^c \leftarrow d_i, Cpa_c \leftarrow Cpa_i \cup \{(x_i, d_i)\}$
25          send CPA($Cpa_c, sendub_c(d_i)$) to $a_c$
    Function NextFeasibleAssignment ($a_c$):
26      $d_i \leftarrow$ the first element in $\tilde{D}_i^c$
27      while $d_i \neq null \wedge lb_i\,(d_i) \geqslant ub_i$ do
28          $\tilde{D}_i^c \leftarrow \tilde{D}_i^c \backslash \{d_i\}$, $opt_i^c(d_i) \leftarrow \infty$, $d_i \leftarrow$ the first element in $\tilde{D}_i^c$
29      return $d_i$
    Function SendBacktrack ():
30      $opt^* \leftarrow \min_{d_i \in D_i} lb_i\,(d_i)$
31      Send BACKTRACK($opt^*$) to $P(a_i)$
```

- $lb_i(d_i)$ is its lower bound for $d_i \in D_i$, i.e.,

$$lb_i(d_i) = \sum_{a_j \in AP(a_i)} f_{ij}(d_i, Cpa_i(a_j)) + \sum_{a_c \in C(a_i) \wedge opt_i^c \neq null} opt_i^c\,(d_i) \tag{2}$$

- opt^* is the current optimal cost of its subproblem under Cpa_i, i.e.,

$$opt^* = \min_{d_i \in D_i} lb_i\,(d_i) \tag{3}$$

TreeBB begins with the root agent sending the first value in its domain to its children (line 1-4). When an agent a_i receives a CPA message from its parent, it first stores the partial assignment Cpa_i and the upper bound ub_i (line 5), then initializes the search domain $\tilde{D}_i^c$ and opt_i^c (line 9,15-16). Next, a_i finds the first feasible assignment, i.e., the first assignment d_i such that $lb_i(d_i) < ub_i$ (line 26-29) and explores the corresponding subproblem (line 10, 17). If such an assignment exists, a_i sends a CPA message together with $sendub_c(d_i)$ calculated by Eq.(1) to its children (line 23-25). Otherwise, it sends a BACKTRACK message with the optimal cost opt^* to its parent (line 18-22, line 30-31).

When a_i receives a BACKTRACK message for d_i from its child a_c, it updates the corresponding optimal cost $opt_i^c(d_i)$ with the actual cost opt^* reported by a_c and continues to explore the subproblem corresponding to its next feasible assignment (line 11, 14). If a_i

has received all the BACKTRACK messages for d_i from its children, it updates the current upper bound for its subproblem (line 12-13). Finally, TreeBB terminates after the root agent exhausts its search domain and a global optimal cost will be got (line 18-20).

2.4 Caching Scheme in Any-space NCBB

To better utilize the historical search results, any-space NCBB proposed that agent a_i caches the historical cost and the corresponding assignments of $Sep(a_i)$, called $context_i$. To this end, any-space NCBB constructs a caching information unit I, a map set $\langle context_i, result\rangle$, to store these historical costs and *contexts* for future use. Here, we refer to such a caching scheme as OCS. Taking Fig.1(c) for example, assume that a_4 has obtained the current optimal cost opt^* of its subproblem. The opt^* is valid as long as a_1 and a_3 do not change their assignments. Accordingly, a_4 could directly get the opt^* for its subproblem from the cache and backtracks since re-exploring is unnecessary.

We next illustrate how OCS works when applied to TreeBB. There are two modifications to the original for implementing the scheme. First, before a_i sends a BACKTRACK message (line 22), it stores the current optimal cost opt^* along with the corresponding $context_i$ into its cache. Second, before exploring its subproblem (line 10), a_i looks up whether the current *context* is already in the cache. If it is, a_i sets the search domain $\tilde{D}_i^c$ to *null* and sends a BACKTRACK message with the opt^* in the cache to its parent. Unfortunately, such a scheme could lead to the incompleteness, which will be illustrated in detail in Subsection 3.1.

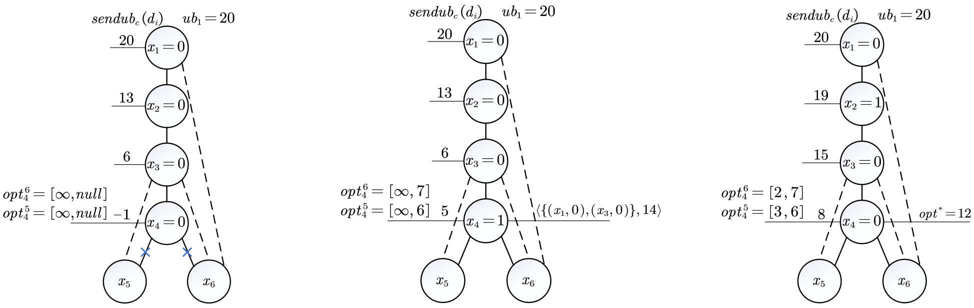
3 Proposed Method

In this section, we present a new caching scheme, named Retention Scheme (RS). We begin with the motivation of our work and then present the details. Finally, we give two cache replacement schemes.

3.1 Motivation

Actually, in OCS, agent a_i caches the optimal cost opt^* under a given ub_i. However, when pruning happens with ub_i, agent a_i could carry out an incomplete exploration, and caches the opt^* and its $context_i$. Clearly, such a cost cannot guarantee to be optimal since its search subspace is not exploited thoroughly yet. Considering a condition where pruning is not carried out, a_i could find a better opt^* by exploring the search space pruned before. Unfortunately, such a condition could happen since ub_i calculated by Eq.(1) may increase with the different assignments of $Anc(a_i)$. Note that, in the same $context_i$, the upper bound for pruning may be different since $Sep(a_i)$ is only a subset of $Anc(a_i)$. As a result, if OCS directly uses the opt^* obtained from a pruned search space from its cache, the completeness of the algorithm can not be guaranteed.

To illustrate the issue, based on TreeBB, we take Fig.1 as an example. Here, we mainly focus on agent a_4 and compare the difference when OCS is applied. Assume that the upper bound for $x_1 = 0$ is 20, thus a_4 could receive a CPA message containing a current partial assignment $Cpa_4 = \{(x_1, 0), (x_2, 0), (x_3, 0)\}$ and an upper bound $ub_4 = 6$ as illustrated in Fig.2(a). Next, since $lb_4(0) = 7$ calculated by Eq.(2) is greater than its $ub_4 = 6$, a_4 prunes the corresponding search space and sets the related cost to infinity, i.e., $opt_4^5(0) = opt_4^6(0) = \infty$. Subsequently, a_4 chooses its next feasible assignment $x_4 = 1$, and stores the best costs from its children into $opt_4^5(1)$ and $opt_4^6(1)$, respectively. Then, the opt^* is calculated according to Eq.(3) and caches together with the corresponding $context_4$ as $\langle\{(x_1, 0), (x_3, 0)\}, 14\rangle$, shown in



(a) **(b)** **(c)**

Figure 2 An example of OCS-based algorithm to show its incompleteness.

Fig.2(b). Now, if a_4 receives a new CPA message containing $Cpa_4 = \{(x_1, 0), (x_2, 1), (x_3, 0)\}$, in OCS, a_4 could directly use the historical result in the cache since the $context_4$ is identical even in difference Cpa_4. However, the better cost for the current Cpa_4 is 12 which is depicted in Fig.2(c), since a_4 explores the search space pruned before under the larger $ub_4 = 15$. As a result, the completeness could be impaired when OCS is applied.

Additionally, in OCS, the cache is accessed only when the current $context_i$ is identical to the items that has already stored in the information unit I. In the worst cases, the cache does not work in synchronous algorithms if $|Anc(a_i)| = |Sep(a_i)|$, as the traversal for the combinations of $context_i$ is ordered. As a result, the next $context_i$ is impossible to be identical to the cached items. However, for each child a_c, its corresponding $context_c$ is only the subset of that of its parent $context_i$, which means the cached costs may also be valid under a different $context_i$ for a_c. Taking a_4 in Fig.2 as an example, the $opt_4^5(1) = 6$ is valid as long as a_3 does not change its assignment, regardless of the assignment of a_1. Therefore, the OCS makes little use of the historical results and leads to a low cache utilization.

3.2 Retention Scheme

In order to solve the issues mentioned above, we propose the Retention Scheme (RS) to ensure the completeness and improve the cache utilization.

For the first issue, given a $context_i$ we additionally record the current upper bounds a_i received into the information unit I for judging whether the cached item is reliable. Specifically, if a newly received ub_i is greater than the stored one under the same $context_i$, the cached opt^* could be unreliable as pointed out in Subsection 3.1, and re-exploration should be carried out. Specially, if the current opt^* is obtained by exhausting the search space of its children, such opt^* is sure to be reliable. Here, we denote such opt^* as the $realcost$ under its corresponding $context_i$. Obviously, if a_i is the leaf agent, the opt^* is the $realcost$. Besides, if the opt^* is less than the received ub_i, the opt^* is the $realcost$, see Lemma 1. Thus, once the current opt^* is detected to be the $realcost$, we set the stored ub_i to be ∞ to avoid unnecessary re-exploration. So far, we have already guaranteed the completeness since only the $realcost$ can be obtained for use.

Next, we illustrate how we solve the second issue. Here, we adopt a fine-grained cache strategy by splitting the $context_i$ in OCS into $context_c$ for $\forall a_c \in C(a_i)$, where $context_c$ only includes the assignments of $Sep(a_c)$ for each a_c. Eq.(4) gives the relationship between $Sep(a_c)$ and $Sep(a_i)$.

$$\bigcup_{a_c \in C(a_i)} Sep\,(a_c) \setminus \{a_i\} \;\subseteq Sep\,(a_i) \tag{4}$$

So, different from OCS, we propose to use I to only store the information map related to its children, i.e., $\langle context_c, \langle opt_c, sub_c \rangle \rangle$, where opt_c and sub_c is the best cost reported by its child a_c and the corresponding upper bound for each child under the current $context_c$, respectively.

Algorithm 2 Retention Scheme for a_i.

```
When Initialization:
1       allocate memory space $k$ for each child by Eq.(5)
Function InitVariable ():
2       foreach $a_c \in C(a_i)$ do
3           foreach $d_i \in D_i$ do
4               $context_c \leftarrow Cpa_i(Sep(a_c)) \cup \{(x_i, d_i)\}$
5               if $IUCache_i^c\,(context_c) \neq null$ then
6                   $\{opt_i^c(d_i), sub_i^c(d_i)\} \leftarrow IUCache_i^c\,(context_c)$
7               else
8                   $opt_i^c(d_i) \leftarrow null$,$sub_i^c(d_i) \leftarrow 0$
9           $\tilde{D}_i^c \leftarrow \{d_i | d_i \in D_i, sub_i^c(d_i) \neq \infty\}$
Function NextFeasibleAssignment ($a_c$):
10      $d_i \leftarrow$ the first element in $\tilde{D}_i^c$
11      while $(d_i \neq null \wedge lb_i\,(d_i) \geqslant ub_i) \vee (d_i \neq null \wedge sendub_c(d_i) \leqslant sub_i^c(d_i))$ do
12          if $lb_i\,(d_i) \geqslant ub_i$ then
13              $\{opt_i^c(d_i), sub_i^c(d_i)\} \leftarrow \{\infty, 0\}$
14          $\tilde{D}_i^c \leftarrow \tilde{D}_i^c \backslash \{d_i\}$
15          $d_i \leftarrow$ the first element in $\tilde{D}_i^c$
16      return $d_i$
Function Backtrack ():
17      FillIUCache ()
18      $opt^* \leftarrow \min_{d_i \in D_i} lb_i\,(d_i)$
19      if $opt^*$ is a realcost then
20          send BACKTRACK $(opt^*, \infty)$ to $P(a_i)$
21      else
22          send BACKTRACK $(opt^*, ub_i)$ to $P(a_i)$
Function FillIUCache ():
23      foreach $a_c \in C(a_i)$ do
24          if $IUCache_i^c\,(context_c) \neq null$ then
25              foreach $d_i \in D_i$ do
26                  if $sub_i^c(d_i) > 0$ then
27                      $IUCache_i^c\,(context_c) \leftarrow \{opt_i^c\,(d_i)\,, sub_i^c\,(d_i)\}$
28          else if $mem_i^c > 0$ then
29              $IUCache_i^c\,(context_c) \leftarrow \{opt_i^c\,(d_i)\,, sub_i^c\,(d_i)\}\,, \forall d_i \in D_i$
30              $mem_i^c \leftarrow mem_i^c - |D_i|$
31          else
32              call certain page replacement procedure
```

Besides, some modifications should be done to implement the strategy appropriately. Specifically, each agent a_i additionally attaches the upper bound to the BACKTRACK message to its parent agent and maintains a list $sub_i^c(d_i), \forall d_i \in D_i, a_c \in C(a_i)$ to record the upper bound sent from a_c and update it if needed. Moreover, for each child a_c, a_i also maintains some additional data structures, which includes an $IUCache_i^c$ to store all information units and a mem_i^c to record the remaining memory space.

Next, we will detail how to implement the RS in TreeBB. Algorithm 2 presents the sketch of RS, and we only describe the difference from TreeBB. Specifically, each agent a_i starts by allocating memory for each child. Here, we set the maximum memory space with a parameter k for each agent, which means a_i can cache $|D_i|^k$ information units I at most and allocate the memory for each child a_c according to Eq.(5) if a_i has more than one child (line 1). The maximum memory function $|D_i|^k$ is widely used in some memory-bounded algorithms, and other functions (e.g., a constant value or $\rho * |D|^{|Sep(a_i)|}$ where $\rho \in (0, 1]$) can also be adopted.

As can be seen from Eq.(5), agent a_i allocates as much memory space as possible for a_c with small $|Sep(a_c)|$, but the initialized memory space mem_i^c should not exceed the maximum memory requirement, i.e.,$|D_i|^{|Sep(a_c)|}$. It is based on an intuitive idea that agent a_i with small $|Sep(a_c)|$ has fewer combinations for $context_c$, which could lead to a higher cache-hit rate.

$$mem_i^c \leftarrow \min\left(\frac{\left(\sum_{a_{c'}\in C(a_i)}|Sep(a_{c'})|-|Sep(a_c)|\right)*|D_i|^k}{(|C(a_i)|-1)*\sum_{a_{c'}\in C(a_i)}|Sep(a_{c'})|}, |D_i|^{|Sep(a_c)|}\right), \forall a_c \in C(a_i) \quad (5)$$

Next, the execution phase of TreeBB starts. When a_i receives a CPA message from its parent, for each child a_c, it judges whether the current $context_c$ is already in the cache (line 2–4). Specifically, if it hits, a_i obtains the corresponding historical results in the information unit cache $IUCache_i^c$ and initializes opt_i^c and sub_i^c (line 5–6). Otherwise, a_i sets $sub_i^c(d_i)$ to 0 and explores its child a_c when $sendub_c(d_i) > 0$ (line 7–8). Then, when the obtained $sub_i^c(d_i)$ is ∞, which means a *realcost* for d_i has already been obtained, then d_i is removed from $\tilde{D}_i^c$ (line 9). After that, when a_i selects the next feasible assignment to explore for child a_c, our RS gives an additional pruning judgement (line 11–15). That is, a value d_i is removed from the search domain $\tilde{D}_i^c$ if a new $sendub_c(d_i) \leqslant sub_i^c(d_i)$ since a_i still cannot get a better cost under the $sendub_c(d_i)$.

Before a_i sends a BACKTRACK message, for each child a_c, it stores the current optimal cost $opt_i^c(d_i)$ and upper bound $sub_i^c(d_i)$ with the corresponding $context_c$ into its cache (line 17). Specifically, if the $context_c$ already exists in $IUCache_i^c$, a_i updates the cached results directly (line 23–27). Here, to make the cached results more effective, a_i performs the update only when $sub_i^c(d_i) > 0$. If the $context_c$ is not in the $IUCache_i^c$ and the remaining memory space mem_i^c is greater than 0, a_i stores the corresponding results as an information unit I and reduces the remaining memory space mem_i^c (line 28-30). If a_i does not have enough cache capacity to store a new information unit, it will perform certain cache replacement procedure (line 31–32). Then, if the current optimal cost opt^* is identified as the *realcost*, a_i replaces the upper bound ub_i with infinity in BACKTRACK messages to avoid unnecessary re-exploration (line 19–20). Otherwise, a_i needs to report the current upper bound (line 21–22).

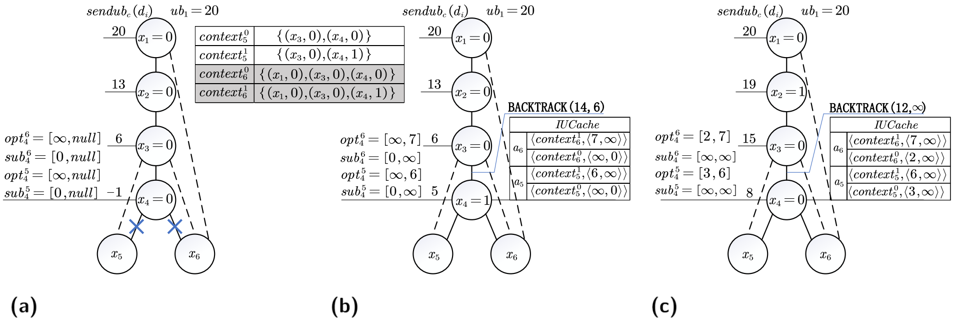

$sendub_c(d_i)$
$ub_1 = 20$
20
13
6
−1
$x_1 = 0$
$x_2 = 0$
$x_3 = 0$
$x_4 = 0$
x_5
x_6
$opt_4^6 = [\infty, null]$
$sub_4^6 = [0, null]$
$opt_4^5 = [\infty, null]$
$sub_4^5 = [0, null]$
$context_5^0$ $\{(x_3,0),(x_4,0)\}$
$context_5^1$ $\{(x_3,0),(x_4,1)\}$
$context_6^0$ $\{(x_1,0),(x_3,0),(x_4,0)\}$
$context_6^1$ $\{(x_1,0),(x_3,0),(x_4,1)\}$
(a)
$sendub_c(d_i)$
$ub_1 = 20$
20
13
6
5
$x_1 = 0$
$x_2 = 0$
$x_3 = 0$
$x_4 = 1$
x_5
x_6
$opt_4^6 = [\infty, 7]$
$sub_4^6 = [0, \infty]$
$opt_4^5 = [\infty, 6]$
$sub_4^5 = [0, \infty]$
BACKTRACK (14, 6)
IUCache
a_6 $\langle context_6^1, \langle 7, \infty\rangle\rangle$
$\langle context_6^0, \langle \infty, 0\rangle\rangle$
a_5 $\langle context_5^1, \langle 6, \infty\rangle\rangle$
$\langle context_5^0, \langle \infty, 0\rangle\rangle$
(b)
$sendub_c(d_i)$
$ub_1 = 20$
20
19
15
8
$x_1 = 0$
$x_2 = 1$
$x_3 = 0$
$x_4 = 0$
x_5
x_6
$opt_4^6 = [2, 7]$
$sub_4^6 = [\infty, \infty]$
$opt_4^5 = [3, 6]$
$sub_4^5 = [\infty, \infty]$
BACKTRACK (12, ∞)
IUCache
a_6 $\langle context_6^1, \langle 7, \infty\rangle\rangle$
$\langle context_6^0, \langle 2, \infty\rangle\rangle$
a_5 $\langle context_5^1, \langle 6, \infty\rangle\rangle$
$\langle context_5^0, \langle 3, \infty\rangle\rangle$
(c)

Figure 3 An example of RS-based algorithm to show its completeness.

To illustrate the effect of RS, we take Fig.1 as an example. Here, we mainly focus on agent a_4 and points out the difference when RS is applied based on Fig.3. Similar to the example in Fig.2(a), we assume the upper bound for $x_1 = 0$ is 20, a_4 receives the same CPA message and prunes the search space corresponding to 0, as illustrated in Fig.3(a). However, different from OCS, the RS additionally constructs sub_4^6 and sub_4^5 and set $sub_4^5(0) = sub_4^6(0) = 0$ (line 12–13). Then, a_4 explores its next feasible assignment $x_4 = 1$ and stores the best costs and upper bound sent from its children, respectively. Note that, a_4 sets $sub_4^5(1) = sub_4^6(1) = \infty$ since a_5 and a_6 are leaf agents and the *realcosts* ($opt_4^5(1) = 6$ and $opt_4^6(1) = 7$) are reported (line 19–20). After that, a_4 caches these results together with the corresponding *contexts* (shown in the table) as information units, and sends a BACKTRACK message including $opt^* = 14$ and $ub_i = 6$ to its parent a_3, shown in Fig.3(b). Next, when a_4 receives a new CPA message containing $Cpa_4 = \{(x_1, 0), (x_2, 1), (x_3, 0)\}$ like the example in Fig.2(c), different from OCS which just uses the cached cost and sends a BACKTRACK message with the suboptimal cost, a_4 re-explores the search space corresponding to $x_4 = 0$ since $sendub_c(0) = 8$ is greater than $sub_4^5(0) = sub_4^6(0) = 0$ (line 11), and for $x_4 = 1$, it directly reuses the *realcost*s stored in cache before. Then, a_4 obtains the *realcosts* for its children by exhausting the search space and updates *IUCache* accordingly (line 17, 23-27), shown as Fig.3(c). At last, a_4 calculates $opt^* = 12$ which is a *realcost* and sends it with an infinite upper bound through a BACKTRACK message to its parent a_3 (line 19–20). It can be seen that the RS successfully guarantees the completeness and brings higher cache utilization compared to OCS.

3.3 Cache Replacement Scheme

To better make use of the cached items with limited memory, a cache replacement scheme is necessary since such a good scheme can often bring higher cache utilization and improve the performance of algorithms. Yeoh et al [30] proposed three cache replacement schemes for asynchronous algorithms, including MaxPriority, MaxEffort and MaxUtility Scheme. All these schemes allow each agent to make decisions according to heuristics constructed by reordering the cached information units. However, none of them can be applied to synchronous algorithms directly due to the fact that the valid information to cache is actually different between synchronous and asynchronous algorithm. Besides, for synchronous algorithms, there is no other relevant research on cache replacement schemes reported before. Based on the background, we introduce two feasible cache replacement schemes including UB and SYS to match our RS for synchronous algorithms.

Specifically, we construct a heuristic scheme named UB in which a_i sorts the cached information units according to sub_c, which allows new information units to preempt the unit with the smallest sub_c in the cache. It is because the larger the sub_c, the more likely the opt_c regarding it is the *realcost*.

In addition, we propose another cache replacement scheme named SYS by carrying out the FIFO (First-In-First-Out) scheme only when a specific agent changes its value. It gives full play to the advantages of the FIFO scheme in synchronous algorithms, and avoids the disadvantages caused by frequent cache replacement. Before introducing the SYS scheme, we first give some notations as follows.

- $H(a_i)$ is the height of a_i on the pseudo tree. In particular, the height of root agent is 0.
- $SepQ_i^c$ is a queue containing all the $Sep(a_c)$ agents for $a_c \in C(a_i)$, which is sorted by the height in a reverse order, i.e., $H\left(SepQ_i^c\left(m\right)\right) > H\left(SepQ_i^c\left(m+1\right)\right)$ where m is the index of $SepQ_i^c$. Taking Fig.1(c) as an example, $SepQ_4^6 = \{a_4, a_3, a_1\}$.

In synchronous algorithms, each agent changes its assignment orderly according to its height on a pseudo tree. That is, the root agent changes its value far less than the leaf agents, which also means an agent with larger index in $SepQ_i^c$ changes its assignment less frequently. Hence, it seems that the traditional FIFO scheme is suitable for synchronous algorithms. Consider a_4 in Fig.1(c), when a_1 changes to its next assignment, i.e., $x_4 = 1$, the items with $context_6 = \{(x_1, 0), (x_3, d_3), (x_4, d_4)\}, \forall d_3 \in D_3, d_4 \in D_4$ cached in information unit cache *IUCache* actually expire, since the next $context_6$ is no longer similar to that a_4 cached before for a_6. Therefore, if a_4 adopts the FIFO scheme, it only needs $|D_3||D_4|$ memory for a_6 to cache the reported results. However, naively using the FIFO scheme might bring some issues in some situations. Still take a_4 in Fig.1(c) as an example and assume the allocated memory mem_4^5 for its child a_5 is limited to $|D_4|$. If a_4 directly uses the FIFO scheme, it starts to replace the cached items under $x_3 = 0$ once a_3 changes to 1. However, it is clear that when a_2 changes to its next value, the items a_4 cached under $x_3 = 0$ is still valid, but they have already been replaced by that under $x_3 = 1$. As a result, according to the FIFO scheme, a_4 may replace the cached items frequently, leading to the fact that items always are unused before replaced. Therefore, the SYS scheme aims to overcome the issue by finding the first agent a_j satisfying Eq.(6) for each child a_c and then discarding the search results directly if $a_j = null$ or the assignment of a_j in the current $context_c$ is consistent with that in the cached items. Otherwise, a_i utilizes the FIFO scheme to replace the cached items.

$$a_j = SepQ_i^c\,(m) \qquad s.t.\ P\left(SepQ_i^c\,(m-1)\right) \neq SepQ_i^c\,(m)\,, m \geqslant \lfloor \log_{|D_i|} mem_i^c \rfloor \tag{6}$$

Still take Fig.1(c) as an example and assume the initialized memory space $mem_4^5 = mem_4^6 = |D_4|$. Different from that in the FIFO scheme where a_4 always replaces the new $context_i$ for the cached ones when the *IUCache* is full, in SYS a_4 does not replace the cached items and directly discards the current search results for a_5 since $a_j = null$. Similarly, for child a_6, a_4 does not replace either until $a_j(a_1)$ changes its assignment.

4 Theoretical Results

In this section, we prove the completeness of the Retention Scheme (RS), and give the complexity analysis of the proposed method.

▶ **Lemma 1.** *The optimal cost opt^* reported by a_i is the realcost if $opt^* < ub_i$.*

Proof. Assuming that the opt^* reported by a_i is not the *realcost*, there must exist a *realcost*, called $cost^*$, which satisfies $cost^* < opt^*$. Therefore, the $cost^*$ must have been pruned since only the current best cost opt^* found so far is reported. However, in tree-based complete synchronous search algorithms, only when $cost^* \geqslant ub_i$, pruning is carried out, which leads to a contradiction to the condition $opt^* < ub_i$ in lemma 1. Thus, Lemma 1 is proved. ◀

▶ **Lemma 2.** *Given a same $context_i$, assume that a_i gets a suboptimal cost under ub_i. If the newly-received upper bound ub_i' for a_i is less than ub_i, i.e.,$ub_i' < ub_i$, a_i still gets a suboptimal cost.*

Proof. According to Lemma 1, the suboptimal cost opt^* should satisfy $opt^* \geqslant ub_i$ owing to opt^* is not a *realcost*. Therefore, we have $opt^* \geqslant ub_i > ub_i'$, which indicates that a *realcost* can not be obtained by exploring with ub_i' either. Thus, Lemma 2 is proved. ◀

▶ **Theorem 3.** *The RS is complete.*

Proof. According to Lemma 1 and Lemma 2, the RS only utilizes the *realcosts* and the additional pruning judgement it adopts does not affect the exploration for such *realcosts*. Thus, the RS is complete. ◀

It is worth noting that the completeness of the RS will not be affected by the arity of constraint functions as it is related only to the cached results of historical exploration. Besides, the cache replacement schemes do not hurt the completeness of the RS as they only determine which results explored should be stored in the cache.

4.1 Complexity

When applied to existing tree-based complete synchronous search algorithms, the RS does not introduce any new messages, and it only attaches the current upper bound to a BACKTRACK message which only requires linear memory.

For each agent a_i, assuming it has enough memory to store all the search results, it requires $|D_i|^{|Sep(a_c)|}$ memory to cache the items for its child $a_c \in C(a_i)$. Therefore, the overall consumption is $O\left(|C\left(a_i\right)||D_i|^{Sep}\right)$, where $Sep = \max_{a_c \in C(a_i)} |Sep(a_c)|$. On the other hand, for each agent a_i, it traverses the information unit cache $IUCache$ to map for its child $a_c \in C(a_i)$, which requires a linear time complexity. So, the entire time complexity is $O(|C(a_i)|)$.

Besides, for the cache replacement schemes, UB requires $O\left(|C(a_i)||D_i|^{Sep} \log |D_i|^{Sep}\right)$ time complexity to perform quick sort for information units, while for SYS it only needs $O(1)$ to calculate the specific agent by Eq.(6).

5 Empirical Evaluation

In this section, we apply the RS to state-of-the-art tree-based complete synchronous search algorithms, and compare them against the originals. Then, when the memory is limited, we investigate the effects of different cache replacement schemes on the RS-based algorithms.

5.1 Experimental Configuration

In order to evaluate the effect of RS, we apply it to tree-based complete synchronous algorithms including TreeBB, PT-FB, HS-CAI and PT-ISBB (a variant of PT-ISABB [8] in DCOP settings), and name the corresponding RS-based version as TreeBB+RS, PT-FB+RS, HS-CAI+RS and PT-ISBB+RS. Specially, to ensure the completeness of OCS when applied to TreeBB for fairness, we also append the upper bound to the information unit and name it as TreeBB+OCS. Besides, we uniformly use the SYS scheme to manage memory when the cache is full. In our experiments, we will compare these RS-based algorithms with their originals on random DCOPs and weighted graph coloring problems. For random DCOPs, we set the graph density to 0.2, the domain size to 3 and vary the agent number from 16 to 28 as the sparse configuration, and the graph density to 0.5, the domain size to 3 and the number of agents varying from 14 to 24 as the dense configuration. For weighted graph coloring problems, similar to the sparse configuration in random DCOPs, we set the graph density to 0.2, the domain size to 3 and vary the agent number from 16 to 28. Besides, we choose $k = 4$ and $k = 8$ as the low and high memory budget for HS-CAI, PT-ISBB and RS, respectively. Furthermore, we compare the performance of different cache replacement schemes including FIFO (First In First Out), FILO (First In Last Out), LRU (Least Recently Used) and LFU (Least Frequently Used), UB, SYS and ORI (an original scheme which discards the next results when the cache is full.) when they are applied to TreeBB+RS, on the dense configuration for random DCOPs and low memory budget. We use the number of messages (Msgs) to measure the communication overheads, and the NCLOs [21] to measure the hardware-independent runtime where the logical operations in the inference and the search are accesses to utilities and constraint checks, respectively. For each experiment, we randomly generate 50 instances with the integer constraint costs in the range of 0 to 100, and report the average over all instances. The experiments are conducted on an i7-7820x workstation with 32GB of memory, and we set the timeout to 30 minutes for each algorithm.

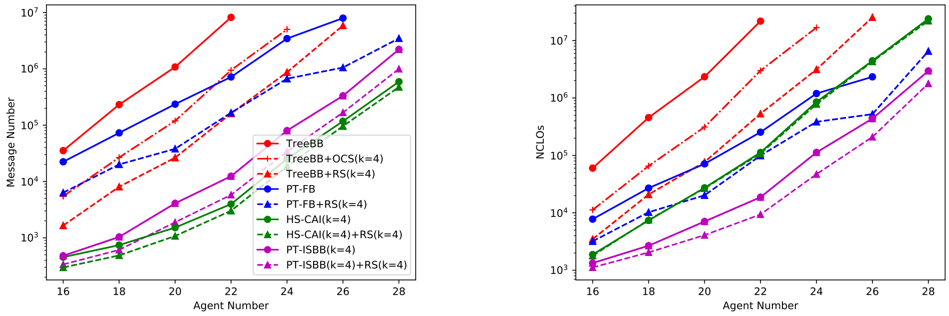
5.2 Experimental Results

Table 1 gives the detailed results of the improvement of the RS-based algorithms over their originals including both memory budget $k = 4$ and $k = 8$, where the numbers greater than zero are shown in bold. It can be seen that the RS-based algorithms outperform the originals on both metrics under both low and high memory budget, i.e., $k = 4$ and $k = 8$. Besides, the improvement under $k = 8$ is slightly higher than that under $k = 4$ in most cases since with larger memory, the RS-based algorithms can cache more *realcosts* for reuse. However, since the number of valid *realcost* is limited, such an improvement fades away. Next, Fig.4 to Fig.6 give the experimental results for the sparse random DCOPs, dense random DCOPs and weighted graph coloring problems with $k = 4$, and we do not present the results with $k = 8$ since they are of the same trend as that with $k = 4$.

Table 1 The improvement of the RS-based algorithms over their respective originals.

Configuration	k	TreeBB+OCS		TreeBB+RS		PTFB+RS		PT-ISBB+RS		HS-CAI+RS	
		Msgs(%)	NCLOs(%)	Msgs(%)	NCLOs(%)	Msgs(%)	NCLOs(%)	Msgs(%)	NCLOs(%)	Msgs(%)	NCLOs(%)
Sparse DCOPs	4	**84∼88**	**81∼86**	**95∼98**	**94∼97**	**71∼86**	**58∼77**	**29∼57**	**16∼52**	**18∼34**	**3∼8**
	8	**85∼89**	**81∼87**	**96∼99**	**95∼98**	**72∼87**	**59∼78**	**30∼49**	**1∼17**	**20∼40**	0∼5
Dense DCOPs	4	**45∼50**	**40∼44**	**72∼76**	**65∼71**	**19∼40**	**13∼42**	**9∼23**	**9∼34**	**2∼8**	0∼2
	8	**46∼50**	**41∼45**	**74∼80**	**69∼75**	**20∼42**	**14∼43**	**8∼15**	0∼12	**2∼15**	0∼6
weighted graph coloring	4	**39∼50**	**45∼49**	**41∼68**	**49∼60**	**39∼52**	**25∼47**	**34∼51**	**19∼40**	**21∼45**	**1∼9**
	8	**40∼54**	**46∼53**	**41∼71**	**49∼61**	**40∼53**	**26∼47**	**34∼48**	**1∼13**	**35∼50**	0∼5

Fig.4 presents the experimental results for the number of messages (a) and NCLOs (b) under different agent numbers on the sparse configuration for random DCOPs, and the corresponding improvement over the originals is displayed in the first row of Table 1. It can be seen that the RS-based algorithms exhibit a great advantage on both metrics over their originals. Especially for TreeBB and PT-FB, the RS improve them more. It is due to the fact that compared to PT-ISBB and HS-CAI, TreeBB and PT-FB perform less pruning due to their less tight lower bounds. Therefore, the cached results could be more likely to be the *realcost* and reused more frequently. Besides, it can be seen from Table 1 that TreeBB+RS is superior to TreeBB+OCS by about 10% on the the sparse problems, which indicates the fine-grained caching could indeed boost the cache-hit rate.

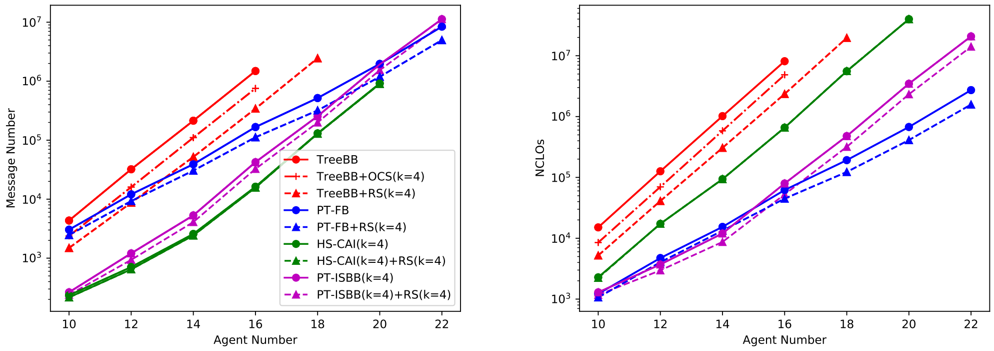


(a) Number of Messages.

(b) NCLOs.

Figure 4 Performance comparison under different agents on sparse DCOPs.

We can also see from Fig.4 that with the help of the RS, TreeBB and PT-FB solved larger problems, scaling up to the problems with 26 agents and with 28 agents, respectively.



(a) Number of Messages.

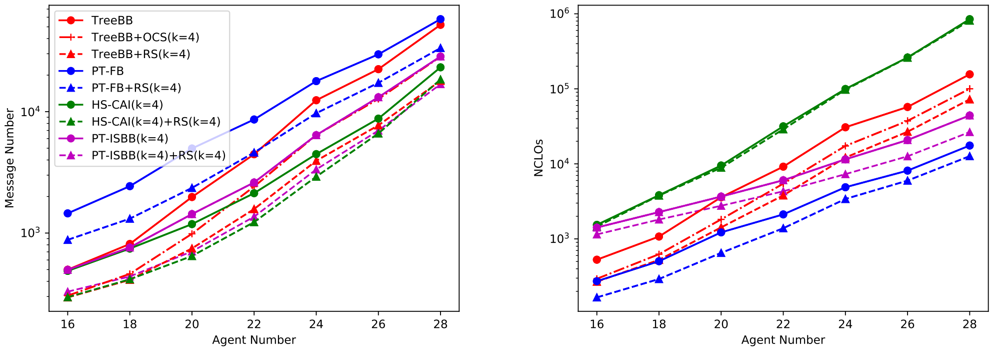
(b) NCLOs.

Figure 5 Performance comparison under different agents on dense DCOPs.

Fig.5 presents the experimental results under different agent numbers on the dense configuration for random DCOPs, and the third row of Table 1 shows the corresponding improvement over the originals. It can be seen that the RS-based algorithms also perform better than their originals in terms of both the number of messages and NCLOs. However, the improvement of RS decreases compared to that on the sparse configuration. It is because $|Sep(a_c)|$ for each agent a_i is larger on the dense configuration. Such large $Sep(a_c)$ will lead to more combinations for the $context_c$ and thus bring down the cache-hit rate. In addition, a_i may not be able to cache all the $context_c$ for large $Sep(a_c)$ with its limited memory, which would lead to the fact that the historical results for some $context_c$ would never be reused. Moreover, it can be seen from Table 1 that the outperformance of TreeBB+RS over TreeBB+OCS increases from about 10% on sparse problems to about 25% on dense problems. It is because under larger $|Sep(a_i)|$, it is more difficult for TreeBB+OCS to match a new $context_i$ to the cached items, while for TreeBB+RS it is much easier since the latter only needs to match the subset of $|Sep(a_i)|$ for its child.

Fig.6 presents the experimental results under different agent numbers on weighted graph coloring problems, and the corresponding improvement over the originals can be found in the fifth row of Table 1. It can be seen that our RS can greatly improve the performance of the originals on both metrics, which is similar to the results on random DCOPs. It is worth noting that, when $k = 4$, the performance of TreeBB+RS is better than all other competitors without RS on the number of messages, which verifies that a cache for reusing historical results has a more significant role than providing a tighter lower bound on weighted graph coloring problems.

Table 2 presents the experimental results for TreeBB+RS with different cache replacement schemes. We can find that our proposed cache replacement schemes perform better than other competitors in most cases. It is because ORI, FIFO, FILO, LRU, LFU are all model-free cache replacement schemes which ignore the structure of the problems, while our proposed methods consider the related information such as the cached upper bounds (UB) and the characteristics of synchronous algorithms (SYS). Besides, the SYS is better than UB in this case, as the SYS aims at the characteristics of sequential assignments in synchronous algorithms and adjusts itself by the initialized memory.



(a) Number of Messages.

(b) NCLOs.

Figure 6 Performance comparison under different agents on weighted graph coloring problems.

Table 2 Performance comparison under different cache replacement schemes on dense DCOPs.

Scheme	Metrics	Agent Number					Metrics	Agent Number				
		10	12	14	16	18		10	12	14	16	18
ORI	Msgs	1463	10841	84208	569718	3890905	NCLOs	5133	49443	449313	3626525	28718461
FIFO		1489	8717	52018	342777	2449699		5214	40978	306201	2353110	19653448
FILO		1466	10748	83715	566028	3825017		5135	48909	445818	3597353	28243741
LRU		1480	8705	52057	342724	2449808		5215	40971	306211	2352837	19655744
LFU		1466	10744	83521	565403	3840109		5137	48864	444652	3592756	28323712
UB		1481	8705	52003	342767	2449665		5215	40972	306185	2353149	19653238
SYS		**1433**	**8619**	**51412**	**340792**	**2437553**		**5002**	**40666**	**303945**	**2347033**	**19578876**

6 Conclusion

To overcome the shortcomings of OCS, we propose a new caching scheme named RS, which can be deployed to all tree-based complete synchronous search algorithms with minor modifications. It ensures the completeness of the algorithms by appending the upper bound to the information unit and further improves the cache utilization by adopting a fine-grained cache information unit. Besides, we also propose two cache replacement schemes UB and SYS to improve the performance of RS when the memory is limited. Finally, we give a theoretical proof for the completeness of RS, and our empirical evaluation shows the superiority of the RS-based algorithms over their originals and the advantage of our cache replacement schemes over the traditional ones. In the future, we will devote to further optimizing the cache information units and designing more appropriate cache replacement schemes for RS.

References

1 James Atlas, Matt Warner, and Keith Decker. A memory bounded hybrid approach to distributed constraint optimization. In *Proc. 10th International Workshop on Distributed Constraint Reasoning*, pages 37–51, 2008.

2 Ismel Brito and Pedro Meseguer. Improving DPOP with function filtering. In *AAMAS*, volume 1435, pages 141–148, 2010.

3 Anton Chechetka and Katia Sycara. No-commitment branch and bound search for distributed constraint optimization. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1427–1429, 2006.

4 Anton Chechetka and Katia P Sycara. An Any-space Algorithm for Distributed Constraint Optimization. In *AAAI Spring Symposium: Distributed Plan and Schedule Management*, pages 33–40, 2006.

5 Dingding Chen, Yanchen Deng, Ziyu Chen, Wenxing Zhang, and Zhongshi He. HS-CAI: A hybrid DCOP algorithm via combining search with context-based inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 7087–7094, 2020.

6 Ziyu Chen, Wenxin Zhang, Yanchen Deng, Dingding Chen, and Qiang Li. RMB-DPOP: Refining MB-DPOP by Reducing Redundant Inference. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, pages 249–257, 2020.

7 Rina Dechter, David Cohen, et al. *Constraint processing*. Morgan Kaufmann, 2003.

8 Yanchen Deng, Ziyu Chen, Dingding Chen, Xingqiong Jiang, and Qiang Li. PT-ISABB: A Hybrid Tree-based Complete Algorithm to Solve Asymmetric Distributed Constraint Optimization Problems. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1506–1514, 2019.

9 Alessandro Farinelli, Alex Rogers, and Nick R Jennings. Agent-based decentralised coordination for sensor networks using the max-sum algorithm. *Autonomous agents and multi-agent systems*, 28(3):337–380, 2014.

10 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 the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*, pages 639–646, 2008.

11 Ferdinando Fioretto, Enrico Pontelli, and William Yeoh. Distributed constraint optimization problems and applications: A survey. *Journal of Artificial Intelligence Research*, 61:623–698, 2018.

12 Ferdinando Fioretto, William Yeoh, Enrico Pontelli, Ye Ma, and Satishkumar J Ranade. A distributed constraint optimization (DCOP) approach to the economic dispatch with demand response. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 999–1007, 2017.

13 Eugene C Freuder and Michael J Quinn. Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems. In *IJCAI*, volume 85, pages 1076–1078. Citeseer, 1985.

14 Patricia Gutierrez and Pedro Meseguer. BnB-ADOPT+ with Several Soft Arc Consistency Levels. In *ECAI*, pages 67–72, 2010.

15 Katsutoshi Hirayama and Makoto Yokoo. Distributed partial constraint satisfaction problem. In *International conference on principles and practice of constraint programming*, pages 222–236. Springer, 1997.

16 Omer Litov and Amnon Meisels. Forward bounding on pseudo-trees for DCOPs and ADCOPs. *Artificial Intelligence*, 252:83–99, 2017.

17 Rajiv T Maheswaran, Jonathan P Pearce, and Milind Tambe. A family of graphical-game-based algorithms for distributed constraint optimization problems. In *Coordination of large-scale multiagent systems*, pages 127–146. Springer, 2006.

18 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 the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 310–317, 2004.

19 Toshihiro Matsui, Hiroshi Matsuo, and Akira Iwata. Efficient Methods for Asynchronous Distributed Constraint Optimization Algorithm. In *Artificial Intelligence and Applications*, pages 727–732, 2005.

20 Pragnesh Jay 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.

21 Arnon Netzer, Alon Grubshtein, and Amnon Meisels. Concurrent forward bounding for distributed constraint optimization problems. *Artificial Intelligence*, 193:186–216, 2012.

22 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.

23 Brammert Ottens, Christos Dimitrakakis, and Boi Faltings. DUCT: An upper confidence bound approach to distributed constraint optimization problems. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(5):1–27, 2017.

24 Adrian Petcu and Boi Faltings. DPOP: A scalable method for multiagent constraint optimization. In *IJCAI 05*, pages 266–271, 2005.

25 Adrian Petcu and Boi Faltings. ODPOP: An algorithm for open/distributed constraint optimization. In *AAAI*, volume 6, pages 703–708, 2006.

26 Adrian Petcu and Boi Faltings. MB-DPOP: A New Memory-Bounded Algorithm for Distributed Optimization. In *IJCAI*, pages 1452–1457, 2007.

27 Evan Sultanik, Pragnesh Jay Modi, and William C Regli. On Modeling Multiagent Task Scheduling as a Distributed Constraint Optimization Problem. In *IJCAI*, pages 1531–1536, 2007.

28 Meritxell Vinyals, Juan A Rodriguez-Aguilar, Jesús Cerquides, et al. Generalizing DPOP: Action-GDL, a new complete algorithm for DCOPs. In *AAMAS (2)*, pages 1239–1240, 2009.

29 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.

30 William Yeoh, Pradeep Varakantham, and Sven Koenig. Caching schemes for DCOP search algorithms. In *AAMAS (1)*, pages 609–616, 2009.

31 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.

CNF Encodings of Binary Constraint Trees

Ruiwei Wang
School of Computing, National University of Singapore, Singapore

Roland H. C. Yap
School of Computing, National University of Singapore, Singapore

Abstract

Ordered Multi-valued Decision Diagrams (MDDs) have been shown to be useful to represent finite domain functions/relations. For example, various constraints can be modelled with MDD constraints. Recently, a new representation called Binary Constraint Tree (BCT), which is a (special) tree structure binary Constraint Satisfaction Problem, has been proposed to encode MDDs and shown to outperform existing MDD constraint propagators in Constraint Programming solvers. BCT is a compact representation, and it can be exponentially smaller than MDD for representing some constraints. Here, we also show that BCT is compact for representing non-deterministic finite state automaton (NFA) constraints. In this paper, we investigate how to encode BCT into CNF form, making it suitable for SAT solvers. We present and investigate five BCT CNF encodings. We compare the propagation strength of the BCT CNF encodings and experimentally evaluate the encodings on a range of existing benchmarks. We also compare with seven existing CNF encodings of MDD constraints. Experimental results show that the CNF encodings of BCT constraints can outperform those of MDD constraints on various benchmarks.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence; Software and its engineering → Constraint and logic languages

Keywords and phrases BCT, CNF, MDD, NFA / MDD constraint, propagation strength

Digital Object Identifier 10.4230/LIPIcs.CP.2022.40

Funding This work was supported in part by the National Research Foundation Singapore under its AI Singapore Programme [AISG-RP-2018-005] and grant T1 251RES2024.

1 Introduction

Ordered Multi-valued Decision Diagram (MDD) [39] is a compact representation which can be used to encode finite domain functions/relations. Many constraints can be encoded into compact MDD constraints, such as the regular constraints [32], table constraints [10], among and sequence constraints [22]. MDD constraints are also useful to model problems requiring specific constraints which are not readily modelled with existing known constraints [11, 21]. In Constraint Programming (CP) solvers, MDD constraints can be directly handled with MDD Generalized Arc Consistency (GAC) propagators, e.g. the MDDc [10], MDD4R [31], CD [42] and CD^{bs} [43] propagators. Alternatively, MDD constraints can also be solved by SAT solvers by encoding MDD constraints into CNF form [1]. In this way, SAT solvers can directly handle the constraints which can be modelled with MDDs constraints[2, 3, 5].

Binary constraint is also a general representation for constraints. Any non-binary constraint can be transformed into binary constraints through binary encodings such as dual encoding [13], hidden variable encoding [37], double encoding [40] and bipartite encoding [47]. Recently, binary encodings with specialized Arc Consistency (AC) propagators [46, 47] has been shown to outperform the GAC propagators of non-binary table constraints [28, 45, 15, 50]. Similar to MDDs, the binary constraints can also be encoded into CNF with different CNF encodings, such as the log encoding [23, 44, 41], direct encoding [44] and support encoding [24, 19].

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 40; pp. 40:1–40:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Recently, a new representation called Binary Constraint Tree (BCT) [48], which is a set of binary constraints with tree structures (a special binary CSP), has been proposed to encode MDDs. BCT is a compact representation, and it can be exponentially smaller than MDD. In this paper, we also show that non-deterministic finite state automaton (NFA) constraints [33, 9] can be transformed into BCT constraints without exponential blow up but not MDD constraints. Furthermore, a GAC propagator of BCT constraints [48] has been shown to outperform the state-of-the-art MDD GAC propagators. The results in [48] show that BCT has great potential for encoding and reducing MDDs.

In this paper, we investigate how to encode BCT constraints into CNF instances and apply them in SAT solvers. We investigate five CNF encodings of BCT constraints, including the log encoding, direct encoding, support encoding and two new transformations: partial support encoding and minimal support encoding. We tailor three well-known CNF encodings of binary constraints, i.e. the log, direct and support encodings, to handle BCT constraints. In addition, we introduce the partial support encoding and minimal support encoding by eliminating clauses and Boolean variables from the support encoding of BCT constraints. Then we analyze the strength of unit propagation on these 5 CNF encodings of BCT constraints. The support encoding of BCT constraints, which implements propagation completeness [6], can have a greater propagation strength than the other CNF encodings. The partial support encoding and minimal support encoding are more compact than the support encoding but their propagation strength is weaker than the support encoding. The log encoding and direct encoding, which do not implement weak consistency, have the weakest propagation strength. We also compare the five CNF encodings of BCT constraints and seven existing MDD encodings [1] using the Kissat SAT solver [17] on a range of existing benchmarks. Our experimental results show that the CNF encodings of BCT constraints can outperform MDD CNF encodings.

The paper is organized as follows. Section 2 provides the preliminaries. Section 3 shows that BCT can be exponentially smaller than MDD on representing NFA constraints. Sections 4 and 5 respectively introduce CNF encodings of binary constraints and BCT constraints. Experimental results are given in Section 6, and Section 7 concludes.

2 Preliminaries

A CSP P is a pair (X, C) where X is a set of variables, $\mathcal{D}(x)$ is the domain of a variable x, and C is a set of constraints. A variable is a *Boolean variable* if $\mathcal{D}(x) = \{true, false\}$. A *literal* of a variable x is a variable value pair (x, a). A *tuple* over a set of variables $\{x_{i_1}, x_{i_2}, \ldots, x_{i_r}\}$ is denoted by a *set of literals* $\{(x_{i_1}, a_1), (x_{i_2}, a_2), \ldots, (x_{i_r}, a_r)\}$. Each constraint c has a constraint scope $scp(c) \subseteq X$ and a relation $rel(c)$ defined by a set of tuples over $scp(c)$. The arity of c is the number of variables in its scope, i.e. $|scp(c)|$. A constraint c is a *binary constraint* if $|scp(c)| = 2$. A constraint c over Boolean variables $\{x_{i_1}, \cdots, x_{i_r}\}$ is a *clause* if it is a disjunction of a set cl of literals $\{(x_{i_1}, a_1), \cdots, (x_{i_r}, a_r)\}$ such that $a_j \in \{true, false\}$ for $1 \leq j \leq r$, so $rel(c)$ consists of the tuples over $scp(c)$ including at least a literal in cl. A CSP is called a *binary CSP* if it only includes binary constraints. A binary CSP is *normalized* if all constraints have different scopes. A CSP (X, C) is called a *Conjunctive Normal Form (CNF)* if all variables in X are Boolean variables and all constraints in C are clauses.

Given any set of variables V and a tuple τ, we use $\tau[V] = \{(x, a) \in \tau | x \in V\}$ to denote a subset of τ, while $T[V] = \{\tau[V] | \tau \in T\}$ is the *projection* of tuples T on V. In addition, $P|_\tau$ denotes a subproblem $(\{x' | x \in X\}, C)$ of $P = (X, C)$ generated by assigning a tuple τ where $\mathcal{D}(x') = \mathcal{D}(x)$ if τ does not include any literal of x, otherwise $\mathcal{D}(x') = \{(x, a) \in \tau | a \in \mathcal{D}(x)\}$.

Note that $F|_\tau = F$ if $\tau = \emptyset$. A tuple τ is an assignment if $a \in \mathcal{D}(x)$ for all $(x, a) \in \tau$. An assignment τ over X is a solution of P if $\tau[\mathit{scp}(c)] \in \mathit{rel}(c)$ for all constraints $c \in C$. $\mathit{sol}(X, C)$ (or $\mathit{sol}(P)$) denotes all solutions of P. A CSP (X, C) is satisfiable if $\mathit{Sol}(X, C) \neq \emptyset$, otherwise it is unsatisfiable.

A *support of a value* $a \in \mathcal{D}(x)$ on a constraint c is a tuple $\tau \in \mathit{rel}(c)$ such that $(x, a) \in \tau$ and $b \in \mathcal{D}(y)$ for all $(y, b) \in \tau$. A variable $x \in \mathit{scp}(c)$ is *Generalized Arc Consistent* (GAC) on c if a has a support on c for all $a \in \mathcal{D}(x)$. c is GAC if all variables in $\mathit{scp}(c)$ are GAC on c. A CSP (X, C) is GAC if every constraint in C is GAC. For binary CSPs, GAC is also called *Arc Consistency* (AC). For any CNF F, GAC is also called *unit propagation*, where $UP(F)$ is used to denote a CNF generated from F by removing all variable values which are not GAC on F. If $UP(F)$ includes any empty variable domain, F is unsatisfiable.

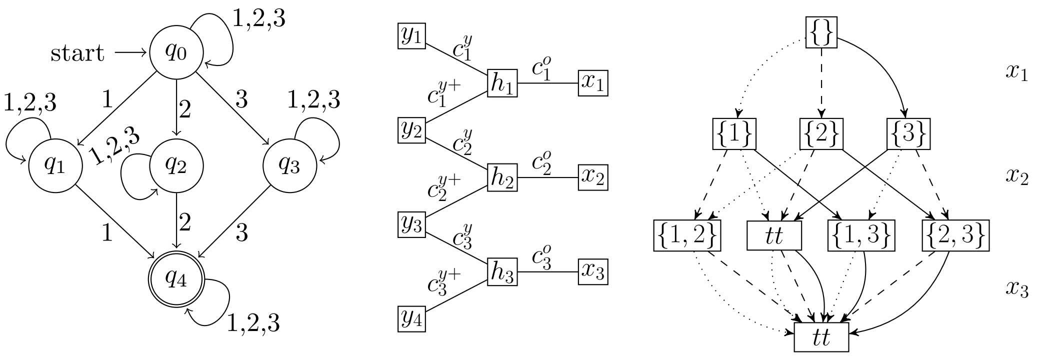
2.1 CNF encoding and unit propagation strength

A *CNF encoding* of a CSP $P = (X, C)$ is a CNF which is equisatisfiable with P, i.e. the CNF is satisfiable iff P is satisfiable. Typically, a CNF encoding consists of a variable encoding (VE) and a constraint encoding (CE) where VE encodes the variables in X as a set φ^X of Boolean variables and each constraint c in C corresponds to a constraint (Boolean function) φ^c over the Boolean variables, while CE encodes the constraint φ^c as a CNF F^c over Boolean variables Y such that $\mathit{scp}(\varphi^c) \subseteq Y$ and $\mathit{sol}(F^c)[\mathit{scp}(\varphi^c)] = \mathit{rel}(\varphi^c)$. There have been many VEs which can be used to transform finite domain variables into Boolean variables, such as direct encoding and log encoding [44] . In addition, there are also various CEs for encoding constraints, e.g. many CNF encodings of MDD constraints [1]. A way to analyze a CE is evaluating the strength of unit propagation on the encoded constraint F^c. We will use the following four levels to classify the strength of unit propagation on encoding F^c:

- F^c implements *weak consistency* if for any tuple τ over $\mathit{scp}(\varphi^c)$ such that $F^c|_\tau$ is unsatisfiable, some variable domains in $UP(F^c|_\tau)$ are empty.
- F^c implements *domain consistency* if for any tuple τ over a subset of $\mathit{scp}(\varphi^c)$ and a literal (v, a) in $UP(F^c|_\tau)$ such that $v \in \mathit{scp}(\varphi^c)$ and all variable domains in $UP(F^c|_\tau)$ are not empty, there is at least a solution of $F^c|_\tau$ including (v, a).
- F^c implements *unit refutation completeness* [14, 1, 25, 26] if for any tuple τ over a subset of Y such that $F^c|_\tau$ is unsatisfiable, some variable domains in $UP(F^c|_\tau)$ are empty.
- F^c implements *propagation completeness* [6, 1, 25, 26] if for any tuple τ over a subset of Y and a literal l in $UP(F^c|_\tau)$ such that all variable domains in $UP(F^c|_\tau)$ are not empty, there is at least a solution of $F^c|_\tau$ including l.

In this classification, propagation completeness is the strongest level, unit refutation completeness is incomparable with domain consistency, and weak consistency is the weakest level. Note that the definition of weak/domain consistency is defined for evaluating the strength of unit propagation and will be used in the rest of this paper.

▶ **Example 1.** Assume the CNF F^c is $(\{x, y, z\}, \{x \vee y, \neg y \vee z, x \vee \neg z\})$ and the scope $\mathit{scp}(\varphi^c)$ is $\{x, y\}$. (x, false) is the only literal in F^c which cannot be extended to a solution of F^c. (x, false) is in $UP(F^c)$, so F^c does not implement propagation completeness. Then x is in $\mathit{scp}(\varphi^c)$, thus, F^c also does not implement domain consistency. For any tuple τ over a subset of $\{x, y, z\}$, if $F^c|_\tau$ is unsatisfiable, $UP(F^c|_\tau)$ has empty variable domains. So F^c implements unit refutation completeness and weak consistency.



(a) NFA: nodes and edges respectively denote states and transitions.

(b) BCT: nodes and edges are variables and constraints.

(c) MDD: dotted (dashed and solid) lines denote the value 1 (2 and 3).

Figure 1 Different representations of a constraint over 3 variables $\{x_1, x_2, x_3\}$.

3 BCT versus MDD on representing NFA constraints

Binary Constraint Tree (BCT) is a compact representation which can be exponentially smaller than Ordered Multi-valued Decision Diagram (MDD). In this section, we show that any NFA constraint can be encoded as a BCT constraint without exponential blow up, where NFA may be exponentially smaller than the corresponding MDD.

A non-deterministic finite state automaton (NFA) is a quintuple $(Q, \sum, \Delta, q_0, Q_t)$ consisting of a finite set Q of states, a finite set $\sum$ of input symbols, a transition function $\Delta : Q \times \sum \to 2^Q$, an initial state $q_0 \in Q$, and a set $Q_t \subseteq Q$ of accepting states, where there is a transition in the NFA from a state $q_i \in Q$ to a state $q_j \in Q$ via a symbol $a \in \sum$ if $q_i \in \Delta(q_j, a)$. A string $a_1 ... a_r$ is accepted by the NFA if there is a sequence of states, $s_0, s_1, ..., s_r$, such that: $s_0 = q_0$, $s_i \in Q$ and $s_{i+1} \in \Delta(s_i, a_{i+1})$ for $0 \leq i < r$, and $s_r \in Q_t$. A NFA constraint c is a pair (G, O) such that O is an ordering over $scp(c)$, G is a NFA and $rel(c)$ is the set of tuples $\{(O_1, a_1), ..., (O_r, a_r)\}$ over $scp(c)$ such that the string $a_1 ... a_r$ is accepted by the NFA [33, 9].

▶ **Example 2.** Consider the constraint $\bigvee_{i=1}^{r} \bigvee_{j=i+1}^{r} (x_i = x_j)$ over r variables $\{x_1, ..., x_r\}$ with variable domain $\{1, .., r\}$, which expresses the negation of an *alldifferent* constraint [35]. The size of the negation of the MDDs (Ordered Multi-valued Decision Diagrams) representing alldifferent constraints is exponential in r [4].

We can use a NFA $(\{q_0, ..., q_{r+1}\}, \{1, ..., r\}, \Delta, q_0, \{q_{r+1}\})$ to model the constraint such that $\Delta(q_0, i) = \{q_0, q_i\}$, $\Delta(q_i, i) = \{q_i, q_{r+1}\}$, $\Delta(q_{r+1}, i) = \{q_{r+1}\}$ and $\Delta(q_j, i) = \{q_j\}$ for $1 \leq i \leq r$ and $j \neq 0, i, r+1$. Figure 1a gives a NFA for $r = 3$, and Figure 1c is a MDD modelling the NFA, where every subset of the domain corresponds to a node in the MDD.

▶ **Definition 3** (BCT and BCT constraint [48]). *A* Binary Constraint Tree *(BCT) is a normalized binary CSP whose constraint graph is a tree. A* BCT constraint *c is a pair (V, P) such that $P = (X, C)$ is a BCT, $scp(c) = V$, $V \subseteq X$ and $rel(c) = sol(X, C)[V]$, where the variables in $scp(c)$ and $X \setminus scp(c)$ are respectively called the* original *and* hidden *variables.*

We recap BCT, see [48] for more details. BCT is viewed as a single non-binary constraint, the BCT constraint, modelled as a binary CSP with hidden variables. Given the tree structure of the BCT, AC on the BCT can achieve GAC on the BCT constraint. It has been shown in [48] that any MDD constraint can be encoded into a BCT constraint with the same size as the MDD. A BCT can be further optimized with the reduction rules given in [48]. After the reduction, BCT constraints can be much smaller than the corresponding MDD constraints.

3.1 Direct tree binary encoding

For any NFA constraint c^* over r variables, we can encode the states and transitions of the NFA into a sequence of hidden variables, and then representing the NFA constraint c^* as a BCT over the hidden variables such that every tuple in the constraint relation denotes a sequence of r transitions from the initial state to an accepting state in the NFA. The details of the encoding are given in Definition 4.

▶ **Definition 4.** *A* direct tree binary encoding *(DTBE) of a NFA constraint $c^* = (G, O)$ is a BCT $dtbe(c^*)$= $(Y \cup H \cup scp(c^*), \{c_1^o, c_1^y, c_1^{y+}, ..., c_r^o, c_r^y, c_r^{y+}\})$ where*

- *$r = |scp(c^*)|$ and $G = (Q, \sum, \Delta, q_0, Q_t)$ and $Y = \{y_1, ..., y_{r+1}\}$ and $H = \{h_1, ..., h_r\}$;*
- *$scp(c_i^o) = \{O_i, h_i\}$, $scp(c_i^{y+}) = \{y_{i+1}, h_i\}$ and $scp(c_i^y) = \{y_i, h_i\}$;*
- *$\mathcal{D}(y_1) = \{q_0\}$, $\mathcal{D}(y_{r+1}) = Q_t$ and $\mathcal{D}(y_i) = Q$ for $2 \le i \le r$;*
- *$\mathcal{D}(h_i)$ is the set of transitions in G for $1 \le i \le r$;*
- *$rel(c_i^y) = \{\{(h_i, tr), (y_i, b)\} | tr \in \mathcal{D}(h_i)$, tr is a transition from $b\}$;*
- *$rel(c_i^o) = \{\{(h_i, tr), (O_i, s)\} | tr \in \mathcal{D}(h_i)$, s is the symbol of the transition $tr\}$;*
- *$rel(c_i^{y+}) = \{\{(h_i, tr), (y_{i+1}, b)\} | tr \in \mathcal{D}(h_i)$, tr is a transition to $b\}$.*

We remark that we use the term DTBE for NFA constraints in the same way as the DTBE encoding of MDD in [48], we refer to [48] for more details. Figure 1b shows the constraint graph of a DTBE for a NFA constraint (G, O), where G is given in Figure 1a and O is the variable order $x_1 \le x_2 \le x_3$. The states and transitions in the NFA are respectively encoded as the hidden variables $y_1, \cdots, y_4$ and $h_1, \cdots, h_3$, where the states and transitions are encoded as hidden variable values, i.e. $\mathcal{D}(y_1) = \{q_0\}$, $\mathcal{D}(y_2) = \mathcal{D}(y_3) = \{q_0, q_1, q_2, q_3, q_4\}$, $\mathcal{D}(y_4) = \{q_4\}$ and the domain of h_1, h_2, h_3 is the set of all transitions $\{(q_i, a, q_j) | q_j \in \Delta(q_i, a), 0 \le i \le 4, 1 \le a \le 3\}$ in the NFA. Then binary constraints are used to combine transitions with symbols and states. The binary constraint relations can be constructed based on Definition 4. For example, $rel(c_1^y) = \{\{(y_1, q_0), (h_1, (q_0, a, q_i))\} | i \in \{0, a\}, a \in \{1, 2, 3\}\}$, $rel(c_1^{y+}) = \{\{(y_2, q_i), (h_1, (q_0, a, q_i))\} | i \in \{0, a\}, a \in \{1, 2, 3\}\}$, $rel(c_1^o)$ $= \{\{(x_1, a), (h_1, (q_0, a, q_i))\} | i \in \{0, a\}, a \in \{1, 2, 3\}\}$. In addition, the reduction rules given in [48] can also be directly used to reduce the DTBE encodings of NFA constraints.

▶ **Theorem 5.** *BCT can be exponentially smaller than MDD on representing NFA constraints.*

The proof of Theorem 5 (also Lemma 10 and Propositions 11, 13, 14, 16) can be found in the Appendix. Theorem 5 shows that there exists a family of NFA constraints (Example 2) for which the BCT representation (DTBE) is exponentially smaller than the MDD representation. For example, give the NFA constraint in Example 2 with $r = 15$, the BCT representation (after reduction) has 2K values and 6K tuples which is much smaller than the MDD representation having 33K nodes and 491K edges.

4 CNF encodings for binary constraints

In this section, we introduce three well-known CNF encodings, i.e. log encoding [23, 44, 41], direct encoding [44] and support encoding [24, 19], which are used to transform any binary CSP (X, C) into CNF. These CNF encodings represent each variable x in X as a set of Boolean variables such that every value in $\mathcal{D}(x)$ corresponds to exactly one tuple over the variables. In addition, for each binary constraint $c \in C$, the encodings use clauses to represent the tuples τ over $scp(c)$ such that $\tau \notin rel(c)$ (or $\tau \in rel(c)$). These CNF encodings can be directly used to encode BCT constraints, since any BCT is also a binary CSP.

4.1 Log encoding

Every variable $x \in X$ is represented as $k = \lceil \log_2(d) \rceil$ Boolean variables $V^x = \{v_1^x, ..., v_k^x\}$, where $d = |\mathcal{D}(x)|$. Let T be the set of the first d assignments over B^x in the lexicographic order. Every value a in $\mathcal{D}(x)$ corresponds to a tuple τ_a in T. In addition, if $|\mathcal{D}(x)|$ is not a power of two, the assignments over V^x which are not in T can be excluded by adding the following clause [41] for each literal $(v_i^x, false)$ in the d^{th} (last) tuple τ in T

$$f(v_1^x) \vee \cdots \vee f(v_{i-1}^x) \vee \neg v_i^x \qquad where \quad f(v_j^x) = \begin{cases} v_j^x & \text{if } (v_j^x, false) \in \tau \\ \neg v_j^x & \text{if } (v_j^x, true) \in \tau \end{cases}$$

For each constraint $c \in C$ and an assignment $\{(x, a), (y, b)\}$ over $scp(c)$, if the tuple is not in $rel(c)$, then the following clause is added to exclude the tuple

$$(\bigvee_{(v_j^x, true) \in \tau_a \cup \tau_b} \neg v_j^x) \vee (\bigvee_{(v_j^x, false) \in \tau_a \cup \tau_b} v_j^x)$$

▶ **Example 6.** The log encoding of the binary CSP $P = (\{x_1, x_2, x_3, x_4\}, \{x_1 + x_3 \leq 5, x_3 = 3 \vee x_4 = 3, x_2 + x_4 \leq 5\})$, where variable domains are $\{1, 2, 3\}$, consists of 8 Boolean variables $\{v_1^{x_1}, v_2^{x_1}, v_1^{x_2}, v_2^{x_2}, v_1^{x_3}, v_2^{x_3}, v_1^{x_4}, v_2^{x_4}\}$ and 10 clauses:

$$\begin{array}{cccc} \neg v_1^{x_1} \vee \neg v_2^{x_1} & \neg v_1^{x_2} \vee \neg v_2^{x_2} & \neg v_1^{x_3} \vee \neg v_2^{x_3} & \neg v_1^{x_4} \vee \neg v_2^{x_4} \end{array}$$

$$\begin{array}{ccc} \neg v_1^{x_1} \vee v_2^{x_1} \vee \neg v_1^{x_3} \vee v_2^{x_3} & \neg v_1^{x_2} \vee v_2^{x_2} \vee \neg v_1^{x_4} \vee v_2^{x_4} & v_1^{x_3} \vee v_2^{x_3} \vee v_1^{x_4} \vee v_2^{x_4} \\ v_1^{x_3} \vee \neg v_2^{x_3} \vee v_1^{x_4} \vee v_2^{x_4} & v_1^{x_3} \vee v_2^{x_3} \vee v_1^{x_4} \vee \neg v_2^{x_4} & v_1^{x_3} \vee \neg v_2^{x_3} \vee v_1^{x_4} \vee \neg v_2^{x_4} \end{array}$$

4.2 Direct encoding

Every variable $x \in X$ is represented as d Boolean variable $B^x = \{v_{a_i}^x | a_i \in \mathcal{D}(x)\}$, where $\mathcal{D}(x) = \{a_1, \cdots, a_d\}$. In addition, an exactly-one constraint over B^x is introduced to guarantee that if a variable in B^x is assigned with *true*, then the other variables in B^x are assigned with *false*. The exactly-one constraint is encoded with *ladder encoding* [20] which includes $d - 1$ additional Boolean variables $A^x = \{w_1^x, \cdots, w_{d-1}^x\}$ and a set of $EO(x)$ clauses:

$$\neg v_{a_1}^x \vee \neg w_1^x \qquad v_{a_1}^x \vee w_1^x \qquad v_{a_d}^x \vee \neg w_{d-1}^x \qquad \neg v_{a_d}^x \vee w_{d-1}^x$$
$$\{w_{i-1}^x \vee \neg w_i^x, v_{a_i}^x \vee w_i^x \vee \neg w_{i-1}^x, \neg v_{a_i}^x \vee \neg w_i^x, \neg v_{a_i}^x \vee w_{i-1}^x | 2 \leq i \leq d-1\}$$

The latter encoding implements propagation completeness [6, 25]. The clauses in $EO(x)$ can guarantee that every value a_i in $\mathcal{D}(x)$ corresponds to exactly one solution $t(x, a_i)$ of the CNF $(B^x \cup A^x, EO(x))$, where $t(x, a_i) = \{(v_b^x, false) | b \in \mathcal{D}(x), b \neq a_i\} \cup \{(v_{a_i}^x, true)\} \cup \{(w_j^x, true) | 1 \leq j < i\} \cup \{(w_j^x, false) | i \leq j < d\}$. Then for each $c \in C$ and an assignment $\tau = \{(x, a_i), (y, b_j)\}$ over $scp(c)$ such that $\tau \notin rel(c)$, the clause $\neg v_{a_i}^x \vee \neg v_{b_j}^y$ is added.

▶ **Example 7.** The direct encoding of the binary CSP given in Example 6 includes 20 Boolean variables and the following clauses:

$$\{\neg v_1^{x_j} \vee \neg w_1^{x_j}, v_1^{x_j} \vee w_1^{x_j}, v_3^{x_j} \vee \neg w_2^{x_j}, \neg v_3^{x_j} \vee w_2^{x_j} | 1 \leq j \leq 4\}$$
$$\{w_1^{x_j} \vee \neg w_2^{x_j}, v_2^{x_j} \vee w_2^{x_j} \vee \neg w_1^{x_j}, \neg v_2^{x_j} \vee \neg w_2^{x_j}, \neg v_2^{x_j} \vee w_1^{x_j} | 1 \leq j \leq 4\}$$
$$\neg v_3^{x_1} \vee \neg v_3^{x_3} \quad \neg v_3^{x_2} \vee \neg v_3^{x_4} \quad \neg v_1^{x_3} \vee \neg v_2^{x_4} \quad \neg v_1^{x_3} \vee \neg v_1^{x_4} \quad \neg v_2^{x_3} \vee \neg v_1^{x_4} \quad \neg v_2^{x_4} \vee \neg v_2^{x_4}$$

4.3 Support encoding

Every variable $x \in X$ is represented as d Boolean variable $B^x = \{v_{a_i}^x | a_i \in \mathcal{D}(x)\}$, where $\mathcal{D}(x) = \{a_1, \cdots, a_d\}$, and an exactly-one constraint over B^x is used to make sure that each value in $\mathcal{D}(x)$ corresponds to exactly one tuple over B^x. The exactly-one constraint is encoded with $(A^x \cup B^x, EO(x))$, i.e. ladder encoding. In addition, for each value $a \in \mathcal{D}(x)$ and a constraint $c \in C$ such that $scp(c) = \{x, y\}$, a clause $cl(x, a, c)$ is added where

$$cl(x,a,c) = \neg v_a^x \vee (\bigvee_{\{(x,a),(y,b)\} \in rel(c) \wedge b \in \mathcal{D}(y)} v_b^y)$$

By using the clauses $cl(x, c) = \{cl(x, a, c) | c \in C, x \in scp(c), a \in \mathcal{D}(x)\}$, unit propagation on the support encoding of a binary CSP can achieve AC on the binary CSP [19].

5 CNF encoding for BCT constraints

For any BCT constraint (V, P), the CNF encodings of binary constraints can be directly used to encode the BCT constraint, because the BCT P is a binary CSP. In addition, binary constraints are special cases of BCT constraints, i.e. every binary constraint is a BCT constraint without any hidden variables. When comparing the unit propagation, there is a subtlety, the strength of unit propagation on the CNF encodings of BCT constraints can be different from that for binary constraints. In this section, we will discuss the strength of unit propagation when using the log, direct and support encodings to encode BCT constraints as CNF. Afterwards, we will propose two further CNF encodings of BCT constraints by eliminating Boolean variables and clauses from the support encoding of the constraints.

5.1 Encodings from binary constraints

The log encoding of binary constraints implements weak consistency but we highlight that it does not do so for BCT constraints, since BCT constraints can include hidden variables. Assume P is the binary CSP given in Example 6 and F is the log encoding of the BCT constraint $(\{x_1, x_2\}, P)$. Then F does not implement weak consistency. For example, the tuple τ=$\{(v_1^{x_1}, true)$, $(v_2^{x_1}, false)$, $(v_1^{x_2}, true)$, $(v_2^{x_2}, false)\}$ is not included by any solution of F, i.e. $F|_\tau$ is unsatisfiable but $UP(F|_\tau)$ does not include any empty variable domain. Therefore, the log encoding of BCT constraints does not implement weak consistency.

▶ **Proposition 8.** *Log encoding of BCT constraints does not implement weak consistency.*

Similarly, the direct encoding of BCT constraints also does not implement weak consistency. For example, the tuple τ=$\{(v_3^{x_1}, true)$, $(v_3^{x_2}, true)\}$ is not included by any solution of the direct encoding F (given in Example 7) of the BCT constraint $(\{x_1, x_2\}, P)$, i.e. $F|_\tau$ is unsatisfiable, but $UP(F|_\tau)$ does not include any empty variable domain.

▶ **Proposition 9.** *Direct encoding of BCT constraints does not implement weak consistency.*

Unit propagation on the support encoding of a binary CSP can achieve AC on the binary CSP. Further, the constraint graph of a BCT is a tree. Hence, unit propagation on the support encoding of a BCT constraint can achieve GAC on the BCT constraint. For any BCT (X, C), we can set a variable $x \in X$ as the root and construct a tree order O over X such that $O_1 = x$, where O is a *tree order* if for any $j > 1$ and $O_j \in X$, there is exactly one constraint $c \in C$ such that $scp(c) = \{O_i, O_j\}$ and $i < j$. In addition, we use T^O to denote a set of clauses $\bigcup\{cl(O_i, c) | c \in C, scp(c) = \{O_i, O_j\}, i < j\}$ with respect to a tree order O.

▶ **Lemma 10.** *Given a BCT $P = (X, C)$ and a tree order O over X, if a literal $(v_a^{O_1}, true)$ is included in $UP(F)$ and all variable domains in $UP(F)$ are not empty, there is $\tau \in sol(P)$ such that $(O_1, a) \in \tau$ and $(v_b^x, true)$ is included in F for all $(x, b) \in \tau$, where $F = (A, C^A)|_{\tau'}$ and $B^x \subseteq A$ for all $x \in X$ and $T^O \subseteq C^A$ and τ' is a tuple over a subset of A.*

We now show the support encoding of BCT constraints implements propagation completeness.

▶ **Proposition 11.** *The support encoding $F = (A \cup B, T \cup E)$ of BCT constraints (V, P) implements propagation completeness, where P is a BCT (X, C) and $A = \bigcup_{x \in X} A^x$ and $B = \bigcup_{x \in X} B^x$ and $T = \{cl(x, c) | c \in C, x \in scp(c)\}$ and $E = \bigcup_{x \in X} EO(x)$.*

5.2 Partial support encoding

We now introduce a new CNF encoding of BCT constraints, called partial support encoding, by eliminating clauses from the support encoding of the BCT constraints. Give any BCT $P = (X, C)$ and variables $V \subseteq X$, the *partial support encoding* of the BCT constraint (V, P) is a CNF $F = (A^V \cup B^V \cup B^H, T \cup E^V)$, where $A^V = \bigcup_{x \in V} A^x$ and $B^V = \bigcup_{x \in V} B^x$ and $B^H = \bigcup_{h \in X \setminus V} B^h$ and $T = \{cl(x, c, a) | c \in C, x \in scp(c), a \in \mathcal{D}(x)\}$ and $E^V = \bigcup_{x \in V} EO(x)$. Partial support encoding has the same Boolean variables and clauses as the support encoding of (V, P), except that the clauses in $EO(h)$ and the Boolean variables in A^h are removed from the support encoding of (V, P) for any hidden variables h in $X \setminus V$.

For each solution τ of P, we can construct a solution of F, e.g. $(\bigcup_{(x,a) \in \tau} t(x, a)) \cup \{(v_a^x, true) | (x, a) \in \tau\} \cup \{(v_a^x, false) | (x, a) \notin \tau, a \in \mathcal{D}(x)\}$. Conversely, every solution τ of F corresponds to at least one solution of P (see Lemma 10), since variable domains in $UP(F|_\tau)$ are not empty. However, the strength of unit propagation on the partial support encoding is weaker than that on the support encoding for BCT constraints. Partial supporting encoding implements domain consistency and unit refutation completeness but not propagation completeness.

For the partial support encoding F of a BCT $(\{x_1, x_2\}, P)$ where P is the binary CSP (BCT) given in Example 6, the literal $(v_3^{x_3}, false)$ cannot be extended to any solution of the CNF $F|_\tau$ but $(v_3^{x_3}, false)$ is in $UP(F|_\tau)$, where $\tau = \{(v_3^{x_4}, false), (v_1^{x_3}, false), (v_2^{x_3}, false), (v_3^{x_1}, false)\}$ is a tuple over the Boolean variables $\tau = \{v_3^{x_4}, v_1^{x_3}, v_2^{x_3}, v_3^{x_1}\}$. Therefore, partial support encoding of BCT constraint does not implement propagation completeness.

▶ **Proposition 12.** *Partial support encoding for BCT constraints does not implement propagation completeness.*

Partial support encoding implements unit refutation completeness (based on Lemma 10), thus, it also implements weak consistency.

▶ **Proposition 13.** *The partial support encoding $F = (A^V \cup B^V \cup B^H, T \cup E^V)$ of a BCT constraint (V, P) implements unit refutation completeness where $P = (X, C)$.*

In addition, from Lemma 10, we can also get that partial support encoding implements domain consistency.

▶ **Proposition 14.** *The partial support encoding $F = (A^V \cup B^V \cup B^H, T \cup E^V)$ of a BCT constraint (V, P) implements domain consistency where $P = (X, C)$.*

5.3 Minimal support encoding

We now give a more compact CNF encoding of BCT constraints called minimal support encoding. Give any BCT $P = (X, C)$ and variables $V \subseteq X$, the *minimal support encoding* of the BCT constraint (V, P) with respect to a tree order O over X is a CNF $F = (A^V \cup B^V \cup B^H, T^O \cup E^V)$, where $P = (X, C)$ and $A^V = \bigcup_{x \in V} A^x$ and $B^V = \bigcup_{x \in V} B^x$ and $B^H = \bigcup_{h \in X \setminus V} B^h$ and $E^V = \bigcup_{x \in V} EO(x)$ and $O_1 \in V$. Minimal support encoding has the same Boolean variables as the partial support encoding but the minimal support encoding does not include the clauses $cl(O_j, c)$ for any binary constraint $c \in C$ between 2 variables $O_i, O_j \in X$ such that $i < j$.

The strength of unit propagation on minimal support encoding is weaker than that on partial support encoding. For the minimal support encoding F of a BCT $(\{x_1, x_2\}, P)$ with respect to a tree order $x_1 < x_3 < x_4 < x_2$ and a tuple $\tau = \{(v_3^{x_1}, true)\}$ where P is the binary CSP given in Example 6, the literal $(v_3^{x_4}, true)$ cannot be extended to a solution of the CNF $F|_\tau$ but $(v_3^{x_4}, true)$ is included in $UP(F|_\tau)$. So the minimal support encoding does not implement domain consistency and propagation completeness.

▶ **Proposition 15.** *Minimal support encoding for BCT constraints does not implement domain consistency and propagation completeness.*

In addition, the minimal support encoding is stronger than the log and direct encodings for BCT constraints. From Lemma 10, we can get that the minimal support encoding implements unit refutation completeness and weak consistency.

▶ **Proposition 16.** *The minimal support encoding F of a BCT constraint (V, P) with respect to a tree order O implements unit refutation completeness where $x = O_1$ and $P = (X, C)$.*

Table 1 summarizes the strength of unit propagation on all five CNF encoding of BCT constraints. The support encoding of BCT constraints, which implements propagation completeness, is the strongest encoding. Then the partial support encoding implementing domain consistency is stronger than the log, direct and minimal support encodings. In addition, the log and direct encodings of BCT constraints are weaker than the minimal support encoding, since the log and direct encodings of BCT constraints do not implement weak consistency.

Table 1 Strength of Unit Propagation on various encodings of BCT constraints. The label ✓(✗) denotes that the CNF encodings (does not) implement a unit propagation strength level.

	Log	Direct	Minimal Support	Partial Support	Support
Weak consistency	✗	✗	✓	✓	✓
Domain consistency	✗	✗	✗	✓	✓
Unit refutation completeness	✗	✗	✓	✓	✓
Propagation completeness	✗	✗	✗	✗	✓

6 Experiments

We evaluate our five CNF encodings of BCT constraints, i.e. the log encoding, direct encoding, support encoding, PS (partial support) encoding and MS (minimal support) encoding, with seven existing CNF encodings [1] of MDD constraints, i.e. the Min (minimal), GMin (GenMiniSAT), Tes (Tseitin), BaP (Basic path), LevP (level path), NNFP (NNF path) and ComP (complete path) encodings. We employ the Kissat SAT solver [17] in default configuration to solve the resulting CNF. We also test a BCT GAC propagator [48] in the Abscon solver [29],[1] where the Abscon solver uses the binary branching MAC and geometric restart strategy[2], lexical value heuristic and five choices of variable heuristics (Lexical, DDeg [38], WDeg [7], Activity [30] and Impact [34]). We highlight the Abscon results are to compare CNF encodings with a SAT solver with a GAC propagator in a CP solver.

Experiments were run on a 3.20GHz Intel i7-8700 machine. Solving time is limited to 10 minutes per instance and memory to 12G. We tested the encodings with existing benchmarks also used by other papers [18, 9], and instances from the 2019 XCSP competition[3] and Minizinc challenge 2021.[4]

Tables 2, 3, 4 and 5 show the average solving times (in seconds) of the CNF encodings and the Abscon solver, and if there are some instances which cannot be solved in 10 minutes, then the tables give the number of unsolvable instances, i.e. the number of time-out or memory-out instances. The "Inst." and "# I" columns respectively give the names of different instance series and the numbers of CSP instances used. The "# Sol" row shows the total number of instances solved in 10 minutes. The best results of all methods are in bold, and the underlined results are the best results of the CNF encodings. The "Itime" row is the average initialization time (in seconds) of different methods and includes the encoding times. In addition, the "Itime" row also shows the number of memory-out instances if the CNF encoding runs out of memory during initialization. For different benchmarks, the Abscon solver results use the best variable heuristic from the 5 heuristics.

Figures 2a, 3a, 4a and 5a compare the performance profiles [16] of the methods VB-BCT, VB-MDD and VB-Abscon, where VB-BCT and VB-MDD respectively denote the virtual best CNF encoding of BCT constraints and MDD constraints respectively. VB-Abscon is the Abscon solver using the virtual best variable heuristic. The y-axis is the percentage of instances solved and the x-axis is the ratio of the solving time of a method to the virtual best method (time ratio). In the figures, we remove (i) the trivial instances which can be solved by all methods (VB-BCT, VB-MDD, and VB-Abscon) in 2 seconds and (ii) the instances which cannot be solved by any method in 10 minutes. Figures 2b, 3b, 4b and 5b use scatter plots to compare the solving times of various CNF encodings. In the figures, each dot denotes a CSP instance of a series, and the dot shapes correspond to instance series. In order to use logarithmic scales, the time on the x/y-axis is set as (1 + solving time). Figures 2c, 3c, 4c and 5c compare the number of clauses of different CNF encodings given by the x and y-axis.

[1] We have used the CP BCT propagator from [48] implemented in Abscon.

[2] The initial $cutoff = 10$ and $\rho = 1.1$. For each restart, $cutoff$ is the allowed number of failed assignments and $cutoff$ increases by $(cutoff \times \rho)$ after restart.

[3] `http://www.cril.univ-artois.fr/XCSP19/`

[4] `https://www.minizinc.org/challenge.html`

6.1 Benchmark Series 1: NFA

We use the 6 NFA series which are also used in [9]. These NFA benchmarks are modelled with NFA constraints. The NFA constraints can be transformed into BCT constraints and also MDD constraints. The direct tree binary encoding introduced in Section 3 are used to encode NFA constraints as BCT constraints, and then the BCT constraints are further reduced with the reduction rules proposed in [48]. In addition, the automaton library dk.brics.automaton[5] is used to minimize and transform any NFA into a DFA, where the DFA is directly expanded into the corresponding (quasi-reduced [1]) MDD.

Table 2 NFA benchmarks.

		BCT					MDD							Abscon
Inst.	#I	Log	Direct	Support	PS	MS	Min	GMin	Tes	BaP	LevP	NNFP	ComP	(DDeg)
NFA-50	14	1 out	166.42	11.62	7.76	4.86	7.91	2 out	3 out	102.18	126.79	109.96	107.57	**1.21**
NFA-36	18	16 out	10 out	61.69	30.40	19.53	1 out	15 out	15 out	11 out	11 out	11 out	13 out	**4.98**
NFA-34	13	89.63	14.57	2.20	1.38	1.25	1.78	16.56	24.90	19.59	25.25	19.16	21.17	**0.54**
NFA-54	15	15 out	14 out	179.23	91.95	69.11	2 out	15 out	15 out	15 out	15 out	15 out	15 out	**11.62**
NFA-57	15	14 out	10 out	1 out	55.42	27.07	1 out	13 out	15 out	10 out	11 out	11 out	12 out	**6.40**
NFA-60	15	14 out	6 out	30.13	16.80	10.13	25.79	12 out	15 out	2 out	3 out	3 out	2 out	**2.50**
#Sol	90	30	50	89	**90**	**90**	86	33	27	52	50	50	48	**90**
Itime	90	0.51	0.38	0.25	0.23	0.23	290.65	292.62	297.76	2 out	6 out	12 out	13 out	2.10

Table 2 shows the average result of the NFA instances. The NFA constraints used in the instances are much smaller than the corresponding MDDs, and the resulting encodings of BCT constraints also fit in memory. However, for MDDs being larger, the MDD CNF encodings BaP, LevP, NNFP and ComP run out of memory (memory-out). For these encodings, there are 2, 6, 12 and 13 memory-out NFA instances in the NFA-54 series, respectively. We remark that if an CNF encoding becomes too large for an instance, it simply cannot be used. The average initialization time of encoding MDD constraints are much larger than that of encoding BCT constraints, e.g. the Itime of Min is 290 seconds but that of MS is less than 1 second. The CNF encodings of BCT constraints are much faster than those of MDD constraints. The MS and PS encodings can solve all 90 NFA instances in 10 minutes but the best CNF encoding of MDD constraints, i.e. the Min encoding, only solves 86 instances. The best result for this series is the Abscon propagator with the DDeg variable heuristic.

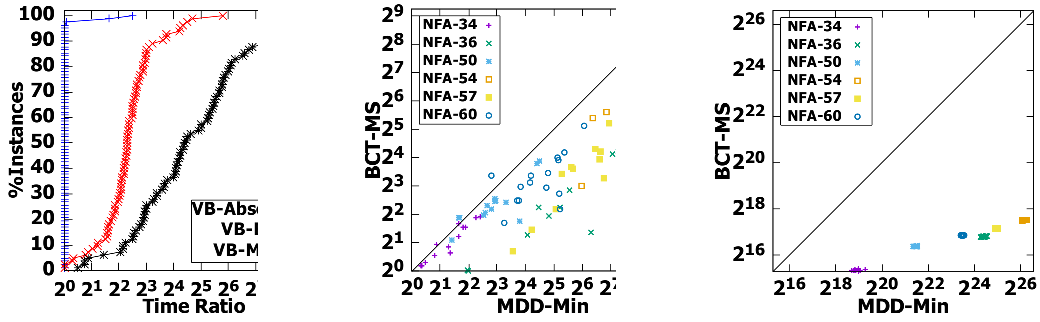
Figure 2a shows that the best overall result for the NFA instances is the VB-Abscon propagator followed by the CNF encodings where VB-BCT overall outperforms VB-MDD on solving the NFA instances. The MS encoding is more compact than the Min encoding, where Min has the best CNF encoding result of MDD constraints for these NFA benchmarks. For example, the number of clauses in MS can be up to 400 times less than that in Min (Figure 1c gives the overall comparison). Correspondingly, MS has potential to be faster than Min. Figure 1b shows that MS is faster than Min on almost all tested NFA instances.

6.2 Benchmark Series 2: Pentominoes

We use all 192 Pentominoes instances from the the pentominoes generator website.[6] Some of the Pentominoes instances were also used in the Minizinc challenge 2021. The instances are separated into 4 series, P-5, P-10, P-15 and P-20, where P-k denotes the instances using

[5] http://www.brics.dk/automaton/

[6] https://github.com/zayenz/minizinc-pentominoes-generator



(a) Virtual best comparison. **(b)** BCT-MS vs MDD-Min. **(c)** The number of clauses.

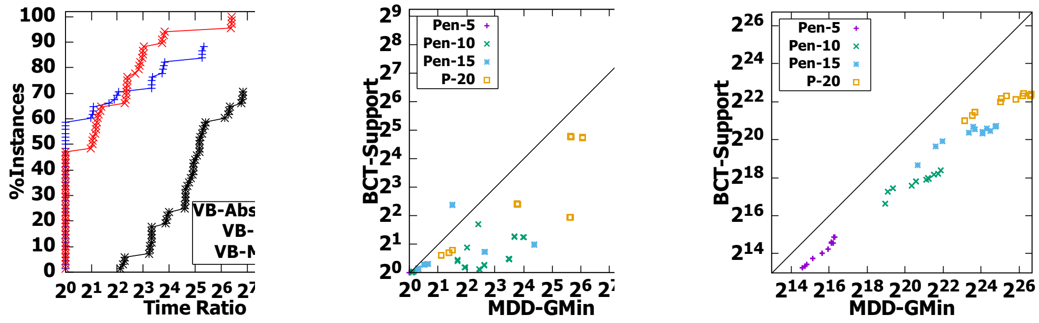
Figure 2 NFA benchmarks.

a $k \times k$ board. The constraints used in these benchmarks are represented with regular expressions (see [27] for more details). We use the dk.brics.automaton library to encode any regular expression into a DFA, and then directly expand the DFA into a MDD. The enocoding from [48] is used to transform any MDD constraint into a BCT constraint.

Table 3 Pentominoes benchmarks.

		BCT					MDD							Abscon
Inst.	#I	Log	Direct	Support	PS	MS	Min	GMin	Tes	BaP	LevP	NNFP	ComP	(Lex.)
P-5	48	0.27	**<0.01**	**<0.01**	**<0.01**	**<0.01**	0.05	0.01	0.06	0.02	0.02	0.02	0.02	0.02
p-10	48	154.70	32.04	**0.60**	0.76	3.62	72.86	4.94	31.65	32.08	17.73	19.42	18.28	9.95
P-15	48	36 out	24 out	**16 out**	20 out	20 out	24 out	24 out	24 out	24 out	24 out	24 out	28 out	20 out
P-20	48	44 out	24 out	16 out	16 out	16 out	24 out	20 out	32 out	36 out	36 out	24 out	28 out	**12 out**
#Sol	192	112	144	**160**	156	156	144	148	136	132	132	144	136	**160**
Itime	192	4.37	3.94	3.39	3.38	3.36	1.23	1.52	2.20	2.48	2.62	2.56	2.73	5.43

Table 3 gives the average result of the Pentominoes instances. The BCT support, PS and MS CNF encodings significantly outperform the MDD CNF encodings. The support encoding is faster or solves more instances than the other CNF encodings on all 4 series. For the MDD CNF encodings, GMin has the best overall result, but the support encoding of BCT constraints can solve 12 more instances than GMin. In addition, the CNF encodings of BCT constraints can be competitive with the Abscon solver. The support encoding gives the best performance on 3 out of 4 Pentominoes series.



(a) Virtual best comparison. **(b)** BCT-Support vs MDD-GMin. **(c)** The number of clauses.

Figure 3 Pentominoes benchmarks.

Figure 3a shows the overall result on Pentominoes. VB-BCT is the best on more than 40% instances, and it can solve 5% more instances than VB-Abscon. Different CNF encodings of BCT constraints solve different instances, thus, VB-BCT can solve more instances than VB-Abscon. From Figure 3c, we can see that the number of clauses of GMin can be much more (up to 20 times more) than that of the support encoding. In addition, GMin is also slower than the support encoding on almost all instances (see Figure 3b).

6.3 Benchmark Series 3: Nurse scheduling

We use four different models of the nurse scheduling problem, namely, N-1, N-2, N-3 and N-4. The nurse scheduling problems come from [8, 18, 48], where nurses are assigned with a day shift, evening shift, night shift or day off for each day. The models have a cardinality [36] constraint per shift and a regular constraint per nurse. The cardinality constraints are used to guarantee that there are enough nurses to meet a demand of each shift. Each model has its own regular constraints as follows:

- In N-1, the model uses regular constraints to restrict that for each 7 days, a nurse work 1 or 2 night shifts, 1 or 2 evening shifts, 1 to 5 day shifts and 2 to 5 days off.
- In N-2, a nurse works 1 or 2 night shifts every 7 days, and 1 or 2 days off every 5 days.
- In N-3, a nurse works 1 or 2 night shifts every 9 days, and 2 or 3 days off every 7 days.
- In N-4, a nurse works 1 or 2 night shifts every 11 days, and 3 or 4 days off every 9 days.

All models restrict that a nurse can only work a second shift after 12 hours of the first. The cardinality and regular constraints are encoded as MDD constraints, and then the MDD constraints are transformed into BCT constraints. For each model, we use 50 instances from the N30 series[7] where the number of nurses of an instance is set to the maximum number of the nurse demand for a day.

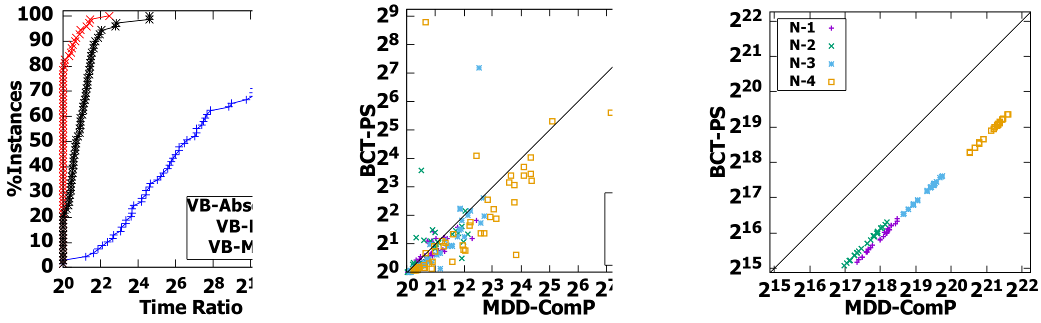
Table 4 Nurse scheduling benchmarks.

		BCT					MDD							Abscon
Inst.	#I	Log	Direct	Support	PS	MS	Min	GMin	Tes	BaP	LevP	NNFP	ComP	(Act.)
N-1	50	1 out	12.88	**0.30**	0.49	3.10	2 out	0.50	0.84	1.09	1.29	0.77	0.64	7 out
N-2	50	11.34	1.86	**0.44**	0.80	0.84	1 out	0.48	0.63	2.95	0.83	0.71	0.72	14 out
N-3	50	2 out	2 out	**1 out**	**1 out**	2 out	4 out	2 out	3 out	**1 out**	**1 out**	**1 out**	**1 out**	13 out
N-4	50	50 out	10 out	3 out	**2 out**	5 out	5 out	4 out	5 out	4 out	**2 out**	3 out	**2 out**	15 out
#Sol	200	147	188	196	**197**	193	188	194	192	195	**197**	196	**197**	151
Itime	200	2.44	1.76	0.57	0.56	0.56	0.45	0.47	0.47	0.49	0.52	0.50	0.51	0.82

Table 4 shows that CNF encodings can overall outperform the Abscon solver for the nurse scheduling instances. The Abscon solver only solves 151/200 instances within timeout but the CNF encodings can solve 197/200 instances. In our detailed results, the CNF encodings of BCT constraints are faster than the CNF encodings of MDD constraints on most instances. For example, the PS encoding is faster than each CNF encoding of MDD constraints on most nurse scheduling instances.

From Figure 4a, we see that VB-BCT is the fastest method on more than 80% instances. VB-BCT can solve 20% more instances than VB-Abscon. The CNF encodings of BCT constraints have better performance than those of MDD constraints. For example, the number of clauses of the PS encoding is around 4 times less than that of the ComP encoding

[7] https://www.projectmanagement.ugent.be/nsp.php



(a) Virtual best comparison. **(b)** BCT-PS vs MDD-ComP. **(c)** The number of clauses.

Figure 4 Nurse scheduling benchmarks.

(shown in Figure 4c), and the PS encoding can be faster than ComP on more than 85% instances (see Figure 4b), where ComP is the best CNF encoding of MDD constraints for the nurse scheduling instances.

6.4 Benchmark Series 4: XCSP

We use five instance series from the XCSP website[8] as they are BDD/MDD instances: bdd-15, bdd-18, mdd-p5 (MDD-half), mdd-p7 (MDD-0.7) and mdd-p9 (MDD-0.9). Some of these instances were also used in the 2019 XCSP competition. The instances bdd-15 and bdd-18 are introduced in [12], and then the instances mdd-p5, mdd-p7 and mdd-p9 are introduced in [10, 49], where mdd-pk is a MDD with sharing probability $\frac{k}{10}$ (see [10, 49] for more details).

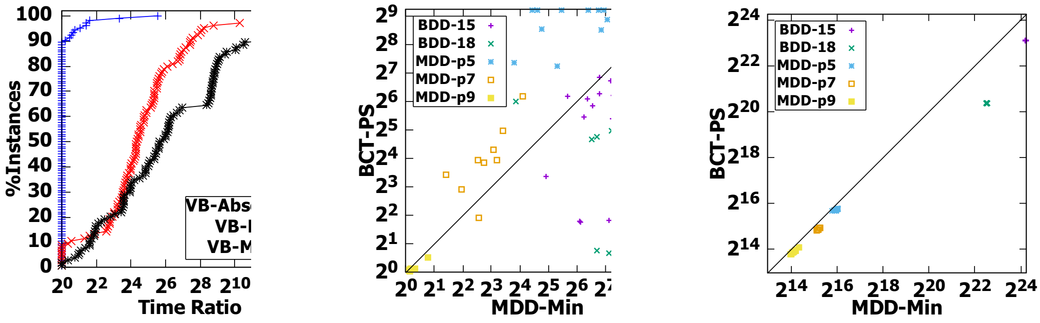
Table 5 XCSP benchmarks.

		BCT					MDD							Abscon
Inst.	#	Log	Direct	Support	PS	MS	Min	GMin	Tes	BaP	LevP	NNFP	ComP	(Act.)
bdd-15	35	35 out	222.99	106.44	67.67	33.05	192.29	152.37	16 out	22 out	29 out	28 out	33 out	**2.42**
bdd-18	35	409.58	129.94	79.64	68.77	22.86	332.11	26 out	29 out	22 out	30 out	25 out	31 out	**0.77**
mdd-p5	25	22 out	23 out	13 out	14 out	13 out	1 out	14 out	16 out	17 out	17 out	19 out	17 out	**73.06**
mdd-p7	9	95.46	68.41	24.73	20.38	23.28	6.82	21.98	44.68	50.12	55.29	39.54	39.58	**1.82**
mdd-p9	10	2.16	0.59	0.19	0.11	0.35	0.18	0.27	0.73	1.08	0.76	0.48	0.46	**0.06**
#Sol	114	57	91	101	100	101	113	74	53	53	38	42	33	**114**
Itime	114	3.96	2.69	1.72	1.64	1.62	1.79	2.21	2.61	2.89	3.03	3.30	3.43	2.09

Table 5 shows that the Abscon solver using the Activity heuristic is the fastest overall for these instances. The Abscon solver can solve all instances while the CNF encodings are time-out on some instances. The CNF encodings of BCT and MDD constraints perform better on different instances. On the bdd-15, bdd-18 and mdd-p9 series, the PS encoding is faster than the CNF encodings of MDD constraints while Min is the best CNF encoding on the mdd-p5 and mdd-p7 series.

Figure 5a shows that VB-Abscon and VB-BCT is the best method on around 90% and 10% instances, respectively. In addition, VB-BCT can be faster than VB-MDD on more than 80% instances. Figure 5b shows the differences between instances, PS is faster than Min on almost all instances in the bdd-15 and bdd-18 series but the opposite happens on the

[8] http://xcsp.org



(a) Virtual best comparison. **(b)** BCT-MS vs MDD-Min. **(c)** The number of clauses.

Figure 5 XCSP benchmarks.

mdd-p5 and mdd-p7 instances, where Min is the best CNF encoding of MDD constraints for these instances. From Figure 5c, we can see that the number of clauses of the PS encoding can be 2-5 times less than that of Min on the bdd-15 and bdd-18 series.

We summarize experiments on all four benchmark series. While there is some initialization and encoding time for all methods, this is overall less significant than the solving time (there are many timeouts for some methods). The initialization time becomes significant when the encoding becomes large, e.g. in the NFA instances, the encoding cost becomes significant in the MDD CNF encodings with some being memory-out. Overall across all four problem series, BCT CNF encodings generally outperform MDD CNF encodings. As with the MDD CNF encoding experiments in [1] where they found performance was mixed between CNF encodings and their propagator comparison, we also find that for some problems the BCT CNF encoding is the best while for other problems the BCT propagator in Abscon is the best. Still BCT CNF encoding is overall competitive or best for many instances and increases the flexibility and choices in solving of BCT (and NFA/MDD) constraints.

7 Conclusion

Binary Constraint Tree (BCT) is more compact than Ordered Multi-valued Decision Diagram (MDD). We show that BCT can be exponentially smaller than MDD when representing NFA constraints. We investigate CNF encodings on BCT constraints which allow solving of BCT constraints with SAT solvers. At the same time, we show this can improve CNF encodings of MDD constraints. We tailor three well-known CNF encodings of binary constraints, i.e. the log encoding, direct encoding and support encoding, to encode BCT constraints. Then we propose two new CNF encodings, partial support encoding and minimal support encoding, which give smaller CNF encodings of BCT constraints. We study and compare the strength of unit propagation on these five CNF encodings of BCT constraints. Our experimental results study our CNF encodings of BCT constraints and also compare with seven existing CNF encodings of MDD constraints on a range of existing benchmarks. Experimental results show that the CNF encodings of BCT constraints can outperform those of MDD constraints. Our results show that solving of BCT constraints as well as NFA/MDD constraints is promising on SAT solvers.

References

1 Ignasi Abío, Graeme Gange, Valentin Mayer-Eichberger, and Peter J Stuckey. On CNF encodings of decision diagrams. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 1–17. Springer, 2016.

2 Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger. A new look at BDDs for pseudo-boolean constraints. *Journal of Artificial Intelligence Research*, 45:443–480, 2012.

3 Ignasi Abío and Peter J Stuckey. Encoding linear constraints into SAT. In *International Conference on Principles and Practice of Constraint Programming*, pages 75–91. Springer, 2014.

4 Jérôme Amilhastre, Hélene Fargier, Alexandre Niveau, and Cédric Pralet. Compiling CSPs: A complexity map of (non-deterministic) multivalued decision diagrams. *International Journal on Artificial Intelligence Tools*, 23(04):1460015, 2014.

5 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 *International Joint Conference on Artificial Intelligence*, pages 555–562, 2017.

6 Lucas Bordeaux and Joao Marques-Silva. Knowledge compilation with empowerment. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 612–624. Springer, 2012.

7 Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *European Conference on Artificial Intelligence*, 2004.

8 Sebastian Brand, Nina Narodytska, Claude-Guy Quimper, Peter Stuckey, and Toby Walsh. Encodings of the Sequence constraint. In *International conference on principles and practice of constraint programming*, pages 210–224. Springer, 2007.

9 Kenil C.K. Cheng, Wei Xia, and Roland H.C. Yap. Space-time tradeoffs for the regular constraint. In *International Conference on Principles and Practice of Constraint Programming*, pages 223–237. Springer, 2012.

10 Kenil C.K. Cheng and Roland H. C. Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.

11 Kenil C.K. Cheng and Roland H.C. Yap. Applying ad-hoc global constraints with the case constraint to still-life. *Constraints*, 11(2-3):91–114, 2006.

12 Kenil C.K. Cheng and Roland H.C. Yap. Maintaining generalized arc consistency on ad-hoc n-ary boolean constraints. In *17th European Conference on Artificial Intelligence*, pages 78–82, 2006.

13 Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366, 1989.

14 Alvaro Del Val. Tractable databases: How to make propositional unit resolution complete through compilation. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 551–561. Elsevier, 1994.

15 Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-Table: efficiently filtering table constraints with reversible sparse bit-sets. In *International Conference on Principles and Practice of Constraint Programming*, pages 207–223, 2016.

16 Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

17 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.

18 Graeme Gange, Peter J Stuckey, and Radoslaw Szymanek. MDD propagators with explanation. *Constraints*, 16(4):407, 2011.

19 Ian P Gent. Arc consistency in SAT. In *European Conference on Artificial Intelligence*, pages 121–125, 2002.

20 Ian P Gent and Peter Nightingale. A new encoding of AllDifferent into SAT. In *International Workshop on Modelling and Reformulating Constraint Satisfaction*, pages 95–110, 2004.

21 Rebecca Gentzel, Laurent Michel, and Willem Jan van Hoeve. Haddock: A language and architecture for decision diagram compilation. In *nternational Conference on Principles and Practice of Constraint Programming*, pages 531–547. Springer, 2020.

22 Willem-Jan van Hoeve, Gilles Pesant, Louis-Martin Rousseau, and Ashish Sabharwal. Revisiting the sequence constraint. In *International conference on principles and practice of constraint programming*, pages 620–634. Springer, 2006.

23 Kazuo Iwama and Shuichi Miyazaki. SAT-variable complexity of hard combinatorial problems. In *IFIP World Computer Congress*, 1994.

24 Simon Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45(3):275–286, 1990.

25 Petr Kucera and Petr Savický. Propagation complete encodings of smooth DNNF theories. *CoRR*, abs/1909.06673, 2019. `arXiv:1909.06673`.

26 Petr Kučera and Petr Savickỳ. Bounds on the size of PC and URC formulas. *Journal of Artificial Intelligence Research*, 69:1395–1420, 2020.

27 Mikael Zayenz Lagerkvist. *Techniques for efficient constraint propagation*. PhD thesis, KTH, 2008.

28 Christophe Lecoutre. STR2: optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.

29 Sylvain Merchez, Christophe Lecoutre, and Frédéric Boussemart. Abscon: A prototype to solve CSPs with abstraction. In *International Conference on Principles and Practice of Constraint Programming*, pages 730–744. Springer, 2001.

30 Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*, 2012.

31 Guillaume Perez and Jean-Charles Régin. Improving GAC-4 for table and MDD constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 606–621. Springer, 2014.

32 Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *International conference on principles and practice of constraint programming*, pages 482–495. Springer, 2004.

33 Claude-Guy Quimper and Toby Walsh. Global grammar constraints. In *International conference on principles and practice of constraint programming*, pages 751–755, 2006.

34 Philippe Refalo. Impact-based search strategies for constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, 2004.

35 Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *National Conference on Artificial Intelligence*, 1994.

36 Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. *National Conference on Artificial Intelligence*, pages 209–215, 1996.

37 Francesca Rossi, Charles J. Petrie, and Vasant Dhar. On the equivalence of constraint satisfaction problems. In *European Conference on Artificial Intelligence*, pages 550–556, 1990.

38 Barbara M Smith and Stuart A Grant. Trying harder to fail first. In *European Conference on Artificial Intelligence*, 1998.

39 Arvind Srinivasan, Timothy Ham, Sharad Malik, and Robert K Brayton. Algorithms for discrete function manipulation. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 92–95, 1990.

40 Kostas Stergiou and Toby Walsh. Encodings of non-binary constraint satisfaction problems. In *AAAI Conference on Artificial Intelligence*, pages 163–168, 1999.

41 Allen Van Gelder. Another look at graph coloring via propositional satisfiability. *Discrete Applied Mathematics*, 156(2):230–243, 2008.

42 Hélene Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Compact-MDD: Efficiently filtering (s)MDD constraints with reversible sparse bit-sets. In *International Joint Conference on Artificial Intelligence*, pages 1383–1389, 2018.

43 Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Extending Compact-Diagram to basic smart multi-valued variable diagrams. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 581–598. Springer, 2019.

44 Toby Walsh. SAT v CSP. In *International Conference on Principles and Practice of Constraint Programming*, pages 441–456, 2000.

45 Ruiwei Wang, Wei Xia, Roland H. C. Yap, and Zhanshan Li. Optimizing simple tabular reduction with a bitwise representation. In *International Joint Conference on Artificial Intelligence*, pages 787–795, 2016.

46 Ruiwei Wang and Roland H. C. Yap. Arc consistency revisited. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 599–615, 2019.

47 Ruiwei Wang and Roland H. C. Yap. Bipartite encoding: A new binary encoding for solving non-binary csps. In *International Joint Conference on Artificial Intelligence*, pages 1184–1191, 2020.

48 Ruiwei Wang and Roland H. C. Yap. Encoding multi-valued decision diagram constraints as binary constraint trees. In *AAAI Conference on Artificial Intelligence*, 2022.

49 Wei Xia and Roland H. C. Yap. Optimizing STR algorithms with tuple compression. In *International Conference on Principles and Practice of Constraint Programming*, pages 724–732, 2013.

50 Roland H. C. Yap, Wei Xia, and Ruiwei Wang. Generalized arc consistency algorithms for table constraints: A summary of algorithmic ideas. In *AAAI Conference on Artificial Intelligence*, pages 13590–13597, 2020.

A Appendix: Proofs

▶ **Theorem 5.** *BCT can be exponentially smaller than MDD on representing NFA constraints.*

Proof. The DTBE of a r arity NFA constraints has $3r+1$ variables and $3r$ binary constraints. The hidden variable domains include at most $max(sn, tn)$ values, where sn and tn are the number of states and transitions in the NFA. In addition, each binary constraint relation has tn tuples. So the size of the DTBE is polynomial in that of the NFA constraint.

The size of the negation of the MDDs (Ordered Multi-valued Decision Diagrams) representing alldifferent constraints is exponential in r [4]. Therefore, the number of nodes in a MDD representing the family of NFA constraints given in Example 2 is exponential in r, where r is constraint arity and the NFA has $r+1$ states and $4r+r^2$ transitions. So BCT can be exponentially smaller than MDD on representing NFA constraints. ◀

▶ **Lemma 10.** *Given a BCT $P = (X, C)$ and a tree order O over X, if a literal $(v_a^{O_1}, true)$ is included in $UP(F)$ and all variable domains in $UP(F)$ are not empty, there is $\tau \in sol(P)$ such that $(O_1, a) \in \tau$ and $(v_b^x, true)$ is included in F for all $(x, b) \in \tau$, where $F = (A, C^A)|_{\tau'}$ and $B^x \subseteq A$ for all $x \in X$ and $T^O \subseteq C^A$ and τ' is a tuple over a subset of A.*

Proof. For any $c \in C$ where $scp(c) = \{O_i, O_j\}$ and $i < j$, if a literal $(v_a^{O_i}, true)$ is included in $UP(F)$, there must be a literal $(v_b^{O_j}, true)$ in $UP(F)$ such that $\{(O_i, a), (O_j, b)\} \in rel(c)$, otherwise unit propagation with the clause $cl(O_i, a, c)$ can remove $(v_a^{O_i}, true)$ from $UP(F)$. Note that the clause $cl(O_i, a, c)$ encodes the implication: if $v_b^{O_j} = false$ for all tuple $\{(O_i, a), (O_j, b)\} \in rel(c)$, then $v_a^{O_i} = false$.

So we can construct a series of tuples $\{\tau_1, ..., \tau_n\}$ such that $n = |X|$ and $\tau_1 = \{(O_1, b_1)\}$ and $b_1 = a$ and for $j > 1$, $\tau_j = \tau_{j-1} \cup \{(O_j, b_j)\}$ and $(v_{b_j}^{O_j}, True)$ is included in $UP(F)$ and $\{(b_i, O_i), (b_j, O_j)\} \in rel(c)$, where c is the only constraint in C such that $scp(c) = \{O_i, O_j\}$ and $i < j$. The tuple τ_n is a solution of P and $(O_1, a) \in \tau_n$. ◀

▶ **Proposition 11.** *The support encoding $F = (A \cup B, T \cup E)$ of BCT constraints (V, P) implements propagation completeness, where P is a BCT (X, C) and $A = \bigcup_{x \in X} A^x$ and $B = \bigcup_{x \in X} B^x$ and $T = \{cl(x, c) | c \in C, x \in scp(c)\}$ and $E = \bigcup_{x \in X} EO(x)$.*

Proof. Assume $F^\tau = UP(F|_\tau)$ and all variable domains in F^τ are not empty where τ is a tuple over a subset of $A \cup B$. The ladder encoding implements completeness propagation, therefore, for any $x \in X$, if F^τ includes a literal l of a variable in $A^x \cup B^x$, then there is a tuple $t(x, a)$ such that F^τ includes $t(x, a)$ and $t(x, a) \in sol(A^x \cup B^x, EO(x))$ and $l \in t(x, a)$ and $(v_a^x, true) \in t(x, a)$. We can set x as root and construct a tree order O over X such that $O_1 = x$, thus, there is a solution of P including (x, a) which corresponds to a solution of $F|_\tau$ including l (based on Lemma 10). So F implements propagation completeness. ◀

▶ **Proposition 13.** *The partial support encoding $F = (A^V \cup B^V \cup B^H, T \cup E^V)$ of a BCT constraint (V, P) implements unit refutation completeness where $P = (X, C)$.*

Proof. Let $x \in X$ and $F^\tau = UP(F|_\tau)$ where τ is a tuple over a subset of $A^V \cup B^V \cup B^H$. If all variable domains in F^τ are not empty, there is $a \in \mathcal{D}(x)$ such that $(v_a^x, true)$ is in F^τ, otherwise unit propagation with $EO(x)$ can remove all values of the variables in $A^x \cup B^x$, since the ladder encoding implements propagation completeness and every tuple in $sol(A^x \cup B^x, EO(x))$ includes at least a value $true$ of a variable in B^x. Therefore, there is a solution of P including (x, a) which corresponds to a solution of $F|_\tau$ including $(v_a^x, true)$ by setting x as root (based on Lemma 10). So F implements unit refutation completeness. ◀

▶ **Proposition 14.** *The partial support encoding $F = (A^V \cup B^V \cup B^H, T \cup E^V)$ of a BCT constraint (V, P) implements domain consistency where $P = (X, C)$.*

Proof. Let $x \in V$ and $F^\tau = UP(F|_\tau)$ where τ is a tuple over a subset of $A^V \cup B^V$ and all variable domains in F^τ are not empty. If a literal l of a variable in $A^x \cup B^x$ is included in F^τ, there is a value $a \in \mathcal{D}(x)$ such that $l \in t(x, a)$ and $(v_a^x, true) \in t(x, a)$ and $(v_a^x, true)$ is included in F^τ (since ladder encoding implements propagation completeness). So there is a solution of P including (x, a) which corresponds to a solution of $F|_\tau$ including l (Lemma 10 by setting x as root). Hence, the partial support encoding implements domain consistency. ◀

▶ **Proposition 16.** *The minimal support encoding F of a BCT constraint (V, P) with respect to a tree order O implements unit refutation completeness where $x = O_1$ and $P = (X, C)$.*

Proof. Let $F^\tau = UP(F|_\tau)$ where τ is a tuple over a subset of $A^V \cup B^V \cup B^H$. If all variable domains in F^τ are not empty, there is $a \in \mathcal{D}(x)$ such that $(v_a^x, true)$ is in F^τ, otherwise unit propagation with $EO(x)$ can remove all values of the variables in $A^x \cup B^x$, since ladder encoding implements propagation completeness and every tuple in $sol(A^x \cup B^x, EO(x))$ includes at least a value $true$. Therefore, there is a solution of P including (x, a) which corresponds to a solution of $F|_\tau$ including $(v_a^x, true)$ based on Lemma 10 (where x is set as root). So F implements unit refutation completeness. ◀

Modeling and Solving Parallel Machine Scheduling with Contamination Constraints in the Agricultural Industry

Felix Winter[1]
Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, DBAI, TU Wien, Austria

Sebastian Meiswinkel
MCP Algorithm Factory, MCP GmbH, Wien, Austria

Nysret Musliu
Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, DBAI, TU Wien, Austria

Daniel Walkiewicz
MCP Algorithm Factory, MCP GmbH, Wien, Austria

Abstract

Modern-day factories of the agricultural industry need to produce and distribute large amounts of compound feed to handle the daily demands of livestock farming. As a highly-automated production process is utilized to fulfill the large-scale requirements in this domain, finding efficient machine schedules is a challenging task which requires the consideration of complex constraints and the execution of optional cleaning jobs to prevent a contamination of the final products. Furthermore, it is critical to minimize job tardiness in the schedule, since the truck routes which are used to distribute the products to customers are sensitive to delays. Thus, there is a strong need for efficient automated methods which are able to produce optimized schedules in this domain.

This paper formally introduces a novel real-life problem from this area and investigates constraint-modeling techniques as well as a metaheuristic approach to efficiently solve practical scenarios. In particular, we investigate two innovative constraint programming model variants as well as a mixed integer quadratic programming formulation to model the contamination constraints which require an efficient utilization of variables with a continuous domain. To tackle large-scale instances, we additionally provide a local search approach based on simulated annealing that utilizes problem-specific neighborhood operators.

We provide a set of new real-life problem instances that we use in an extensive experimental evaluation of all proposed approaches. Computational results show that our models can be successfully used together with state-of-the-art constraint solvers to provide several optimal results as well as high-quality bounds for many real-life instances. Additionally, the proposed metaheuristic approach could reach many optimal results and delivers the best upper bounds on many of the large practical instances in our experiments.

2012 ACM Subject Classification Computing methodologies → Planning and scheduling

Keywords and phrases Parallel Machine Scheduling, Contamination Constraints, Constraint Programming, Mixed Integer Quadratic Progamming, Metaheuristics, Local Search, Simulated Annealing

Digital Object Identifier 10.4230/LIPIcs.CP.2022.41

Supplementary Material *Software (Source Code, Benchmark Set, and Results)*: `https://doi.org/10.5281/zenodo.6797397`

Funding The financial support by the Austrian Federal Ministry for Digital and Economic Affairs,

[1] Corresponding author

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 41; pp. 41:1–41:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

1 Introduction

In the modern agricultural industry large amounts of compound feed are produced and distributed to fulfill the demands of livestock farming. A highly-automated production environment is used to handle these large-scale requirements, where complex machinery handles the mixing and processing of the numerous ingredients of the compound feed products.

Finding efficient production schedules is a challenging task as several problem-specific constraints regarding contamination levels need to be fulfilled. Furthermore, job tardiness is a critical minimization objective in this domain as trucks are needed to distribute the compound feed products to many consumers and the associated routing is sensitive to delays in production. Currently, human planners create the production schedules either manually or basic greedy algorithms are used to produce solutions that often include a large number of tardy jobs and can hardly fulfill all constraints. Therefore, there is a strong need for novel efficient automated scheduling methods in this area.

In this paper we introduce a novel challenging real-life machine scheduling problem originating from the agricultural industry. As a set of predetermined jobs has to be scheduled on multiple machines where processing times depend on the predecessor job, the problem can be categorized as a parallel machine scheduling problem with setup times (PMSP). Finding efficient schedules for such problems is usually a challenging task and even early basic variants were shown to be NP-hard [3]. Therefore, a plethora of heuristic as well as exact solution approaches were proposed in the past and several surveys such as [4, 2] provide an overview of the related literature.

However, as practical machine scheduling problems appear in many different variations regarding the specified constraints and the objective function, complex large-scale applications are still being investigated in the recent literature. For example, [14] recently proposed a constraint programming (CP) approach to solve a resource-constrained PMSP which includes precedence constraints and aims to minimize job completion time. In [9], another variant with cyclical parallel machines originating from the agricultural industry was approached with mathematical programming as well as an adaptive variable neighborhood based metaheuristic. Another problem considering identical machine scheduling with tool requirements was recently investigated in [6]. In their paper, the authors proposed a matheuristic approach that combines a genetic algorithm together with mathematical programming to efficiently solve practical large-scale instances. In [13], large instances of a bi-objective PMSP with resource constraints during setups was tackled by introducing a novel iterated pareto greedy algorithm. The complexity of another real-life PMSP with setup times and resources originating from the manufacturing industry was analyzed in [5], and the authors proposed mixed integer programming (MIP) models to efficiently solve instances which are based on industrial data. Recently, exact and metaheuristic methods based on simulated annealing and MIP were proposed for another unique PMSP variant from the industry [12].

The problem we investigate in this paper can be compared to previously studied PMSPs as a set of given jobs with sequence-dependent processing times is scheduled to unrelated parallel machines. However, in addition to traditional PMSP constraints, a set of unique contamination level constraints needs to be fulfilled which further requires the consideration of including optional cleaning jobs in the schedule. Modeling these contamination constraints requires auxiliary variables with a continuous domain which makes it challenging to find efficient CP formulations and to the best of our knowledge such constraints have not been investigated for PMSPs in the past. The objective function for the investigated problem variant further includes a domain-specific variant of tardiness minimization, since due dates are defined on tours which are associated to groups of jobs.

In addition to formally introducing a novel real-life PMSP, we provide a set of 19 real-life benchmark instances that represent scenarios from the agricultural industry. As exact approaches to the problem we propose a direct- and an interval variable based CP model together with several programmed search strategies as well as a mixed integer quadratic programming (MIQP) formulation that include novel modeling techniques regarding the contamination constraints and cleaning jobs. Furthermore, we investigate a metaheuristic approach using simulated annealing to efficiently solve large-scale real-life problem instances which utilizes four problem specific neighborhood operators and uses randomly generated initial solutions.

An extensive experimental evaluation of all proposed approaches using the real-life benchmark instances shows that the CP approach is able to provide 9 optimal results and further produces high-quality solutions for all instances. The metaheuristic approach we propose further can reach 8 optimal results, similar upper bounds as obtained by the exact methods for the majority of instances, and two overall best upper bounds.

The remainder of the paper is structured as follows: We provide the problem description in Section 2, before we give the direct CP model in Section 3. Afterwards, a MIQP formulation and an alternative CP model using interval variables are proposed in Sections 4 & 5. In Section 6, we then introduce a metaheuristic approach based on local search. The experimental evaluation of all proposed approaches is discussed on Section 7. Finally, we give concluding remarks at the end of the paper.

2 Problem Description

The main aim of the PMSP variant we investigate is to create efficient schedules on multiple machines for a given set of jobs, where each job has to be scheduled on exactly one of its eligible machines. Furthermore, release dates (i.e. earliest start times) are specified for each job depending on the machine and the processing time of each job depends not only on the machine but also on the previously scheduled job. As several steps regarding the mixing of food products are performed within a job (each job produces a unique mix), there are complex domain-specific rules that determine sequence-dependent processing times to fulfill strict food requirements. Thus, we use sequence-dependent job times instead of setup times to specify this problem.

To avoid contamination of the produced goods maximum contamination levels of several contamination factors further must be respected whenever a job is started. Different ingredients, which are associated with the contamination factors, are used to produce unique food mixes in each job. Each ingredient causes a different contamination change regarding the individual factors, while the contamination needs to stay below a maximum to keep the food clean. The maximum level depends on the particular quantity and ingredient mix and thus is specified per factor/job pair. Further, the contamination reduction varies for each machine and factor, which is given as a reduction factor for each factor/machine pair.

The contamination levels are changed through the execution of the jobs, where each job can affect them differently. For example, a particular job could lower the level of one contamination factor and raise the level for another factor during its execution.

Additionally, optional cleaning jobs can be scheduled to reduce the contamination levels. Cleaning jobs behave similarly to regular jobs as they flush machines using a dummy mix and thereby update individual contamination levels. To fully reset all levels multiple cleaning jobs might be needed. Thus, in contrast to regular jobs they are optional and can be scheduled more than once if necessary. In the agricultural industry usually predetermined uniform lengths are used for the cleaning jobs, as it is challenging to give guarantees about contamination changes on variable lengths.

Finally, the main aim of the problem we investigate is to minimize tour tardiness, where each tour represents a truck route that is associated to a group of jobs that produce supplies for that tour. The food mixes each job produces are loaded on trucks which start delivery as soon as they are fully loaded, where food from a single job can be distributed on several tours and to multiple customers within a tour. The tour planning involves product forecasts and is not handled in our problem but predetermined in the input (a tour due date is the latest time the truck should start delivery).

Thus, the end time of the last job in the schedule that is associated to that tour is compared against the tour due date to calculate tour tardiness. The minimization objective then aggregates the tardiness over all tours. Table 1 summarizes the formal parameters of an instance to the investigated problem:

Table 1 Parameters for the parallel machine scheduling problem with contamination constraints.

Description	Parameter
Set of non-cleaning jobs	J^*
Set of cleaning jobs	S
Set of all jobs	$J = J^* \cup S$
Set of tours	T
Tours associated to each job	$jobTours_j \subseteq T \quad \forall j \in J^*$
Set of Machines	M
Processing time of job j after predecessor i on machine m	$p_{i,j,m} \in \mathbb{R}^+ \quad \forall i \in J, j \in J, m \in M$
Processing time of job i on machine m at the start of the schedule	$b_{i,m} \in \mathbb{R}^+ \quad \forall i \in J, m \in M$
Set of eligible machines per job	$E_j \subseteq M \quad \forall j \in J^*$
Tour due dates	$d_t \in \mathbb{N} \quad \forall t \in T$
Job release dates	$r_{j,m} \in \mathbb{N} \quad \forall j \in J, m \in M$
Set of contamination factors	K
Maximum contamination per factor and job	$C^{\max}_{k,j} \in \mathbb{R}^+ \quad \forall k \in K, j \in J$
Contamination volume per job	$C_{k,j} \in \mathbb{R}^+ \quad \forall j \in J, k \in K$
Initial contamination per machine	$C^0_{k,m} \in \mathbb{R}^+ \quad \forall k \in K, m \in M$
Contamination reduction multiplier per factor and machine	$RF_{k,m} \in [0,1] \quad \forall k \in K, m \in M$
Bound on the scheduling horizon	$h \in \mathbb{N}$
Bound on the contamination level	$u \in \mathbb{R}^+$
Bound on the tardiness of a tour	$v \in \mathbb{N}$

To further illustrate the investigated problem, Figure 1 visualizes a schedule for a simple toy example instance with jobs $J1 - J6$, a cleaning job C, and two machines M_1, M_2. In this example, J_1 is scheduled first on M_1 at time 0 and ends at time 2. Afterwards, J_3 is executed before C is scheduled at time 5. Finally, after a short break J_4 is started at time 8. J_4 cannot directly start after J_3 ends, as its release date is 8 in this example ($r_{J_4,M_1} = 8$). J_2 is scheduled first on M_2 and starts at time 1, as $r_{J_2,M_2} = 1$. After completion of J_2, J_5 starts just at time 4 since $r_{J_5,M_2} = 4$. Finally, J_6 is scheduled at time 6 and ends at time 9.

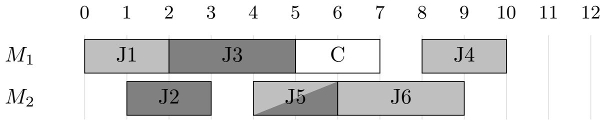


Figure 1 An example schedule for 6 jobs and a cleaning job on two machines.

Note that the jobs have processing times of 2 or 3 in the example, which are determined by the corresponding predecessor job. Therefore, the job lengths could be different if the job order was changed. The background colors of each job indicate the associated tours. In the example, jobs with a dark gray color belong to tour T_1, whereas light gray jobs belong to tour T_2. C has a white background and is not associated to any tour (which is always the case with cleaning jobs as they do not provide any demands), and J_5 is actually associated to both T_1 and T_2. Thus, production for T_1 is finished with the end of J_5 at time 6 and tour T_2 is completed at time 10. Let the tour due dates for this example be $d_{T_1} = 5$ and $d_{T_2} = 8$. Then T_1 would be late by one time unit and T_2 would be late by two units.

Figure 2 further illustrates the change in contamination levels on M_1 regarding two contamination factors $K = \{k_1, k_2\}$. Both factors start at level 0 at time point 0 but get raised through the execution of J_1 to 0.2 and 0.5. The execution of J_3 actually lowers the level of k_2 to 0.2, however, k_1 is raised to 1.5 by J_3 as it causes a strong contamination regarding that factor. C then reduces contamination for both factors down to $k_1 = 0.2, k_2 = 0$ so that J_4 can be scheduled, which again increases the contamination levels regarding both factors.

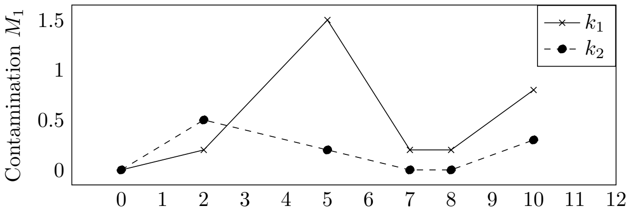


Figure 2 Contamination levels for factors k_1 and k_2 for the jobs scheduled on machine M_1 in the example schedule shown in Figure 1.

3 Constraint Programming Formulation

In this section, we propose a direct CP formulation for the PMSP with contamination constraints. The model serves as a formal specification of the problem, but can also be used as an exact solution approach together with a CP solver.

3.1 Decision Variables

The direct CP model uses decision variables that represent the job predecessors and thereby determines the complete job sequence on each machine:

- Predecessors of regular jobs: $x_j \in J \cup M \quad \forall j \in J^*$
- Cleaning job predecessors: $x_s \in J \cup M \cup \{\bot\} \quad \forall s \in S$

A predecessor either is another job or a machine, in the latter case the associated job starts the machine schedule. Cleaning jobs additionally can have their predecessor set to $\bot$ in which case they are not scheduled at all. Further, the following auxiliary variables are specified:

- Machine assignment for each job: $y_j \in M \quad \forall j \in J^*$
- Machine assignment for each cleaning job: $y_j \in M \cup \{\bot\} \quad \forall j \in S$
- Start time of each job: $start_j \in \mathbb{N} \quad \forall j \in J$
- End time of each job: $end_j \in \mathbb{N} \quad \forall j \in J$
- End time of each tour: $end_t \in \mathbb{N} \quad \forall t \in T$
- Contamination level before each job: $c_{k,j} \in \mathbb{R}^+ \quad \forall k \in K, j \in J$

These variables define the machine assignments for each job (if a cleaning job is not used its machine assignment is ⊥) in addition to start- and end times for jobs and tours. Furthermore, the contamination states before each job are captured by a set of contamination level variables.

3.2 Constraints

In the following we formally specify the constraints of the investigated problem (Note that we implicitly make use of the element global constraint and constraint reification):

- Job start times need to be greater than or equal to the job release date:

$$start_j \geq r_{j,y_j} \quad \forall j \in J \tag{1}$$

- Tour end time variables should be greater than or equal to the associated job end times:

$$end_t \geq end_j \quad \forall j \in J, t \in jobTours_j \tag{2}$$

- Jobs can only be assigned to eligible machines:

$$y_j \in E_j \quad \forall j \in J^* \tag{3}$$

- Channel predecessor variables with start- and end time variables:

$$(x_j \in J) \Rightarrow (start_j > end_{x_j} \wedge end_j = start_j + p_{(x_j),j,(y_j)} \wedge y_j = y_{(x_j)}) \quad \forall j \in J \tag{4}$$

$$(x_j \in M) \Rightarrow (start_j \geq 0 \wedge end_j = start_j + b_{j,(y_j)} \wedge y_j = x_j) \quad \forall j \in J \tag{5}$$

$$(x_j = \bot) \Rightarrow (start_j = 0 \wedge end_j = 0 \wedge y_j = \bot) \quad \forall j \in S \tag{6}$$

- All predecessor assignments need to be different (unless they are set to ⊥):

$$x_{(j_1)} \neq x_{(j_2)} \vee x_{(j_1)} = \bot \quad \forall j_1, j_2 \in J \text{ where } j_1 \neq j_2 \tag{7}$$

- Contamination levels must stay below the maximum values:

$$(c_{k,j} + C_{k,j}) \cdot (1 - RF_{k,(y_j)}) < C^{max}_{k,j} \quad \forall k \in K, j \in J \text{ where } x_j \in J \tag{8}$$

- Contamination levels at the beginning must be greater or equal to the initial state:

$$c_{k,j} \geq C^0_{k,(x_j)} \quad \forall k \in K, j \in J \text{ where } x_j \in M \tag{9}$$

- Contamination levels are updated based on the job sequence:

$$c_{k,j} \geq RF_{k,(y_j)} \cdot (c_{k,(x_j)} + C_{k,(x_j)}) \quad \forall k \in K, j \in J \text{ where } x_j \in J \tag{10}$$

3.3 Objective Function

As in the practical application it is better to have several tours that are a bit late than having a single tour that is late by a large amount of time, the tardiness of each tour is squared in the objective function:

$$minimize \quad \sum_{t \in T} \max\{0, end_t - d_t\}^2 \tag{11}$$

4 Mixed Integer Quadratic Programming Model

In this section, we specify a MIQP formulation of the PMSP with contamination and can be used as an exact solution approach with state-of-the-art MIQP solvers.

4.1 Decision Variables

A set of Boolean decision variables is used to determine the job sequence on all machines by capturing the job predecessors. Additionally, several auxiliary variables determine machine assignments, start- and end times, contamination levels, and tour tardiness:

- Boolean job predecessor variables that are set to 1 if and only if job i is a direct predecessor of job j on machine m (ω is a dummy job indicating the initial machine state):

$$x_{i,j,m} \in \{0,1\} \quad \forall i,j \in J'(J' = J \cup \{\omega\}), m \in M \tag{12}$$

- Boolean machine assignment variables:

$$y_{j,m} \in \{0,1\} \quad \forall j \in J, m \in M \tag{13}$$

- Start time of each job: $start_j \in \{0,\dots,h\} \quad \forall j \in J$
- End time of each job: $end_j \in \{0,\dots,h\} \quad \forall j \in J$
- End time of each tour: $end_t \in \{0,\dots,h\} \quad \forall t \in T$
- Contamination level before each job: $c_{k,j} \in [0,u] \quad \forall k \in K, j \in J$
- Tour tardiness: $tardiness_t \in \{0,\dots,v\} \quad \forall t \in T$

4.2 Constraints

The following list of linear constraints are used in the MIQP formulation:

- Each job can be used as at most one predecessor:

$$\sum_{j \in J \setminus \{i\}, m \in M} x_{i,j,m} \leq 1 \quad \forall i \in J \tag{14}$$

- Each machine can be used as at most one predecessor:

$$\sum_{j \in J} x_{0,j,m} \leq 1 \quad \forall m \in M \tag{15}$$

- Any job that is used as a predecessor also needs to have a single predecessor itself:

$$\sum_{m \in M, i \in J' \setminus \{j\}} x_{i,j,m} = \sum_{m \in M, i \in J' \setminus \{j\}} x_{j,i,m} \quad \forall j \in J \tag{16}$$

- If a job j has a predecessor i, i must have another predecessor on the same machine:

$$\sum_{k \in J' \setminus \{i\}} x_{k,i,m} \geq x_{i,j,m} \quad \forall i \in J, j \in J, m \in M \tag{17}$$

- If a job has a predecessor on a machine, it also needs to be assigned on the same machine:

$$\sum_{i \in J' \setminus \{j\}} x_{i,j,m} = y_{j,m} \quad \forall j \in J, m \in M \tag{18}$$

- If a job is a predecessor on a machine, it also needs to be assigned on the same machine:

$$\sum_{j \in J' \setminus \{i\}} x_{i,j,m} = y_{i,m} \quad \forall i \in J, m \in M \tag{19}$$

- Each job can only be assigned to eligible machines:

$$\sum_{m \in E_j} y_{j,m} = 1 \quad \forall j \in J^*, \quad \sum_{m \in E_j} y_{j,m} \leq 1 \quad \forall j \in S, \quad \sum_{m \in M \setminus E_j} y_{j,m} = 0 \quad \forall j \in J \tag{20}$$

- Each job needs a predecessor on an eligible machine (cleaning jobs may have one):

$$\sum_{i \in J, m \in E_j} x_{i,j,m} = 1 \quad \forall j \in J^*, \quad \sum_{i \in J, m \in E_j} x_{i,j,m} \leq 1 \quad \forall j \in S, \quad \sum_{i \in J, m \in M \setminus E_j} x_{i,j,m} = 0 \quad \forall j \in J \tag{21}$$

- Release dates have to be respected:

$$start_j \geq r_{j,m} - (1 - y_{j,m}) \cdot h \quad \forall j \in J, m \in M \tag{22}$$

- Channel tour end times to the job end times:

$$end_t \geq end_j \quad \forall j \in J, t \in jobTours_j \tag{23}$$

- Channel job start- and end times with the job predecessors (this also prevents cyclic predecessor assignments):

$$start_j > end_i - (1 - \sum_{m \in E_j} x_{i,j,m}) \cdot (h+1) \quad \forall i \in J', j \in J \tag{24}$$

$$end_j \geq start_j + \sum_{m \in E_j} (x_{i,j,m} \cdot p_{j,m,i}) - (1 - \sum_{m \in E_j} x_{i,j,m}) \cdot (h+1) \quad \forall i \in J', j \in J \tag{25}$$

- Contamination levels must stay below the maximum values:

$$(c_{k,j} + C_{k,j}) \cdot (1 - RF_{k,m}) < C_{k,j}^{max} + (1 - x_{i,j,m}) \cdot (1+u) \quad \forall k \in K, i \in J', j \in J, m \in E_j \tag{26}$$

- Set initial contamination levels for jobs at the start of the schedule:

$$c_{k,j} \geq C_{k,m}^0 - (1 - x_{0,j,m}) \cdot (1+u) \quad \forall k \in K, j \in J, m \in E_j \tag{27}$$

- Set contamination levels in the sequence based on the job predecessor assignments:

$$c_{k,j} \geq RF_{k,m} \cdot (c_{k,i} + C_{k,i}) - (1 - x_{i,j,m}) \cdot (1+u) \quad \forall k \in K, i \in J, j \in J, m \in E_j \tag{28}$$

- Channel the tour tardiness variables to tour end time variables:

$$tardiness_t \geq end_t - d_t \tag{29}$$

4.3 Objective Function

The objective function aims to minimize the sum of each squared tour tardiness:

$$minimize \quad \sum_{t \in T} tardiness_t^2 \tag{30}$$

5 Alternative Constraint Programming Model using Interval Variables

In this section, we propose an alternative CP model that utilizes a widely used technique to capture the scheduling aspects of the investigated problem with optional interval variables and specialized global constraints [10, 11].

As alrady mentioned in Section 2 the practical application uses sequence-dependent processing times instead of setup times in its input parameters. However, in the interval based model we want to utilize efficient scheduling global constraints that only accept setup-time based input. Thus, we transform the sequence-dependent processing times into equivalent shorter processing and appropriate setup times in a preprocessing phase for this formulation (by taking the overall minimum processing time and calculating setup times based on the differences regarding each job predecessor).

We specify the following additional input parameters needed by this formulation:

- Setup time between two jobs on each machine: $s_{m,i,j} \in \mathbb{N} \quad \forall m \in M, i \in J', j \in J$
- Processing time (not sequence-dependent) of each job (ω has a processing time of 0): $p_j \in \mathbb{N} \quad \forall j \in J'$
- Minimum release time: $m_j = \min\{r_{j,m} | m \in M\} \quad \forall j \in J$
- Contamination volume for unused job positions is 0: $C_{k,0} = 0 \quad \forall k \in K$
- Acceptable difference for floating point comparison: $\epsilon \in \mathbb{R}^+$
- All jobs associated to a tour: $tourJobs_t = \{j \in J | t \in jobTours_j\}$

5.1 Decision Variables

We use optional interval variables which formally are decision variables whose domain values are a convex interval: $\{\perp\} \cup \{[s, e)|s, e \in \mathbb{Z}, s \leq e\}$, where s and e are the start- and end times of the interval and $\perp$ is a special value indicating that the interval is not scheduled at all. The start- and end times of such a variable var can be accessed via functions $startOf(var), endOf(var)$, and the function $presenceOf(var)$ returns true if and only if an optional interval is scheduled. Furthermore, several global constraints can be defined on interval variables. For example the $alternative(var, V)$ global constraint ensures that exactly one of the interval variables in the set V must have identical start- and end times to the interval variable var. We specify interval variables (optional interval variables) with the notation $intervalVar(p, [l, b])$ ($optIntervalVar(p, [l, b])$), where p denotes the processing time and $[l, b]$ specifies the time period in which the interval may be scheduled.

In addition to the interval variables, we make use of sequence variables which capture a permutation over a given set of interval variables and thereby express the sequence of all present interval variables. Thus, a sequence variable π can be used with global constraints such as $noOverlap(\pi)$, which ensures that all intervals in the sequence do not interfere temporally. Furthermore, sequence variables can serve as arguments for functions such as $first(\pi, var)$ and $typeOfPrev(\pi, var, -1)$, where $first(\pi, var)$ ensures that the interval variable var is scheduled first in sequence π and $typeOfPrev(\pi, var, -1)$ returns the job id which is the predecessor of interval var in sequence π or -1 if var is scheduled at the start of the sequence. The following decision variables are used in the interval variable based model:

- Interval variables for jobs and optional interval variables for cleaning jobs:
$$x_j : \text{intervalVar}(p_j, [m_j, h]) \quad \forall j \in J^*, \quad x_s : \text{optIntervalVar}(p_s, [m_s, h]) \quad \forall s \in S \tag{31}$$
- Optional interval variables that model machine assignments for each job:
$$xm_{m,j} : \text{optIntervalVar}(p_j, [r_{j,m}, h]) \quad \forall j \in J', m \in M \tag{32}$$
- A sequence variable for each machine:
$$\pi_m : \text{seq}(\{xm_{m,j}|j \in J'\}, J') \quad \forall m \in M \tag{33}$$

These interval variables are sufficient to determine the full schedule. However, to model the contamination levels we further need sets of auxiliary variables that capture the positions of jobs as well as auxiliary variables with a continuous domain that determine the contamination levels in the sequence. As we designed the interval based model for the use with CPoptimizer [11], which does not support floating point variables but only dynamic floating point expressions, we cannot rely on the recursive formulation of the contamination level constraints that was used for the models proposed in sections 3 & 4. Instead, we introduce additional auxiliary variables to represent all possible job positions which can be used together with floating point dynamic expressions for the contamination constraints:

- Variables storing the job positions (0 is used if the job is not scheduled on that machine):
$$jobPos_{m,j} \in \{0, \ldots, |J|\} \quad \forall m \in M, j \in J \tag{34}$$
- Variables storing the job scheduled at each position (0 is used if no job is scheduled):
$$jobAt_{m,j} \in \{0\} \cup J \quad \forall m \in M, j \in \{1, \ldots, |J|\} \tag{35}$$
- Variables storing the job predecessors (0 if the job is not scheduled on the machine, -1 if it has no predecessor):
$$prev_{m,j} \in \{-1, 0\} \cup J' \quad \forall m \in M, j \in J \tag{36}$$
- Dynamic expressions representing the contamination levels before each job position:
$$c_{k,m,0} = C^0_{k,m}, \quad c_{k,m,i} = RF_{k,m} \cdot (c_{k,m,(i-1)} + C_{k,(jobAt_{m,i})}) \quad \forall k \in K, m \in M, i \in \{1, \ldots, |J|\} \tag{37}$$
- Dynamic expressions representing the contamination volume at each position:
$$cv_{k,m,i} = (c_{k,m,i-1} + C_{k,(jobAt_{m,i})}) \cdot (1 - RF_{k,m}) \quad \forall k \in K, m \in M, i \in \{1, \ldots, |J|\} \tag{38}$$

5.2 Constraints

The following set of constraints are specified for the interval variable based model:

- Release dates must be respected for each present job interval:

$$\text{presenceOf}(xm_{m,j}) \Rightarrow (\text{startOf}(xm_{m,j}) - s_{m,(prev_{m,j}),j} \geq r_{j,m}) \quad \forall j \in J, m \in M \tag{39}$$

- Each job can only be scheduled on one machine:

$$\text{alternative}(x_j, \{xm_{m,j} | m \in M\}) \quad \forall j \in J \tag{40}$$

- The dummy job ω is scheduled as the first job on all machines:

$$\text{first}(\pi_m, xm_{m,\omega}) \wedge \text{presenceOf}(xm_{m,\omega}) \quad \forall m \in M \tag{41}$$

- All jobs in a machine sequence must not overlap (considering also the setup times):

$$\text{noOverlap}(\pi_m, \{s_{m,i,j} | i \in J', j \in J\}) \quad \forall m \in M \tag{42}$$

- Maximum contamination levels must not be violated:

$$cv_{k,m,i} \leq C^{\max}_{k,(jobAt_{m,i})} - \epsilon \quad \forall k \in K, m \in M, i \in \{1, \ldots, |J|\} \tag{43}$$

- Jobs cannot be scheduled on ineligible machines:

$$\neg\text{presenceOf}(xm_{m,j}) \quad \forall j \in J, m \in M \setminus E_j \tag{44}$$

- Channel job interval variables to job predecessor variables:

$$prev_{m,j} = \text{typeOfPrev}(\pi_m, xm_{m,j}, -1) \quad \forall m \in M, j \in J \tag{45}$$

- Channel job predecessor variables to sequence position variables:

$$(prev_{m,j} = \omega) \Leftrightarrow (jobPos_{m,j} = 1) \quad \forall m \in M, j \in J \tag{46}$$

$$(prev_{m,j} = 0) \Leftrightarrow (jobPos_{m,j} = 0) \quad \forall m \in M, j \in J \tag{47}$$

$$(prev_{m,j} = i) \Rightarrow (jobPos_{m,i} = jobPos_{m,j} - 1) \quad \forall m \in M, i, j \in J \tag{48}$$

$$(jobPos_{m,i} = jobPos_{m,j} - 1 \wedge jobPos_{m,i} \neq 0) \Rightarrow (prev_{m,j} = i) \quad \forall m \in M, i, j \in J \tag{49}$$

- Channel sequence position variables to job position variables:

$$(jobPos_{m,i} = j) \Leftrightarrow (jobAt_{m,j} = i) \quad \forall m \in M, i \in J, j \in \{1, \ldots, |J|\} \tag{50}$$

5.3 Objective Function

The objective function minimizes the squared tour tardiness of all tours:

$$minimize \quad \sum_{t \in T} \max(\{0\} \cup \{\text{endOf}(x_j) - d_t | \forall j \in tourJobs_t\})^2 \tag{51}$$

6 Metaheuristic Approach

In this section, we propose a local search approach using simulated annealing for the PMSP with contamination constraints. First, we describe the solution representation, the used cost function, and the generation of initial solutions. Then, we explain the generation of neighborhood solutions and further describe how a simulated annealing based acceptance function is utilized.

We note that a local search approach based on simulated annealing that partly use similar neighborhoods for another PMSP variant was proposed in [12]. However, we further introduce two additional neighborhood operators to deal with the unique properties of the problem proposed in this paper.

6.1 Solution Representation, Cost Function & Initial Solutions

We represent candidate solutions by using arrays storing the sequence of job ids assigned to each machine. The length of each array is set to the maximum number of all jobs, and empty positions in the arrays are aligned to the end and set to a null value. Any candidate solution ensures that regular jobs are scheduled exactly once on any machine, cleaning jobs however are optional. The value of the objective function for a given candidate solution is determined by calculating the earliest possible start time for each job in the sequence (i.e. either directly after the predecessor ends or at the release date). However, as candidate solutions may also cause constraint violations, we additionally include the number of violations V in the cost function $cost(S)$ that is used to evaluate a candidate solution S:

$$cost(S) = \sum_{t \in T} \max\{0, end_t - d_t\}^2 + V \cdot M \tag{52}$$

The cost function adds the number of constraint violations V multiplied by a given factor M to the objective function, where M is set to a large value so that a single hard constraint violation becomes incomparingly more expensive than any objective value. We determine the value V by counting the number of contamination level violations together with the number of eligible machine violations. Thereby, each job scheduled on an ineligible machine is counted as a violation and the number of contamination level violations is calculated by checking if the maximum contamination level is exceeded for each job position and factor.

To randomly construct initial solutions, we shuffle the list of jobs and then simply assign one job after the other to a randomly selected eligible machine. Note that this construction procedure may produce infeasible solutions as contamination level constraints may appear, however, local search usually can quickly repair infeasible solutions.

6.2 Search Neighborhoods

We use four different neighborhood operators for local search:

1. **Swap jobs**: This operator selects two jobs in the schedule and simply swaps their positions, where the jobs can be on the same machine or on different machines.
2. **Shift job**: A job is moved to another position in the schedule. Potential targets are any position between consecutive jobs as well as the start and end of any machine schedule.
3. **Insert cleaning job**: Inserts a single cleaning job into any position of the schedule. Similar as with the shift job neighborhood, the target position can be between any pair of consecutive scheduled jobs, or at the start/end of a machine schedule.
4. **Remove cleaning job**: Removes a single cleaning job that is currently scheduled.

Regarding the *swap jobs* and *shift job* operators, we additionally consider *block swap* and *block shift* versions where the main idea is to swap or shift blocks of up to k consecutively scheduled jobs at once, where k is a parameter given to the algorithm. For example, if $k = 3$ the *swap job* neighborhood would not only consider swapping two single jobs, but could potentially swap two blocks of consecutively scheduled jobs with block lengths of two or three. Similarly, a block shift neighborhood move could shift blocks of jobs to a new position in the schedule. The intuition behind these block moves is that short job sequences that work well regarding job processing times and contamination levels can be moved at once to find improving neighboring solutions without the need of performing solution quality worsening intermediate steps.

When dealing with large-scale real-life instance, exploring the complete neighborhood can quickly become computationally expensive. Thus, in the proposed metaheuristic we do not explore the full neighborhood, but instead randomly select a single move out of the complete neighborhood in each iteration. Thereby, we select a single random move per iteration in 2 steps: First, we randomly select one of the neighborhoods. Then, we uniformly sample a single move from the chosen neighborhood.

6.3 Neighborhood Move Acceptance

After a single random move is selected, we evaluate the change to the current solution's quality that would be caused by the move. Based on the result we then decide whether the move should be applied to the current solution. We use a move acceptance function based on simulated annealing [8] which ensures that a cost-improving move is always accepted, whereas a non-cost-improving move is only accepted with probability p that depends on the change in solution quality as well as the current temperature value T. Equation 53 shows how probability p is calculated based on the costs of the current solution S and the candidate solution S^* which has bigger costs than S.

$$p = exp(\frac{-(cost(S^*) - cost(S))}{T}) \tag{53}$$

Regarding the temperature T, we set the initial temperature T_{init} and the final temperature T_{final} by user defined parameters. The cooling rate, which determines how fast the temperature is lowered after each search iteration, is determined dynamically after each iteration in our approach. Thus, the actual cooling rate which is applied for the next iteration is calculated by looking at the average runtime per move and the remaining time budget. Thereby, we set the cooling rate to a value that ensures that the temperature converges to the final temperature value T_{final} at the end of the runtime, assuming the current average runtime per iteration.

7 Experimental Evaluation

In this section, we present the results from an extensive evaluation of all proposed approaches on realistic problem instances from the industry. First, we describe our experimental environment and the benchmark instances. Afterwards, we present and discuss the detailed computational results.

7.1 Experimental Environment

All experiments were run on a computing cluster with 10 identical nodes, each having 24 cores, an Intel(R) Xeon(R) CPU E5–2650 v4 @ 2.20GHz and 252 GB RAM. To evaluate the MIQP model we used Cplex 20.1 [1] and Gurobi 9.5 [7], whereas we used CPoptimizer 20.1 [11] to evaluate the CP models. As CPoptimizer does not support floating point variables, we had to adapt the direct model from Section 3 so that it uses dynamic expressions to capture the contamination levels for each job position. We did this by using similar modeling techniques regarding the contamination constraints as in Section 5, with the only difference that the auxiliary job position variables were channeled to the predecessor variables from the direct model instead of the interval variables. For CPoptimizer we set the relative optimality gap to 10^{-7}, besides that we used default parameters for all solvers, but restricted them to single-threaded solving. Further, we used reasonable values based on the size of the real-life instances for the model parameters h, u, v and ϵ. The local search parameters were set based on manual tuning trials: $k = 4$, $T_{init} = 10^{12}$, and $T_{end} = 10^{-4}$.

All approaches were given a time limit of 1 hour per instance. As the metaheuristic approach utilizes a randomly created initial solution as well as randomly generated moves, we conducted 10 repeated experimental runs per instance with the local search approach and present the mean costs in the final results.

Table 2 The upper bounds for each instance achieved by the proposed methods.

Inst.	CP direct	CP interval	MIQP Gurobi	MIQP Cplex	LS
I 1	**403617**	**403617**	406472	445837	**403617**
I 2	**394696**	**394696**	412688	**394696**	**394696**
I 3	**25702**	**25702**	30250	27702	**25702**
I 4	**2305**	**2305**	2393	3573	**2305**
I 5	39193760	**33707**	44530	52907	**33707**
I 6	**136288**	**136288**	143421	157525	**136288**
I 7	245418700	**959455**	18291650	24524154	**959455**
I 8	6571216	**20816**	5375830	23556806	**20816**
I 9	4901583	**97586**	15987063		**97586**
I 10		**106274**	32417279		**106274**
I 11	10539330	2434331			**2384267.3**
I 12	324562700	**696805**		135383618	697368.6
I 13	440502600	**1020957**			**1020957**
I 14	910107	**407439**	3098941	1826389	**407439**
I 15	7178927	**1454166**	121659015	51971121	**1454166**
I 16	1144669	**773924**	29362144		776526
I 17		**581404**	22263532	17479750	**581404**
I 18		**33251**	19796349	188558145	**33251**
I 19	1082993		6697489		**259614**

We gathered 19 problem instances that directly represent real-life scheduling scenarios from our industry partners to evaluate the proposed methods on realistic problems. All instances together with the detailed results are available for download at `https://doi.org/10.5281/zenodo.6797397`. Detailed size parameters of the instances can further be found in Appendix A.

We further experimented with different programmed search strategies in early experiments. For the final experimental results presented in this section we used the solver's default search strategy with the direct CP model and selected a search strategy that assigns values to the π sequence variables first for the interval based model, as these strategies performed best in the early experiments. Detailed information about the search strategies and related experiments are given in Appendix B.

Based on feedback from the reviewers we additionally experimented with another variant of the MIQP model that uses smaller big Ms in constraints 22, 25, and 27. Although the best dual bound found within 1 hour of runtime could be slightly improved for most instances, the best upper bound found with MIQP heuristics was better without these changes for the majority of instances. Thus, we decided to not include these changes in our final experiments.

7.2 Computational Results

A summary of the best upper bounds produced by all evaluated approaches is given in Table 2. The table shows the best upper bounds produced by the direct- and interval variable based CP models with CPoptimizer (CP direct/CP interval), the best upper bounds reached by the MIQP model with Gurobi (MIQP Gurobi) and Cplex (MIQP Cplex), as well as the mean cost results achieved over 10 runs with local search (LS). Overall best upper bounds per instance are formatted in bold face, and empty cells denote that no solution could be found within the runtime.

We see in Table 2 that the interval variable based CP model was the overall best performing exact method as it provided best solutions for 17 instances. Only instance 19 could not be solved, whereas the direct CP model and Gurobi could find a solution within the runtime. Local search also achieved best upper bounds in 17 cases, only for instances 11 and 16 the mean cost results were not on par with the CP approach. However, regarding instances 11 and 19 the metaheuristic achieved improved results over the exact methods.

Table 3 The lower bounds for each instance achieved by exact methods.

Inst.	**CP direct**	**CP interval**	**MIQP Gurobi**	**MIQP Cplex**
I 1	369489	**403617**	389058	388824
I 2	98575	**295211**	246813	229617
I 3	9377	**25702**	17273	15845
I 4	1370	**2305**	1936	1521
I 5	14885	**24086**	21284	21284
I 6	72197	**136288**	87066	82979
I 7	306161	**707234**	487155	322028
I 8	11601	**20816**	10517	11008
I 9	72097	**97586**	91988	62341
I 10	35145	**80015**	31561	324
I 11	90555	**412877**	277498	220423
I 12	491242	**604160**	577154	534484
I 13	634342	**1020957**	842490	802717
I 14	276676	**407439**	338875	330226
I 15	764585	**1268739**	1192610	1169638
I 16	412942	**773924**	600003	434803
I 17	352486	**528173**	461746	448509
I 18	20449	**32662**	28694	28694
I 19	223730	224186	**232434**	224946

The metaheuristic actually found solutions of similar quality over all 10 runs for all instances except for instances 11/12 where the solution costs were 2383478/2391371 in the best case and 696805/702441 in the worst case.

Table 3 further displays the best lower bounds achieved with exact methods. Columns 2-6 show from left to right: Lower bounds achieved with the direct CP model (CP direct), the interval variable model (CP interval), and the MIQP model with Gurobi and Cplex (MIQP Gurobi/MIQP Cplex). We see that the interval variable model produced the best lower bounds for instances 1-18, whereas Gurobi achieved the best lower bound for instance 19.

Finally, Table 4 summarizes the overall best lower bounds (LB) and upper bounds achieved by any of the exact methods (Exact) and compares it to the best upper bound achieved over all 10 runs by local search (LS). Further, the table includes the duality gap between the lower bound and the best upper bound in percentage (Gap), as well as the time in seconds required until the best upper bound was found by the exact and local search methods (Time (Exact) and Time (LS)). We see in the results that instances 1,3,4,6,8,9,13,14, and 16 could be solved to optimality by exact methods. Surprisingly, the metaheuristic approach produced cost equivalent or improved results compared to exact methods for most instances, as only for instance 16 a better solution was achieved with CP. This indicates that the proposed metaheuristic approach can efficiently escape local optima and thereby reach high-quality results for real-life instances. However, CP based methods also produced equal or better results for 17 instances and were necessary to guarantee optimal solutions. Additionally, we see in the results that exact methods using CP can find high-quality solutions quickly as for several instances the best bound was achieved within 60 seconds of runtime. However, for some instances the best solution was found later during the search process. The best results with local search were mostly found after about half of the given runtime budget was consumed. This is an expected result, given that the used simulated annealing scheme dynamically determines the cooling scheme so that the final temperature is reached at the end of the runtime.

We further compared the best schedules produced by the proposed methods with schedules created by human planners that currently utilize a construction heuristic to generate solutions for the same set of real-life instances in the industry. For proprietary reasons we cannot provide detailed results and additional information about the used heuristic, but our approaches could successfully improve the quality of the schedules and thereby reduce the number of

Table 4 The overall best lower bounds and best results achieved with exact and heuristic methods.

Inst.	**LB**	**Gap**	**Exact**	**Time (Exact)**	**LS**	**Time (LS)**
I 1	403617	0.00	**403617**	65	**403617**	1365.27
I 2	295211	25.21	**394696**	45.22	**394696**	1555.05
I 3	25702	0.00	**25702**	43.08	**25702**	1233.50
I 4	2305	0.00	**2305**	54.95	**2305**	1394.50
I 5	24086	28.54	**33707**	55.58	**33707**	1515.24
I 6	136288	0.00	**136288**	47.19	**136288**	1544.77
I 7	707234	26.29	**959455**	1142.8	**959455**	1445.58
I 8	20816	0.00	**20816**	70.33	**20816**	1462.24
I 9	97586	0.00	**97586**	128.54	**97586**	1577.79
I 10	80027	24.70	**106274**	859.68	**106274**	1745.08
I 11	412877	82.68	2434331	1225.75	**2383478**	1548.49
I 12	604160	13.30	**696805**	428.97	**696805**	1505.05
I 13	1020957	0.00	**1020957**	70.28	**1020957**	1484.74
I 14	407439	0.00	**407439**	617.8	**407439**	1729.00
I 15	1268739	12.75	**1454166**	2035.72	**1454166**	1615.52
I 16	773924	0.00	**773924**	284.15	776526	1615.68
I 17	528173	9.16	**581404**	228.03	**581404**	1607.26
I 18	32662	1.77	**33251**	281.08	**33251**	1366.23
I 19	232434	10.47	1082993	3600	**259614**	1446.79

delayed tours up to roughly 10%. Reducing delayed tours indicates a decent improvement as tour delays lead to large costs in practice. Furthermore, as the instances are not easy and the manual planning is done by experienced planners, a 10% reduction can be considered as a good enhancement.

8 Conclusion

In this work, we introduced a novel real-life PMSP from the agricultural industry and provided a set of challenging real-life instances that we gathered from industrial partners.

We proposed a direct- and an interval variable based CP approach as well as a MIQP approach and thereby considered alternative modeling techniques to efficiently capture unique aspects such as contamination constraints, optional cleaning jobs, and a tour tardiness objective. Furthermore, we investigated a metaheuristic based on local search and simulated annealing using problem specific neighborhood operators to approach large-scale instances.

An extensive experimental evaluation with the introduced real-life problem instances shows that the CP approach using the interval variable based model was the overall best performing exact method as it provided 9 optimal results and high-quality upper bounds for most instances. Additionally, the metaheuristic could provide solutions to all instances and thereby improved results of exact methods for two large-scale instances.

An interesting subject of future work could be to hybridize the proposed exact and metaheuristic techniques within a large-neighborhood search based approach.

References

1 IBM ILOG CPLEX Optimization Studio 20.1.0. User's manual for cplex, November 2021. URL: `https://www.ibm.com/docs/en/icos/20.1.0?topic=cplex-users-manual`.

2 Ali Allahverdi. The third comprehensive survey on scheduling problems with setup times/costs. *European Journal of Operational Research*, 246(2):345–378, October 2015.

3 Ali Allahverdi, Jatinder N. D Gupta, and Tariq Aldowaisan. A review of scheduling research involving setup considerations. *Omega*, 27(2):219–239, April 1999.

4 Ali Allahverdi, C. T. Ng, T. C. E. Cheng, and Mikhail Y. Kovalyov. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research*, 187(3):985–1032, June 2008.

5 Abdoul Bitar, Stéphane Dauzère-Pérès, and Claude Yugma. Unrelated parallel machine scheduling with new criteria: Complexity and models. *Computers & Operations Research*, 132:105291, August 2021.

6 Quang-Vinh Dang, Thijs van Diessen, Tugce Martagan, and Ivo Adan. A matheuristic for parallel machine scheduling with tool replacements. *European Journal of Operational Research*, 291(2):640–660, June 2021.

7 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022. URL: `https://www.gurobi.com`.

8 S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

9 Chuleeporn Kusoncum, Kanchana Sethanan, Rapeepan Pitakaso, and Richard F. Hartl. Heuristics with novel approaches for cyclical multiple parallel machine scheduling in sugarcane unloading systems. *International Journal of Production Research*, 59(8):2479–2497, April 2021.

10 Philippe Laborie. IBM ILOG CP Optimizer for Detailed Scheduling Illustrated on Three Problems. In Willem-Jan van Hoeve and John N. Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Lecture Notes in Computer Science, pages 148–162, Berlin, Heidelberg, 2009. Springer.

11 Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP optimizer for scheduling. *Constraints*, 23(2):210–250, April 2018.

12 Maximilian Moser, Nysret Musliu, Andrea Schaerf, and Felix Winter. Exact and metaheuristic approaches for unrelated parallel machine scheduling. *Journal of Scheduling*, December 2021.

13 Juan C. Yepes-Borrero, Federico Perea, Rubén Ruiz, and Fulgencia Villa. Bi-objective parallel machine scheduling with additional resources during setups. *European Journal of Operational Research*, 292(2):443–455, July 2021.

14 Pinar Yunusoglu and Seyda Topaloglu Yildiz. Constraint programming approach for multi-resource-constrained unrelated parallel machine scheduling problem with sequence-dependent setup times. *International Journal of Production Research*, 0(0):1–18, February 2021.

A Benchmark Instances

Table 5 summarizes the features of the 19 problem instances that directly represent real-life scheduling scenarios from our industry partners. The table displays in each row the instance id (I1-I19), the number of jobs ($|J|$) , the number of machines ($|M|$), the number of tours ($|T|$), and the number of contamination factors ($|K|$). Furthermore, Table 5 also includes the number of variables and constraints used in the direct CP model (Vars/Cons), the interval based model (I.Vars/I.Cons), and the MIQP model (M.Vars/M.Cons). We see that the MIQP formulation in general uses more variables but less constraints than the CP models. Further, the interval based model uses slightly more variables and constraints compared to the direct CP model.

B Programmed Search Strategies

We evaluated several search strategies for the direct- and interval variable based CP models together with CPoptimizer. These search strategies are based on variable- and value selection heuristics, which determine the order of the explored variables and the value assignments. This can play a critical role in reducing the search space that needs to be enumerated by the CP solver. As the search strategies define heuristics only on a subset of all variables, the solver's default strategy is used after all mentioned variables were fixed. Furthermore, ties are broken lexicographically. Table 6 describes the search strategies used with the direct- and interval variable based models.

Table 5 Size parameters of the real-life benchmark instances.

Inst.	$\|J\|$	$\|M\|$	$\|T\|$	$\|K\|$	**Vars**	**Cons**	**I.Vars**	**I.Cons**	**M.Vars**	**M.Cons**
I 1	26	2	41	58	874	5267470	897	5268554	183289	43798
I 2	16	2	37	58	824	5267209	840	5267828	110164	28379
I 3	15	2	37	58	819	5267184	833	5267761	76058	26854
I 4	18	2	34	58	834	5267264	846	5267965	68021	31473
I 5	28	2	42	58	884	5267522	907	5268710	199545	47099
I 6	21	2	43	58	849	5267341	869	5268178	152501	36753
I 7	76	2	76	58	1124	5268762	1174	5273691	764573	132270
I 8	67	2	78	58	1079	5268529	1128	5272594	658176	115596
I 9	77	2	82	58	1129	5268790	1181	5273814	808008	135722
I 10	94	2	82	58	1214	5269230	1274	5276173	1097447	171276
I 11	61	2	74	58	1049	5268380	1096	5271881	590600	104791
I 12	78	2	90	58	1134	5268816	1200	5273945	884457	137331
I 13	67	2	92	58	1079	5268535	1141	5272580	706558	114135
I 14	68	2	85	58	1084	5268557	1131	5272703	653867	116774
I 15	63	2	90	58	1059	5268428	1115	5272113	647114	107794
I 16	65	2	69	58	1069	5268478	1119	5272341	640025	111787
I 17	66	2	70	58	1074	5268506	1121	5272460	630976	112971
I 18	65	2	73	58	1069	5268481	1121	5272351	645552	111137
I 19	60	2	71	58	1044	5268350	1099	5271769	607280	101845

Table 7 summarizes the results on the 19 instances for all evaluated search strategies with the direct- and interval variable based CP models. Each row shows results for a single search strategy, where Columns 1-5 display from left to right: The model and search strategy (Search), the number of optimal solutions achieved (O), the number of solutions achieved (S), the number of best upper bounds achieved when compared with other strategies using the same model (B), and the number of provided optimality proofs.

The results displayed in Table 7 show that the direct model could find solutions for 16 instances with the solver's default search strategy, whereas other strategies could solve less problem instances, even though *direct-search1* could find more optimal solutions and best upper bounds. No search strategy was able to prove an optimum with the direct model (lower bounds achieved with the interval model were used to determine how many optimal solutions were found with the direct formulation). As *direct-default* could solve the most instances, we only present detailed results achieved by this strategy with the direct model in Section 7.

We further see in the results shown in Table 7 that the interval variable based model could successfully solve 18 instances with several search strategies, whereas the solver's default strategy could solve 13 instances. The search strategies *interval-search3* and *interval-search7* produced the overall best results as they could find 9 optimal solutions, achieved the best upper bounds for 18 instances, and provided 8 optimality proofs. Actually, they achieved the exact same upper bounds, and produced different lower bounds only for two instances. Both search types fix sequence variables first, which indicates that this was the most efficient strategy in our experiments. In Section 7 whenever we refer to the best results produced with the interval variable based model we mean results achieved by the *interval-search3* strategy, only for the best lower bounds presented in Table 3 we combined the best bounds achieved with *interval-search3* and *interval-search7*.

Table 6 Search strategies for the direct- and interval variable based CP models.

Search Strategy	Description
direct-default	Uses the solver's default search strategy for the direct model.
direct-search1	Starts with the assignment of all x decision variables. Variables with the smallest values in their domains are selected first and a minimum value first strategy is used for value selection.
direct-search2	As *direct-search1*, but uses a smallest domain size variable selection heuristic instead of the smallest domain value strategy.
direct-search3	As *direct-search1*, but operates on the $start$ instead of the x variables.
direct-search4	As *direct-search2*, but operates on the $start$ instead of the x variables.
direct-search5	As *direct-search1*, but operates on the $endTour$ instead of the x variables.
direct-search6	As *direct-search1*, but operates on the end instead of the x variables.
direct-search7	As *direct-search2*, but operates on the end instead of the x variables.
interval-default	Uses the solver's default search strategy for the interval variable model.
interval-search1	Assigns values to the xm interval variables first. Each interval chooses a presence status first and then assigns a start- and end time. Interval variables with a small start- and end date are fixed first.
interval-search2	As *interval-search1*, but operates on the x instead of the xm variables.
interval-search3	Assigns values to the π sequence variables first. Thereby, the full order of intervals associated to the sequences and the presence status of each interval are fixed before start- and end times are assigned in a later search phase.
interval-search4	As *interval-search1*, but operates on the xc instead of the xm variables.
interval-search5	Starts with *interval-search3* and then continues with *interval-search1*.
interval-search6	Starts with *interval-search3* and then continues with *interval-search2*.
interval-search7	Starts with *interval-search3* and then continues with *interval-search4*.

Table 7 Summarized results produced with different programmed search strategies.

Search	**O**	**S**	**B**	**P**
direct-default	4	16	9	0
direct-search1	6	12	11	0
direct-search2	4	14	9	0
direct-search3	0	5	0	0
direct-search4	0	5	0	0
direct-search5	0	5	0	0
direct-search6	0	6	0	0
direct-search7	0	6	0	0

Search	**O**	**S**	**B**	**P**
interval-direct	9	13	11	8
interval-search1	7	11	9	6
interval-search2	4	6	5	4
interval-search3	9	18	18	8
interval-search4	6	12	7	5
interval-search5	9	18	17	8
interval-search6	9	18	17	7
interval-search7	9	18	18	8