

27th International Conference on Principles and Practice of Constraint Programming

CP 2021, October 25–29, 2021, Montpellier, France (Virtual
Conference)

Edited by

Laurent D. Michel



Editor

Laurent D. Michel 

University of Connecticut, USA
laurent.michel@uconn.edu

ACM Classification 2012

Mathematics of computing → Solvers; Hardware → Theorem proving and SAT solving; Theory of computation → Constraint and logic programming; Software and its engineering → Constraints; Computing methodologies → Machine learning; Mathematics of computing → Mathematical optimization

ISBN 978-3-95977-211-2

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/978-3-95977-211-2>.

Publication date

October, 2021

Bibliographic information published by the Deutsche Nationalbibliothek

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

License

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0):
<https://creativecommons.org/licenses/by/4.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.CP.2021.0

ISBN 978-3-95977-211-2

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.

Editorial Board

- Luca Aceto (*Chair*, Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Mikolaj Bojanczyk (University of Warsaw, PL)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University - Brno, CZ)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

■ Contents

Preface	
<i>Laurent D. Michel</i>	0:ix–0:x
Organization	
.....	0:xi–0:xiii
List of Authors	
.....	0:xv–0:xxi

Invited Talks

The Bi-Objective Long-Haul Transportation Problem on a Road Network	
<i>Claudia Archetti, Ola Jabali, Andrea Mor, Alberto Simonetto, and M. Grazia Speranza</i>	1:1–1:1
Constrained-Based Differential Privacy	
<i>Ferdinando Fioretto</i>	2:1–2:1
Learning in Local Branching	
<i>Defeng Liu and Andrea Lodi</i>	3:1–3:2

Short Papers

Filtering Isomorphic Models by Invariants	
<i>João Araújo, Choiwah Chow, and Mikoláš Janota</i>	4:1–4:9
Improving Local Search for Structured SAT Formulas via Unit Propagation	
Based Construct and Cut Initialization	
<i>Shaowei Cai, Chuan Luo, Xindi Zhang, and Jian Zhang</i>	5:1–5:10
Unit Propagation with Stable Watches	
<i>Markus Iser and Tomáš Balyo</i>	6:1–6:8
Towards Better Heuristics for Solving Bounded Model Checking Problems	
<i>Anissa Kheireddine, Etienne Renault, and Souheib Baarir</i>	7:1–7:11
Integrating Tree Decompositions into Decision Heuristics of Propositional Model	
Counters	
<i>Tuukka Korhonen and Matti Järvisalo</i>	8:1–8:11
Failure Based Variable Ordering Heuristics for Solving CSPs	
<i>Hongbo Li, Minghao Yin, and Zhanshan Li</i>	9:1–9:10
Generating Magical Performances with Constraint Programming	
<i>Guilherme de Azevedo Silveira</i>	10:1–10:13

Regular Papers

Vehicle Dynamics in Pickup-And-Delivery Problems Using Electric Vehicles	
<i>Saman Ahmadi, Guido Tack, Daniel Harabor, and Philip Kilby</i>	11:1–11:17

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel



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

Building High Strength Mixed Covering Arrays with Constraints <i>Carlos Ansótegui, Jesús Ojeda, and Eduard Torres</i>	12:1–12:17
On How Turing and Singleton Arc Consistency Broke the Enigma Code <i>Valentin Antuori, Tom Portoleau, Louis Rivière, and Emmanuel Hebrard</i>	13:1–13:16
Combining Monte Carlo Tree Search and Depth First Search Methods for a Car Manufacturing Workshop Scheduling Problem <i>Valentin Antuori, Emmanuel Hebrard, Marie-José Huguet, Siham Essodaigui, and Alain Nguyen</i>	14:1–14:16
Practical Bigraphs via Subgraph Isomorphism <i>Blair Archibald, Kyle Burns, Ciaran McCreesh, and Michele Sevegnani</i>	15:1–15:17
The Hybrid Flexible Flowshop with Transportation Times <i>Eddie Armstrong, Michele Garraffa, Barry O’Sullivan, and Helmut Simonis</i>	16:1–16:18
CLR-DRNets: Curriculum Learning with Restarts to Solve Visual Combinatorial Games <i>Yiwei Bai, Di Chen, and Carla P. Gomes</i>	17:1–17:14
An Interval Constraint Programming Approach for Quasi Capture Tube Validation <i>Abderahmane Bedouhene, Bertrand Neveu, Gilles Trombettoni, Luc Jaulin, and Stéphane Le Menec</i>	18:1–18:17
Exhaustive Generation of Benzenoid Structures Sharing Common Patterns <i>Yannick Carissan, Denis Hagebaum-Reignier, Nicolas Prkovic, Cyril Terrioux, and Adrien Varet</i>	19:1–19:18
Combining VSIDS and CHB Using Restarts in SAT <i>Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux</i>	20:1–20:19
On the Tractability of Explaining Decisions of Classifiers <i>Martin C. Cooper and João Marques-Silva</i>	21:1–21:18
A Collection of Constraint Programming Models for the Three-Dimensional Stable Matching Problem with Cyclic Preferences <i>Ágnes Cseh, Guillaume Escamocher, Begüm Genç, and Luis Quesada</i>	22:1–22:19
Bounds on Weighted CSPs Using Constraint Propagation and Super-Reparametrizations <i>Tomáš Dlask, Tomáš Werner, and Simon de Givry</i>	23:1–23:18
Parallel Model Counting with CUDA: Algorithm Engineering for Efficient Hardware Utilization <i>Johannes K. Fichte, Markus Hecher, and Valentin Roland</i>	24:1–24:20
Complications for Computational Experiments from Modern Processors <i>Johannes K. Fichte, Markus Hecher, Ciaran McCreesh, and Anas Shahab</i>	25:1–25:21
A Job Dispatcher for Large and Heterogeneous HPC Systems Running Modern Applications <i>Cristian Galleguillos, Zeynep Kiziltan, and Ricardo Soto</i>	26:1–26:16
The Dungeon Variations Problem Using Constraint Programming <i>Gaël Glorian, Adrien Debesson, Sylvain Yvon-Palot, and Laurent Simon</i>	27:1–27:16

Refined Core Relaxation for Core-Guided MaxSAT Solving <i>Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo</i>	28:1–28:19
A Linear Time Algorithm for the k -Cutset Constraint <i>Nicolas Isoart and Jean-Charles Régin</i>	29:1–29:16
A k -Opt Based Constraint for the TSP <i>Nicolas Isoart and Jean-Charles Régin</i>	30:1–30:16
The Seesaw Algorithm: Function Optimization Using Implicit Hitting Sets <i>Mikoláš Janota, António Morgado, José Fragoso Santos, and Vasco Manquinho</i>	31:1–31:16
Reasoning Short Cuts in Infinite Domain Constraint Satisfaction: Algorithms and Lower Bounds for Backdoors <i>Peter Jonsson, Victor Lagerkvist, and Sebastian Ordyniak</i>	32:1–32:20
Learning TSP Requires Rethinking Generalization <i>Chaitanya K. Joshi, Quentin Cappart, Louis-Martin Rousseau, and Thomas Laurent</i>	33:1–33:21
SAT Modulo Symmetries for Graph Generation <i>Markus Kirchweber and Stefan Szeider</i>	34:1–34:16
Counterfactual Explanations via Inverse Constraint Programming <i>Anton Korikov and J. Christopher Beck</i>	35:1–35:16
Utilizing Constraint Optimization for Industrial Machine Workload Balancing <i>Benjamin Kovács, Pierre Tassel, Wolfgang Kohlenbrein, Philipp Schrott-Kostwein, and Martin Gebser</i>	36:1–36:17
Minimizing Cumulative Batch Processing Time for an Industrial Oven Scheduling Problem <i>Marie-Louise Lackner, Christoph Mrkvicka, Nysret Musliu, Daniel Walkiewicz, and Felix Winter</i>	37:1–37:18
Combining Clause Learning and Branch and Bound for MaxSAT <i>Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He</i>	38:1–38:18
Improving Local Search for Minimum Weighted Connected Dominating Set Problem by Inner-Layer Local Search <i>Bohan Li, Kai Wang, Yiyuan Wang, and Shaowei Cai</i>	39:1–39:16
Automatic Generation of Declarative Models For Differential Cryptanalysis <i>Luc Libralesso, François Delobel, Pascal Lafourcade, and Christine Solnon</i>	40:1–40:18
A Bound-Independent Pruning Technique to Speeding up Tree-Based Complete Search Algorithms for Distributed Constraint Optimization Problems <i>Xiangshuang Liu, Ziyu Chen, Dingding Chen, and Junsong Gao</i>	41:1–41:17
Data Driven VRP: A Neural Network Model to Learn Hidden Preferences for VRP <i>Jayanta Mandi, Rocsildes Canoy, Víctor Bucarey, and Tias Guns</i>	42:1–42:17
Statistical Comparison of Algorithm Performance Through Instance Selection <i>Théo Matricon, Marie Anastacio, Nathanaël Fijalkow, Laurent Simon, and Holger H. Hoos</i>	43:1–43:21

Enabling Incrementality in the Implicit Hitting Set Approach to MaxSAT Under Changing Weights <i>Andreas Niskanen, Jeremias Berg, and Matti Järvisalo</i>	44:1–44:19
Solving the Non-Crossing MAPF with CP <i>Xiao Peng, Christine Solnon, and Olivier Simonin</i>	45:1–45:16
Positive and Negative Length-Bound Reachability Constraints <i>Luis Quesada and Kenneth N. Brown</i>	46:1–46:16
Evaluating the Hardness of SAT Instances Using Evolutionary Optimization Algorithms <i>Alexander Semenov, Daniil Chivilikhin, Artem Pavlenko, Ilya Otpuschennikov, Vladimir Ulyantsev, and Alexey Ignatiev</i>	47:1–47:18
Optimising Training for Service Delivery <i>Ilankaikone Senthoooran, Pierre Le Bodic, and Peter J. Stuckey</i>	48:1–48:15
Human-Centred Feasibility Restoration <i>Ilankaikone Senthoooran, Matthias Klapperstueck, Gleb Belov, Tobias Czauderna, Kevin Leo, Mark Wallace, Michael Wybrow, and Maria Garcia de la Banda</i>	49:1–49:18
SAT-Based Approach for Learning Optimal Decision Trees with Non-Binary Features <i>Pouya Shati, Eldan Cohen, and Sheila McIlraith</i>	50:1–50:16
Pseudo-Boolean Optimization by Implicit Hitting Sets <i>Pavel Smirnov, Jeremias Berg, and Matti Järvisalo</i>	51:1–51:20
An Algorithm-Independent Measure of Progress for Linear Constraint Propagation <i>Boro Sofranac, Ambros Gleixner, and Sebastian Pokutta</i>	52:1–52:17
Differential Programming via OR Methods <i>Shannon Sweitzer and T. K. Satish Kumar</i>	53:1–53:15
Learning Max-CSPs via Active Constraint Acquisition <i>Dimosthenis C. Tsouros and Kostas Stergiou</i>	54:1–54:18
Parallelizing a SAT-Based Product Configurator <i>Nils Merlin Ullmann, Tomáš Balyo, and Michael Klein</i>	55:1–55:18
Solution Sampling with Random Table Constraints <i>Mathieu Vavrille, Charlotte Truchet, and Charles Prud’homme</i>	56:1–56:17
Making Rigorous Linear Programming Practical for Program Analysis <i>Tengbin Wang, Liqian Chen, Taoqing Chen, Guangsheng Fan, and Ji Wang</i>	57:1–57:17
Engineering an Efficient PB-XOR Solver <i>Jiong Yang and Kuldeep S. Meel</i>	58:1–58:20
Automated Random Testing of Numerical Constrained Types <i>Ghiles Ziat, Matthieu Dien, and Vincent Botbol</i>	59:1–59:19
The Effect of Asynchronous Execution and Message Latency on Max-Sum <i>Roie Zivan, Omer Perry, Ben Rachmut, and William Yeoh</i>	60:1–60:18

■ Preface

This volume is the collection of papers presented at the 27th International Conference on Principles and Practice of Constraint Programming (CP 2021), held online during October 25–29, 2021. The conference was meant to be hosted by LIRMM in Montpellier, France. Yet, the evolving pandemic prompted us to revise those plan and opt for an online format to ensure that the conference would remain accessible to as many as possible despite the ongoing health crisis.

There were 129 submissions to the conference. In the end, the Program Committee selected 57 papers spread across several thematic tracks meant to encourage as diverse a participation as possible. I wish to extend my thanks to the *authors of all submissions*. Your manuscripts offered a rich collection of exciting ideas, directions and ongoing work that embody a living community. Laying down one’s idea for peer review is both delicate and difficult, yet critical to the existence of any scientific community. Irrespective of the final outcome for your submission, I wish to thank you all for participating in the process.

The program of the conference featured 3 invited talks, 4 tutorials and 5 thematic tracks featuring both short and long papers. The themes of the tracks are:

■ Technical Track	Chair: Laurent Michel
■ Application Track	Chairs: Louis-Martin Rousseau, Michele Lombardi
■ Operation Research Track	Chair: Willem-Jan van Hoeve
■ Machine Learning Track	Chair: Michela Milano
■ Verification Track	Chair: Nadjib Lazaar

The respective chairs deserve all our support and thanks for managing tracks and providing the necessary oversight throughout the review process.

The Senior Program Committee members were an integral part of the effort as they contributed to the reviewing efforts, but also guided the conversations during the discussion period and produced meta-reviews as summaries of the discussions. They played a key role in providing the necessary input to the selection. The process used 4 weeks to author reviews, 1 week to collect author feedback and 2 weeks to discuss all papers and reach decisions. The process was capped with a synchronous online meeting to finalize the selection. I wish to extend my gratitude to all SPC members with a special nod for attending the live meetings.

The reviewer assignment was produced with an IP model developed by Thomas Schiex and Simon de Givry in 2019, extended by Helmut Simonis in 2020 and further tweaked this year. Despite the tight schedule and the offered extensions, the pool of reviewers delivered hundreds of detailed evaluations (at least 3, sometimes 4) for all the submissions and engaged in lively discussions afterwards. The dedication of such a large group of individuals must be acknowledged as this function is essential to the community.

The conference itself cannot thrive without the active engagement of a whole slate of people who took key responsibilities. I wish to thank Andre Cire (University of Toronto) serving as Workshop Chair, Anastasia Papparizou (CNRS, CRIL, Lens, France) as our Tutorial Chair, Jeremias Berg (University of Helsinki, Finland) who managed the Doctoral Program as well as Zeynep Kiziltan (University of Bologna, Italy) who orchestrated all the video preparation and distribution. While the conference is online, rather than in Montpellier, Carmen Gervet and Philippe Vismara (Montpellier, LIRMM, France) played a critical role

for the overall organization and all the logistics that accompany a conference. Their team included Véronique Rousseau (Communication), Gilles Trombettoni and Clément Carbonnel (Web). Their role and time commitment to enable an online conference is greatly appreciated!

Let me close by extending my thanks to the Executive Committee of the Association for Constraint Programming for their support and their trust in organizing the conference.

July 2021, Tolland CT, USA

Laurent D. Michel

Organization

Senior Program Committee

Maria Garcia de La Banda	Monash University, Australia
Armin Biere	JKU, Linz, Austria
Ian Gent	Saint Andrews University, Linz, United Kingdom
Carmen Gervet	University of Montpellier, Espace-Dev, France
Christophe Lecoutre	University of Artois, CRIL, France
Michela Milano	University of Bologna, Italy
Andrea Rendl	Satalia, United Kingdom
Louis-Martin Rousseau	Ecole Polytechnique de Montréal, Canada
Pierre Schauss	University of UCLouvain, Belgium
Helmut Simonis	Insight Center for Data Analytics, Ireland
Peter J. Stuckey	University of Melbourne, Australia
Kostas Stergiou	University of New Macedonia, Greece
Guido Tack	Monash University, Australia
Gilles Trombetti	University of Montpellier, LIRMM, France
Willem-Jan Van Hoeve	Carnegie Mellon University, USA
Roland Yap	National University of Singapore, Singapore
Standa Zivny	University of Oxford, United Kingdom

Program Committee

Özgür Akgün	University of St Andrews, United Kingdom
Carlos Ansótegui	University of Lleida, Spain
Sebastien Bardin	CEA, France
Roman Barták	Charles University, Czech Republic
Christopher Beck	University of Toronto, Canada
Nicolas Beldiceanu	IMT Atlantique, France
Russell Bent	Los Alamos National Laboratory, USA
Jeremias Berg	Helsinki Institute for Information Technology, Finland
David Bergman	University of Connecticut, USA
Miquel Bofill	University of Girona, Spain
Andrea Borghesi	University of Bologna, Italy
Ken Brown	University College Cork, Ireland
Clément Carbonnel	CNRS, France
Hadrien Cambazard	University of Grenoble, France
Andre Augusto Cire	University of Toronto Scarborough, Canada
Martin Cooper	University of Toulouse, France
Allegra De Filippo	University of Bologna, Italy
Sophie Demassey	Mines ParisTech, France
Catherine Dubois	ENSII, France
Ferdinando Fioretto	Syracuse University, Syracuse
Pierre Flener	Uppsala University, Sweden
David Gerault	University of Surrey, United Kingdom
Simon de Givry	INRAE, France
Vijay Ganesh	University of Waterloo (Canada), France

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).
Editor: Laurent D. Michel



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


Tias Guns	Vrije Universiteit Brussel, Belgium
Tarik Hadzic	United Technologies Research Center, USA
Emmanuel Hebrard	CNRS, France
John Hooker	Carnegie Mellon University, USA
Serdar Kadioglu	Brown University, USA
George Katsirelos	INRAE, France
Zeynep Kiziltan	University of Bologna, Italy
Lars Kotthoff	University of Wyoming, USA
Edward Lam	Monash University, Australia
Jimmy Lee	The Chinese University of Hong Kong, China
Samir Loudni	University of Caen, France
Ciaran McCreesh	University of Glasgow, United Kingdom
Arnaud Malapert	University of Nice-Sophia Antipolis, France
Kuldeep S. Meel	National University of Singapore
Claude Michel	University of Nice-Sophia Antipolis, France
Ian Miguel	University of St Andrews, United Kingdom
Peter Nightingale	University of York, United Kingdom
Justin Pearson	Uppsala University, Sweden
Marie Pelleau	University of Nice-Sophia Antipolis, France
Gilles Pesant	Polytechnique Montréal, Canada
Andreas Podelski	University of Freiburg, Germany
Enrico Pontelli	New Mexico State University, USA
Charles Prud'Homme	IMT Atlantique, France
Claude-Guy Quimper	University of Laval, Canada
Jean-Charles Régin	University of Nice-Sophia Antipolis, France
Michel Rueher	University of Côte d'Azur, France
Thomas Schiex	INRAE, France
Paul Shaw	IBM, France
Mohamed Siala	INSA, France
Neil York Smith	Delft University of Technology, Netherlands
Cyril Terrioux	Aix-Marseille University, France
Charlotte Truchet	University of Nantes, France
Phebe Vayanos	University of Southern California, USA
Hélène Verhaeghe	UCLouvain, Belgium
Petr Vilim	IBM, Czech Republic
Lebbah Yahia	University of Oran, Algeria
Tallys Yunes	Miami Herbert Business School, USA
Neng-Fa Zhou	Brooklyn College, USA

Additional Reviewers

Josep, Alos	Ion, Mandoiu
Blair, Archibald	Maxime, Mulamba Ke Tchomba
Noureddine, Aribi	Ali, Najafabadi
Gilles, Audemard	Saeed, Nejati
Behrouz, Babaki	Jesus, Ojeda
Federico, Baldo	Philippe, Olivier
Victor, Bucarey Lopez	Abdelkader, Ouali
Mikaël, Capelle	Alexandre, Papadopoulos

Mats, Carlsson	William, Pettersson
Violet, Chen	Steve, Prestwich
Nguyen, Dang	Aida, Rahmattalabi
Elisabetta, De Maria	Philippe, Refalo
Eghonghon-Aye, Eigbe	András Z., Salamon
Ozgun, Elci	Joe, Scott
Joan, Espasa Arxer	Aditya A., Shrotri
Julien, Ferry	Anil, Shukla
Arthur, Godet	Fabio, Tardivo
Ruth, Hoffmann	Eduard, Torres Montiel
Elizabeth, Hu	Cuong, Tran
Isaac, Huang	Mathieu, Vavrille
Christopher, Jefferson	Abdelrahman, Zayed
Håkan, Kjellerstrand	Allen, Zhong
James, Kotary	Heytem, Zitoun
Chunxiao, Li	

■ List of Authors

Saman Ahmadi  (11)

Department of Data Science and AI, Monash University, Victoria, Australia; CSIRO Data61, Canberra, Australia

Marie Anastacio  (43)


Leiden Institute of Advanced Computer Science, Leiden, The Netherlands

Carlos Ansótegui  (12)

Logic & Optimization Group (LOG), University of Lleida, Spain

Valentin Antuori (13, 14)


Renault, Plessis-Robinson, France; LAAS-CNRS, Université de Toulouse, CNRS, France

João Araújo  (4)

NOVA University Lisbon, Portugal

Claudia Archetti (1)

Department of Information Systems, Decision Sciences and Statistics, ESSEC Business School, Cergy, France

Blair Archibald  (15)

School of Computing Science, University of Glasgow, UK

Eddie Armstrong (16)

Johnson & Johnson Research Centre, Limerick, Ireland

Souheib Baarir (7)

Sorbonne Université, CNRS UMR 7606 LIP6, France; Université Paris Nanterre, France

Yiwei Bai (17)

Cornell University, Ithaca, NY, USA

Tomáš Balyo (6, 55)

CAS Software AG, Karlsruhe, Germany

J. Christopher Beck (35)

Department of Mechanical & Industrial Engineering, University of Toronto, Canada

Abderahmane Bedouhene (18)

LIGM, Ecole des Ponts ParisTech, Université Gustave Eiffel, CNRS, Marne-la-Vallée, France

Gleb Belov  (49)

Data Science & AI, Monash University, Clayton, Australia

Jeremias Berg  (28, 44, 51)

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

Vincent Botbol (59)

Nomadic labs, Paris, France

Kenneth N. Brown  (46)


Insight Centre for Data Analytics, School of Computer Science, University College Cork, Ireland

Víctor Bucarey  (42)

Institute of Engineering Sciences, Universidad de O'Higgins, Rancagua, Chile

Kyle Burns  (15)

School of Computing Science, University of Glasgow, UK

Shaowei Cai  (5, 39)

State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China; School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China

Rocsildes Canoy  (42)

Data Analytics Laboratory, Vrije Universiteit Brussel, Belgium

Quentin Cappart (33)

Ecole Polytechnique de Montréal, Canada

Yannick Carissan  (19)

Aix Marseille Univ, CNRS, Centrale Marseille, ISM2, Marseille, France

Di Chen (17)

Cornell University, Ithaca, NY, USA

Dingding Chen (41)

College of Computer Science, Chongqing University, China

Liqian Chen (57)

College of Computer, National University of Defense Technology, Changsha, China

Taoqing Chen (57)

State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, Changsha, China

Ziyu Chen (41)

College of Computer Science, Chongqing University, China

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel



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

- Mohamed Sami Cherif  (20)
Aix-Marseille Univ, Université de Toulon, CNRS,
LIS, Marseille, France
- Daniil Chivilikhin (47)
ITMO University, St. Petersburg, Russia
- Choiwah Chow  (4)
Universidade Aberta, Lisbon, Portugal
- Eldan Cohen (50)
Department of Mechanical and Industrial
Engineering, University of Toronto, Canada
- Jordi Coll (38)
Aix Marseille Univ, Université de Toulon, CNRS,
LIS, Marseille, France
- Martin C. Cooper  (21)
IRIT, Université de Toulouse III, France
- Ágnes Cseh  (22)
Hasso-Plattner-Institute, Universität Potsdam,
Germany; Institute of Economics, Centre for
Economic and Regional Studies, Pécs, Hungary
- Tobias Czauderna  (49)
Human-Centred Computing, Monash University,
Clayton, Australia
- Guilherme de Azevedo Silveira  (10)
Alura, São Paulo, Brazil
- Simon de Givry  (23)
Université Fédérale de Toulouse, ANITI,
INRAE, UR 875, France
- Maria Garcia de la Banda  (49)
Data Science & AI, Monash University, Clayton,
Australia
- Adrien Debesson (27)
Ubisoft, Bordeaux, Nouvelle-Aquitaine, France
- François Delobel (40)
LIMOS, CNRS UMR 6158, University Clermont
Auvergne, Aubière, France
- Matthieu Dien (59)
Université de Caen, France
- Tomáš Dlask  (23)
Faculty of Electrical Engineering, Czech
Technical University in Prague, Czech Republic
- Guillaume Escamocher  (22)
Insight Centre for Data Analytics, School of
Computer Science and Information Technology,
University College Cork, Ireland
- Siham Essodaigui (14)
Renault, Plessis-Robinson, France
- Guangsheng Fan (57)
State Key Laboratory of High Performance
Computing, College of Computer, National
University of Defense Technology, Changsha,
China
- Johannes K. Fichte  (24, 25)
TU Dresden, Germany
- Nathanaël Fijalkow  (43)
CNRS, LaBRI, Bordeaux, France,; The Alan
Turing Institute of data science, London, UK
- Ferdinando Fioretto  (2)
Syracuse University, NY, USA
- José Fragoso Santos  (31)
INESC-ID/IST, University of Lisbon, Portugal
- Cristian Galleguillos  (26)
Pontificia Universidad Católica de Valparaíso,
Chile; University of Bologna, Italy
- Junsong Gao (41)
College of Computer Science, Chongqing
University, China
- Michele Garraffa (16)
Confirm SFI Research Centre for Smart
Manufacturing, Limerick, Ireland; School of
Computer Science, University College Cork,
Ireland
- Martin Gebser (36)
Universität Klagenfurt, Austria; Technische
Universität Graz, Austria
- Begüm Genç  (22)
Insight Centre for Data Analytics, School of
Computer Science and Information Technology,
University College Cork, Ireland
- Ambros Gleixner (52)
Zuse Institute Berlin, Germany; HTW Berlin,
Germany
- Gaël Glorian (27)
LaBRI – CNRS UMR 5800, Université de
Bordeaux, Talence, Nouvelle-Aquitaine, France;
Ubisoft, Bordeaux, Nouvelle-Aquitaine, France
- Carla P. Gomes (17)
Cornell University, Ithaca, NY, USA
- Tias Guns (42)
Data Analytics Laboratory, Vrije Universiteit
Brussel, Belgium; Department of Computer
Science, KU Leuven, Belgium


- Djamal Habet (20, 38)
Aix-Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France
- Denis Hagebaum-Reignier  (19)
Aix Marseille Univ, CNRS, Centrale Marseille, ISM2, Marseille, France
- Daniel Harabor (11)
Department of Data Science and AI, Monash University, Victoria, Australia
- Kun He (38)
Huazhong University of Science and Technology, Wuhan, China
- Emmanuel Hebrard  (13, 14)
LAAS-CNRS, Université de Toulouse, CNRS, ANITI, France
- Markus Hecher  (24, 25)
TU Wien, Austria; Universität Potsdam, Germany
- Holger H. Hoos  (43)
Leiden Institute of Advanced Computer Science, Leiden, The Netherlands; University of British Columbia, Vancouver, Canada
- Marie-José Huguet (14)
LAAS-CNRS, Université de Toulouse, CNRS, INSA, France
- Alexey Ignatiev (47)
Monash University, Melbourne, Australia
- Hannes Ihalainen (28)
HIIT, Department of Computer Science, University of Helsinki, Finland
- Markus Iser  (6)
Karlsruhe Institute of Technology (KIT), Germany
- Nicolas Isoart (29, 30)
Université Côte d'Azur, Nice, France
- Ola Jabali (1)
Department of Electronics, Information and Bioengineering, Politecnico di Milano, Italy
- Mikoláš Janota  (4, 31)
Czech Technical University in Prague, Czech Republic
- Luc Jaulin (18)
Lab-STICC, ENSTA-Bretagne, Brest, France
- Peter Jonsson (32)
Department of Computer and Information Science, Linköping University, Sweden
- Chaitanya K. Joshi  (33)
Institute for Infocomm Research, A*STAR, Singapore
- Matti Järvisalo  (8, 28, 44, 51)
HIIT, Department of Computer Science, University of Helsinki, Finland
- Anissa Kheireddine  (7)
EPITA, LRDE, Kremlin-Bicêtre, France; Sorbonne Université, UMR 7606 LIP6, Paris, France
- Philip Kilby (11)
CSIRO Data61, Canberra, Australia
- Markus Kirchweger (34)
Algorithms and Complexity Group, TU Wien, Austria
- Zeynep Kiziltan  (26)
University of Bologna, Italy
- Matthias Klapperstueck  (49)
Human-Centred Computing, Monash University, Clayton, Australia
- Michael Klein (55)
CAS Software AG, Karlsruhe, Germany
- Wolfgang Kohlenbrein (36)
Kostwein Holding GmbH, Klagenfurt, Austria
- Tuukka Korhonen  (8)
HIIT, Department of Computer Science, University of Helsinki, Finland
- Anton Korikov (35)
Department of Mechanical & Industrial Engineering, University of Toronto, Canada
- Benjamin Kovács (36)
Universität Klagenfurt, Austria
- T. K. Satish Kumar (53)
Department of Computer Science, Department of Physics and Astronomy, Department of Industrial and Systems Engineering, Information Sciences Institute, University of Southern California, Los Angeles, CA, USA
- Marie-Louise Lackner (37)
Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, DBAI, TU Wien, Austria
- Pascal Lafourcade (40)
LIMOS, CNRS UMR 6158, University Clermont Auvergne, Aubière, France


Victor Lagerkvist (32)
Department of Computer and Information
Science, Linköping University, Sweden

Thomas Laurent (33)
Loyola Marymount University, LA, USA

Pierre Le Bodic  (48)
Data Science & AI, Monash University, Clayton,
Australia

Stéphane Le Menec (18)
MBDA, Le Plessis Robinson, France

Kevin Leo  (49)
Data Science & AI, Monash University, Clayton,
Australia

Bohan Li  (39)
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of
Sciences, Beijing, China; School of Computer
Science and Technology, University of Chinese
Academy of Sciences, Beijing, China

Chu-Min Li (38)
Huazhong University of Science and Technology,
Wuhan, China; Université de Picardie Jules
Verne, Amiens, France; Aix Marseille Univ,
Université de Toulon, CNRS, LIS, Marseille,
France

Hongbo Li  (9)
School of Information Science and Technology,
Northeast Normal University, Changchun, China

Zhanshan Li (9)
College of Computer Science and Technology,
Jilin University, Changchun, China


Luc Libralesso (40)
LIMOS, CNRS UMR 6158, University Clermont
Auvergne, Aubière, France

Defeng Liu (3)
CERC, Polytechnique Montréal, Canada

Xiangshuang Liu (41)
College of Computer Science, Chongqing
University, China

Andrea Lodi  (3)
Jacobs Technion-Cornell Institute, Cornell Tech
and Technion - Israel Institute of Technology,
New York, NY, USA; CERC, Polytechnique
Montréal, Canada

Chuan Luo  (5)
School of Software, Beihang University, Beijing,
China


Jayanta Mandi  (42)
Data Analytics Laboratory, Vrije Universiteit
Brussel, Belgium

Vasco Manquinho  (31)
INESC-ID/IST, University of Lisbon, Portugal

Felip Manyà (38)
Artificial Intelligence Research Institute, CSIC,
Bellaterra, Spain

João Marques-Silva  (21)
IRIT, CNRS, Toulouse, France


Théo Matricon  (43)
Univ. Bordeaux, CNRS, LaBRI, UMR 5800,
F-33400, Talence, France

Ciaran McCreesh  (15, 25)
School of Computing Science, University of
Glasgow, UK

Sheila McIlraith (50)
Department of Computer Science, University of
Toronto, Canada; Vector Institute, Toronto,
Canada

Kuldeep S. Meel (58)
School of Computing, National University of
Singapore, Singapore

Andrea Mor (1)
Department of Economics and Management,
University of Brescia, Italy


António Morgado  (31)
INESC-ID Lisbon, Portugal

Christoph Mrkvicka (37)
MCP GmbH, Wien, Austria





Nysret Musliu (37)
Christian Doppler Laboratory for Artificial
Intelligence and Optimization for Planning and
Scheduling, DBAI, TU Wien, Austria

Bertrand Neveu (18)
LIGM, Ecole des Ponts ParisTech, Université
Gustave Eiffel, CNRS, Marne-la-Vallée, France

Alain Nguyen (14)
Renault, Plessis-Robinson, France

Andreas Niskanen  (44)
HIIT, Department of Computer Science,
University of Helsinki, Finland

Barry O'Sullivan (16)
Confirm SFI Research Centre for Smart
Manufacturing, Limerick, Ireland; School of
Computer Science, University College Cork,
Ireland

- Jesús Ojeda  (12)
Logic & Optimization Group (LOG), University of Lleida, Spain
- Sebastian Ordyniak (32)
Algorithms Group, University of Sheffield, UK
- Ilya Otpuschennikov (47)
ISDCT SB RAS, Irkutsk, Russia
- Artem Pavlenko (47)
ITMO University, St. Petersburg, Russia;
JetBrains Research, St. Petersburg, Russia
- Xiao Peng (45)
CITI, INRIA, INSA Lyon, F-69621, Villeurbanne, France
- Omer Perry  (60)
Ben Gurion University of the Negev, Beer Sheva, Israel
- Sebastian Pokutta (52)
Zuse Institute Berlin, Germany; TU Berlin, Germany
- Tom Portoleau (13)
LAAS-CNRS, Université de Toulouse, CNRS, France
- Nicolas Prcovic (19)
Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France
- Charles Prud'homme (56)
TASC, IMT-Atlantique, LS2N-CNRS, F-44307 Nantes, France
- Luis Quesada  (22, 46)
Insight Centre for Data Analytics, School of Computer Science and Information Technology, University College Cork, Ireland
- Ben Rachmut  (60)
Ben Gurion University of the Negev, Beer Sheva, Israel
- Etienne Renault  (7)
EPITA, LRDE, Kremlin-Bicêtre, France
- Louis Rivière (13)
LAAS-CNRS, Université de Toulouse, CNRS, ANITI, France
- Valentin Roland (24)
TU Dresden, Germany
- Louis-Martin Rousseau (33)
Ecole Polytechnique de Montréal, Canada
- Jean-Charles Régim (29, 30)
Université Côte d'Azur, Nice, France
- Philipp Schrott-Kostwein (36)
Kostwein Holding GmbH, Klagenfurt, Austria
- Alexander Semenov (47)
ITMO University, St. Petersburg, Russia
- Ilankaikone Senthooan  (48, 49)
Data Science & AI, Monash University, Clayton, Australia
- Michele Sevegnani  (15)
School of Computing Science, University of Glasgow, UK
- Anas Shahab (25)
TU Dresden, Germany
- Pouya Shati (50)
Department of Computer Science, University of Toronto, Canada
- Laurent Simon  (27, 43)
LaBRI – CNRS UMR 5800, Université de Bordeaux, Talence, Nouvelle-Aquitaine, France
- Alberto Simonetto (1)
Multiprotexion srl, Gropello Cairoli, Italy
- Olivier Simonin (45)
CITI, INRIA, INSA Lyon, F-69621, Villeurbanne, France
- Helmut Simonis (16)
Confirm SFI Research Centre for Smart Manufacturing, Limerick, Ireland; School of Computer Science, University College Cork, Ireland
- Pavel Smirnov (51)
HIIT, Department of Computer Science, University of Helsinki, Finland
- Boro Sofranac (52)
Zuse Institute Berlin, Germany; TU Berlin, Germany
- Christine Solnon (40, 45)
INSA Lyon, CITI, INRIA CHROMA, F-69621 Villeurbanne, France
- Ricardo Soto  (26)
Pontificia Universidad Católica de Valparaíso, Chile
- M.Grazia Speranza (1)
Department of Economics and Management, University of Brescia, Italy
- Kostas Stergiou (54)
Dept. of Electrical & Computer Engineering, University of Western Macedonia, Kozani, Greece

Peter J. Stuckey  (48)

Data Science & AI, Monash University, Clayton, Australia

Shannon Sweitzer (53)

Department of Industrial and Systems Engineering, University of Southern California, Los Angeles, CA, USA

Stefan Szeider (34)


Algorithms and Complexity Group, TU Wien, Austria

Guido Tack (11)

Department of Data Science and AI, Monash University, Victoria, Australia

Pierre Tassel (36)

Universität Klagenfurt, Austria

Cyril Terrioux  (19, 20)

Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

Eduard Torres  (12)

Logic & Optimization Group (LOG), University of Lleida, Spain

Gilles Trombettoni (18)

LIRMM, Université de Montpellier, CNRS, France

Charlotte Truchet (56)

Laboratoire des Sciences du Numérique de Nantes, 44322 Nantes, France

Dimosthenis C. Tsouros (54)

Dept. of Electrical & Computer Engineering, University of Western Macedonia, Kozani, Greece

Nils Merlin Ullmann (55)

CAS Software AG, Karlsruhe, Germany

Vladimir Ulyantsev (47)

ITMO University, St. Petersburg, Russia

Adrien Varet (19)

Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

Mathieu Vavrille (56)

Laboratoire des Sciences du Numérique de Nantes, 44322 Nantes, France

Daniel Walkiewicz (37)

MCP GmbH, Wien, Austria

Mark Wallace  (49)

Data Science & AI, Monash University, Clayton, Australia

Ji Wang (57)

State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, Changsha, China

Kai Wang (39)

School of Computer Science and Information Technology, Northeast Normal University, Changchun, China

Tengbin Wang (57)

College of Computer, National University of Defense Technology, Changsha, China

Yiyuan Wang  (39)


School of Computer Science and Information Technology, Northeast Normal University, Changchun, China; Key Laboratory of Applied Statistics of MOE, Northeast Normal University, Chnagchun, China

Tomáš Werner  (23)

Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic

Felix Winter (37)

Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, DBAI, TU Wien, Austria

Michael Wybrow  (49)

Human-Centred Computing, Monash University, Clayton, Australia

Zhenxing Xu (38)

Huazhong University of Science and Technology, Wuhan, China

Jiong Yang (58)

School of Computing, National University of Singapore, Singapore

William Yeoh  (60)


Washington University in Saint Louis, MO, USA

Minghao Yin (9)

School of Information Science and Technology, Northeast Normal University, Changchun, China

Sylvain Yvon-Palot (27)

Ubisoft, Bordeaux, Nouvelle-Aquitaine, France

Jian Zhang  (5)

State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of
Sciences, Beijing, China; School of Computer
Science and Technology, University of Chinese
Academy of Sciences, Beijing, China

Xindi Zhang  (5)

State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of
Sciences, Beijing, China; School of Computer
Science and Technology, University of Chinese
Academy of Sciences, Beijing, China

Ghiles Ziat (59)

ISAE-SUPAERO, Université de Toulouse,
France

Roie Zivan  (60)

Ben Gurion University of the Negev, Beer Sheva,
Israel

The Bi-Objective Long-Haul Transportation Problem on a Road Network

Claudia Archetti ✉

Department of Information Systems, Decision Sciences and Statistics,
ESSEC Business School, Cergy, France

Ola Jabali ✉

Department of Electronics, Information and Bioengineering, Politecnico di Milano, Italy

Andrea Mor ✉

Department of Economics and Management, University of Brescia, Italy

Alberto Simonetto ✉

Multiprotexion srl, Gropello Cairoli, Italy

M.Grazia Speranza ✉

Department of Economics and Management, University of Brescia, Italy

Abstract

Long-haul truck transportation is concerned with freight transportation from shipments' origins to destinations, with vehicle trips lasting from some hours to several days. Drivers performing long-haul transportation are subject to strict rules derived from Hours of Service (HoS) regulations. There exists a large body of literature integrating HoS regulations within long-haul transportation. The optimization problems in this context generally deal with routing and scheduling decisions aimed at determining where a driver should stop and how long a rest should be. However, the overwhelming majority of the literature on long-haul transportation ignores refueling decisions and treats fuel costs as proportional to the traveled distance.

In this talk we analyze a long-haul truck scheduling problem where a path has to be determined for a vehicle traveling from a specified origin to a specified destination. We consider refueling decisions along the path while accounting for heterogeneous fuel prices in a road network. Furthermore, the path has to comply with Hours of Service (HOS) regulations. Therefore, a path is defined by the actual road trajectory traveled by the vehicle, as well as the locations where the vehicle stops due to refueling, compliance with HOS regulations, or a combination of the two. This setting is cast in a bi-objective optimization problem, considering the minimization of fuel cost and the minimization of path duration. An algorithm is proposed to solve the problem on a road network. The algorithm builds a set of non-dominated paths with respect to the two objectives. Given the enormous theoretical size of the road network, the algorithm follows an interactive path construction mechanism. Specifically, the algorithm dynamically interacts with a geographic information system to identify the relevant potential paths and stop locations. Computational tests are made on real-sized instances where the distance covered ranges from 500 to 1500 km. The algorithm is compared with solutions obtained from a policy mimicking the current practice of a logistics company. The results show that the non-dominated solutions produced by the algorithm significantly dominate the ones generated by the current practice, in terms of fuel costs, while achieving similar path durations. The average number of non-dominated paths is 2.7, which allows decision-makers to ultimately visually inspect the proposed alternatives.

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

Keywords and phrases Truck scheduling problem, hours of service regulations, fuel costs

Digital Object Identifier 10.4230/LIPIcs.CP.2021.1

Category Invited Talk

Related Version *Full Version:* <https://www.sciencedirect.com/science/article/pii/S0305048321001316>



© Claudia Archetti, Ola Jabali, Andrea Mor, Alberto Simonetto, and M.Grazia Speranza;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).



Editor: Laurent D. Michel; Article No. 1; pp. 1:1–1:1



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Constrained-Based Differential Privacy

Ferdinando Fioretto   

Syracuse University, NY, USA

Abstract

Data sets and statistics about groups of individuals are increasingly collected and released, feeding many optimization and learning algorithms. In many cases, the released data contain sensitive information whose privacy is strictly regulated. For example, in the U.S., the census data is regulated under Title 13, which requires that no individual be identified from any data released by the Census Bureau. In Europe, data release is regulated according to the General Data Protection Regulation, which addresses the control and transfer of personal data.

Differential privacy [1] has emerged as the de-facto standard to protect data privacy. In a nutshell, differentially private algorithms protect an individual's data by injecting random noise into the output of a computation that involves such data. While this process ensures privacy, it also impacts the quality of data analysis, and, when private data sets are used as inputs to complex machine learning or optimization tasks, they may produce results that are fundamentally different from those obtained on the original data and even rise unintended bias and fairness concerns.

In this talk, I will first focus on the challenge of releasing privacy-preserving data sets for complex data analysis tasks. I will introduce the notion of *Constrained-based Differential Privacy* (C-DP), which allows casting the data release problem to an optimization problem whose goal is to preserve the salient features of the original data. I will review several applications of C-DP in the context of very large hierarchical census data [3], data streams [2], energy systems [4], and in the design of federated data-sharing protocols. Next, I will discuss how errors induced by differential privacy algorithms may propagate within a decision problem causing biases and fairness issues [5, 6]. This is particularly important as privacy-preserving data is often used for critical decision processes, including the allocation of funds and benefits to states and jurisdictions, which ideally should be fair and unbiased. Finally, I will conclude with a roadmap to future work and some open questions.

2012 ACM Subject Classification Security and privacy → Privacy-preserving protocols; Computing methodologies → Artificial intelligence; Computing methodologies → Optimization algorithms

Keywords and phrases Optimization, Differential Privacy, Fairness

Digital Object Identifier 10.4230/LIPIcs.CP.2021.2

Category Invited Talk

Funding This research is partially supported by NSF grant 2133169. Its views and conclusions are those of the authors only.

References

- 1 Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pages 265–284, 2006.
- 2 Ferdinando Fioretto and Pascal Van Hentenryck. Optstream: Releasing time series privately. *Journal of Artificial Intelligence Research*, 65:423–456, 2019.
- 3 Ferdinando Fioretto, Pascal Van Hentenryck, and Keyu Zhu. Differential privacy of hierarchical census data: An optimization approach. *Artificial Intelligence*, pages 639–655, 2021.
- 4 Terrence W.K. Mak, Ferdinando Fioretto, Lyndon Shi, and Pascal Van Hentenryck. Privacy-preserving power system obfuscation: A bilevel optimization approach. *IEEE Transactions on Power Systems*, 35(2):1627–1637, March 2020. doi:10.1109/TPWRS.2019.2945069.
- 5 Cuong Tran, Ferdinando Fioretto, Pascal Van Hentenryck, and Zhiyan Yao. Decision making with differential privacy under the fairness lens. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, page (to appear), 2021.
- 6 Keyu Zhu, Pascal Van Hentenryck, and Ferdinando Fioretto. Bias and variance of post-processing in differential privacy. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 11177–11184, 2021.



© Ferdinando Fioretto;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 2; pp. 2:1–2:1

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Learning in Local Branching

Defeng Liu ✉

CERC, Polytechnique Montréal, Canada

Andrea Lodi¹ ✉ 🏠 

Jacobs Technion-Cornell Institute, Cornell Tech and Technion – Israel Institute of Technology,
New York, NY, USA

CERC, Polytechnique Montréal, Canada

Abstract

Although state-of-the-art solvers for Mixed-Integer Programming (MIP) experienced a dramatic performance improvement over the past decades, the resolution of some MIPs is still challenging, requiring hours of computations while, in practice, high-quality solutions are often required to be computed within a very restricted time frame. In such cases, it might be preferable to provide *anytime solutions*, i.e., a first reasonable solution should be generated as early as possible, then better ones produced in the subsequent computation with the user deciding where to stop.

In this respect, the *local branching* (LB) heuristic [2] was proposed to improve an incumbent solution either at very early stages of the computation within a general MIP framework or as a stand-alone algorithmic framework. Roughly speaking, given a feasible solution, the method iterates by first defining a solution neighborhood through the so-called *local branching cut*, then by exploring it by calling a black-box MIP solver. In the local branching algorithm, the choice of the neighborhood size is crucial to performance. In principle, it is desirable to have neighborhoods to be relatively small for efficient computation but still large enough to contain improving solutions. In [2], the size of the neighborhood is mostly initialized by a fixed constant value, then adjusted at run time. Nonetheless, it is reasonable to believe that there is no *a priori* single best neighborhood size and the choice of the value should depend on the characteristics of the problem. Furthermore, it is worth noting that, in many applications, instances of the same problem are solved repeatedly. Real-world problems have a rich structure: while more and more data points are collected, patterns and regularities appear. Therefore, problem-specific and task-specific knowledge can be learned from data and applied to adapting the corresponding optimization scenario. This motivates a broader paradigm of sizing the solution neighborhoods in local branching.

Following the line of work analyzed and surveyed in [1] on the use of Machine Learning (ML) for combinatorial optimization, in this work, we aim to guide the (local) search of the local branching heuristic by ML techniques. In particular, given a problem instance and a time limit for (heuristically) solving it, we exploit ML tools to predict reasonable good values of the neighborhood size, in order to maximize the performance of the local branching algorithm. We computationally show that the neighborhood size can indeed be learnt leading to improved performances and that the overall algorithm generalizes well both with respect to the instance size and, more surprisingly, across instances.

2012 ACM Subject Classification Computing methodologies → Machine learning; Mathematics of computing → Discrete optimization

Keywords and phrases Local search, learning, mixed-integer programming

Digital Object Identifier 10.4230/LIPIcs.CP.2021.3

Category Invited Talk

Acknowledgements The authors are indebted to Matteo Fischetti for several conversations about the work in progress.

¹ Corresponding Author and Invited Speaker



© Defeng Liu and Andrea Lodi;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 3; pp. 3:1–3:2



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

References

- 1 Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d’horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.
- 2 Matteo Fischetti and Andrea Lodi. Local branching. *Mathematical programming*, 98(1-3):23–47, 2003.

Filtering Isomorphic Models by Invariants

João Araújo 

NOVA University Lisbon, Portugal

Choiwah Chow  

Universidade Aberta, Lisbon, Portugal

Mikoláš Janota  

Czech Technical University in Prague, Czech Republic

Abstract

The enumeration of finite models of first order logic formulas is an indispensable tool in computational algebra. The task is hindered by the existence of isomorphic models, which are of no use to mathematicians and therefore are typically filtered out a posteriori. This paper proposes a divide-and-conquer approach to speed up and parallelize this process. We design a series of invariant properties that enable us to partition existing models into mutually non-isomorphic blocks, which are then tackled separately. The presented approach is integrated into the popular tool Mace4, where it shows tremendous speed-ups for a variety of algebraic structures.

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

Keywords and phrases finite model enumeration, isomorphism, invariants, Mace4

Digital Object Identifier 10.4230/LIPIcs.CP.2021.4

Category Short Paper

Funding *João Araújo*: Fundação para a Ciência e a Tecnologia, through the projects UIDB/00297-/2020 (CMA), PTDC/MAT-PUR/31174/2017, UIDB/04621/2020 and UIDP/04621/2020.

Mikoláš Janota: The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902. This scientific article is part of the RICAIP project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306.

1 Introduction

There are many types of relational algebras (groups, semigroups, quasigroups, fields, rings, MV-algebras, lattices, etc.) using operations and relations of many arities, but the overwhelming majority of the most popular only use operations of arity at most 2; in the words of two famous algebraists, *It is a curious fact that the algebras that have been most extensively studied in conventional (albeit modern!) algebra do not have fundamental operations of arity greater than two.* (See page 26 of [4])

To study and get intuition on them, mathematicians resort to libraries of all order n models of the algebra they are interested in (for small values of n). These libraries allow experiments such as testing and/or forming conjectures etc., to gain insights. Therefore, it comes as no surprise that GAP [8], the most popular computational algebra system, has many such libraries. For groups it has the list of almost all small groups up to order a few thousands and the list of all primitive groups up to degree a few thousands, among others; for semigroups it has the list of all small models up to order 8 [6]; for quasigroups up to order 6 [16]; there is also a library of Lie algebras and many others. These libraries are so important that the search for them has a long history in mathematics predating for many years the use of computers. For example, the search for libraries of degree n primitive groups



© João Araújo, Choiwah Chow, and Mikoláš Janota;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 4; pp. 4:1–4:9

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

started long ago: Jordan (1872) for $n \leq 7$; Burnside (1897) for $n \leq 8$; Manning (1906-1929) for $n \leq 15$; Sims (1970) for $n \leq 20$, Pogorelov (1980) for $n \leq 50$; Dixon and Mortimer (1988) for $n \leq 1000$. (See Appendix B of [7]; and for more recent results in OEIS [17]).

Many more such libraries are needed. For example, SMALLSEMI [6] has the list of semigroups up to order 8 (there are too many semigroups of order 9 to be storable), but if we impose extra properties on the semigroup (such as being inverse, a band, regular, or Clifford, etc. – there are tens of classes of semigroups –) their numbers decrease and hence libraries of models of higher orders could be produced and stored.

Many of these algebras can be defined in first order logic (FOL) and there are tools to allow mathematicians to encode their algebras and produce a meaningful library. The problem is that usually the tools that can be easily learned and used by mathematicians generate too many isomorphic models, thus wasting time generating redundant models and then wasting more time to get rid of them. For example, Mace4 [13], a very popular finite model enumerator among mathematicians due to its very intuitive and user-friendly language, would produce 28,947,734 inverse semigroups of order 8 when given the following simple first-order formulas as input [1] (with binary operation $*$ and unary operation $'$).

$$\begin{aligned} (x * y) * z &= x * (y * z). & (x * x') * x &= x. & 0 * 0 &= 0. \\ ((x * x') * y) * y' &= ((y * y') * x) * x'. & x'' &= x. \end{aligned}$$

During the search, the number of output models in this example is already greatly reduced by the Least Number Heuristic (LNH) and the special symmetry breaking input clause $0 * 0 = 0$. Out of the almost 29 millions output models, only 4,637 ($\approx 0.016\%$) are pairwise non-isomorphic. The proportion of non-isomorphic models in the outputs tends to get smaller very fast as the order of the algebraic structure goes higher.

Redundant models may either be eliminated during search or filtered out afterwards. Guaranteeing that search never produces isomorphic models is a hard problem and is rarely seen in modern solvers. This paper therefore tackles the second problem, i.e., the removal of redundant models from an already enumerated set.

In our context, the complexity of checking whether two models are isomorphic is only part of the problem. Another source of complexity is the large number of models that need to be checked. If all pairs of models are checked, the performance degrades rapidly as the total number of models increases (see Section 5).

To tackle this problem, we explored many different strategies eventually concluding that the best one is to assign to every generated model a vector that is invariant under isomorphism. This allows us to partition the output with all the isomorphic models living inside the same block (or part). This splits the problem into substantially smaller sub-problems. Moreover, processing inside each block can easily be done in parallel as models across blocks cannot be isomorphic. This is an important facet of the approach since modern-day computers are more often than not equipped with multiple cores.

What made this project take off was the identification of a large number of general algebra properties invariant under isomorphism coupled with experiments to identify a small subset of these properties without losing discriminating power. This approach will help mathematicians on two levels: first, it provides them with a tool on their desktop that quickly produces a library for the algebra they are working with; second, the tool may be run on a cluster of computers to pre-compute libraries for the most famous classes of algebras, and add them to GAP [8] or a similar system.

Our contributions to the area of isomorphic model elimination are (see Section 3):

- Devise an invariant-based algorithm that can be applied to algebras defined in FOL and containing at least one binary operation.
- Design a small set of invariant properties that in practice have high discriminating power, and yet are inexpensive to compute.
- Use a hash-map to store models partitioned by the invariant-based algorithm to allow fast storage and retrieval of models in the same block.

We apply the proposed partitioning technique to Mace4's isomorphic model filtering programs, and observe orders of magnitude speed-up in its isomorphic model elimination step (see Section 4).

2 Mathematical Background

Algebra is a pair (A, Ω) , where A is a set and Ω is a set of operations, that is, functions $f : A^n \rightarrow A$ (in this case f is said to be an operation of arity n). Let $A = (D, *_A)$ and $B = (D, *_B)$ be two algebras, each with one binary operation on a finite domain (or universe) D . An isomorphism of these two algebras is a bijective function $f : A \rightarrow B$ such that $f(a *_A b) = f(a) *_B f(b)$, for all $a, b \in A$. Two models are said to be isomorphic if there exists an isomorphism between them. The relation A is isomorphic to B is clearly an equivalence relation and hence induces a partition of the algebras considered. Only one representative algebra in each block is needed.

The definition of isomorphism can easily be extended to cover algebras with multiple binary operations. Formally, suppose A and B are algebras of type $(2^m, 1^n)$, where m, n are non-negative integers; then we can assume that the binary operations are $(*_1, \dots, *_m)$ and the unary operations are (g_1, \dots, g_n) . An isomorphism between them is a bijection $f : A \rightarrow B$ such that $f(a *_i b) = f(a) *_i f(b)$, for all $a, b \in D$ and every binary operation $*_i$, and for any unary operation g_i , we have $f(g_i(a)) = g_i(f(a))$, for all $a \in D$.

3 Invariant-based Algorithm

Let A and B be two algebras and $f : A \rightarrow B$ an isomorphism between them; in addition, suppose $e^2 = e \in A$ is an idempotent. Then $f(ee) = f(e)$ implies that $f(e)f(e) = f(e)$, that is, $f(e)$ under an isomorphism is also an idempotent. As isomorphisms map idempotents onto idempotents, it follows that the number of idempotents in A must be smaller or equal to the number of idempotents on B . Since the inverse of an isomorphism is an isomorphism, A and B must have the same number of idempotents. We call these properties that are preserved by isomorphisms (such as the *number of idempotents*) *invariant properties* or *invariants* for short. These invariant properties are the basis of our proposed algorithm.

Guided by fundamental concepts heavily appearing in different parts of mathematics, we design 10 invariant properties that collectively have high discriminating powers, and yet are inexpensive to compute. For a binary operation in a model with finite domain D , we compute the invariant properties for each domain element x as:

1. The smallest integer n such that $x^n = x^k$, $n > k \geq 1$ where we define x^n to be $(\dots (x * x) * x) \dots$ for n x 's (*periodicity*).
2. The number of $y \in D$ such that $x = (xy)x$ (*number of inverses*).
3. The number of distinct xy for all $y \in D$ (*size of right ideal*).
4. The number of distinct yx for all $y \in D$ (*size of left ideal*).
5. 1 if $xx = x$, 0 otherwise (*idempotency*).

6. The number of $y \in D$ such that $x(yy) = (yy)x$ (*number of commuting squares*).
7. The number of $y \in D$ such that $x = yy$ (*number of square roots*).
8. The number of $y \in D$ such that $x(xy) = (xx)y$ (*number of square associatizers*).
9. The number of pairs of $y, z \in D$ such that $zy = yz = x$ (*number of symmetries*).
10. The number of $y \in D$ such that there exists pairs of $s, t \in D$ where $x = st$ and $y = ts$ (*number of conjugates*).

Invariant 5 is the idempotent property of the domain element and is preserved by isomorphisms as discussed before. The correctness of invariants in general hinges on the following lemma (folklore). Let F be a FOL formula on the signature of the algebra and M and M' two isomorphic models. It holds that the sets S and S' defined by F in M and M' , respectively, are of the same cardinality. This is because the isomorphism induces a bijection between the two sets (*cf.* Theorem 1.1.10 in [12]). In other words, invariants based on solution counting are guaranteed to be correct.

We call the ordered list of invariant properties so calculated the invariant vector of that domain element. Each model with n domain elements will be associated with n invariant vectors. Isomorphic models must have the same set of invariant vectors.

To facilitate comparisons of invariant vectors, we sort the invariant vectors by the lexicographical order of their elements (see the example below for more explanations). It follows that models isomorphic to each other must have the same sorted invariant vectors. If the model has multiple binary operations, then invariant vectors are calculated for each of the binary operations, and all the invariant vectors of the same domain element are concatenated to form a combo invariant vector for that domain element. The combo invariant vectors will then be sorted to yield the final ordered list of invariants.

Often we are not only to compare 2 models for isomorphism, but to extract all non-isomorphic models from a list of models. In that case, we set up a hash map to store the blocks of the models. We use the invariant vectors for each model to send the model quickly to the block (in the hash map) to which it belongs. That is, the keys in this hash map are the invariant vectors, and the values are the blocks of the models. After all models are hashed into the hash map, the blocks stored in the hash map can be processed separately, and possibly in parallel, to extract one representative model from each isomorphism class.

Note that our invariant-based algorithm does not compare models for isomorphism. It only cuts down the size of the problem to improve the speed of existing isomorphism filters such as Mace4's *isofilter*.

As an example to show how invariant vectors are constructed and used, suppose we want to find all non-isomorphic models in a list of 3 quasigroups, A , B , and C , of order 4. Suppose further that their domain is $D = \{0, 1, 2, 3\}$ and their operation tables are given in Table 1.

■ **Table 1** Operation tables of Quasigroups A , B and C .

Model A					Model B					Model C				
*A	0	1	2	3	*B	0	1	2	3	*C	0	1	2	3
0	0	1	2	3	0	0	1	2	3	0	0	1	2	3
1	1	0	3	2	1	1	2	3	0	1	1	0	3	2
2	2	3	1	0	2	2	3	0	1	2	2	3	0	1
3	3	2	0	1	3	3	0	1	2	3	3	2	1	0

The 10 invariant properties can easily be calculated for each of the domain elements of these models. Note that while the invariant vector for each domain element is calculated separately, it is not important exactly which domain element gives a particular invariant vector. It is the set of invariant vectors as a whole that matters.

Invariant vectors of Model A Invariant vectors of Model B Invariant vectors of Model C

0: 2 1 4 4 1 4 2 4 4 1	0: 2 1 4 4 1 4 2 4 4 1	0: 2 1 4 4 1 4 4 4 4 1
1: 3 1 4 4 0 4 2 4 4 1	2: 3 1 4 4 0 4 2 4 4 1	1: 3 1 4 4 0 4 0 4 4 1
2: 5 1 4 4 0 4 0 4 4 1	1: 5 1 4 4 0 4 0 4 4 1	2: 3 1 4 4 0 4 0 4 4 1
3: 5 1 4 4 0 4 0 4 4 1	3: 5 1 4 4 0 4 0 4 4 1	3: 3 1 4 4 0 4 0 4 4 1

■ **Figure 1** Lexicographically sorted invariant vectors with discerning properties highlighted.

Next we sort the invariant vectors of each model by their elements lexicographically. Invariant vectors of models *A* and *C* need no change as they are already in the desired sort order. Invariant vectors of model *B* will be in sort order by interchanging the invariant vectors of elements 1 and 2, which are the second and third row. The final invariant vectors are shown in the Figure 1. Note that the first column in the tables is the domain element, and the next 10 columns are its invariant properties.

The highlighted numbers in the figure are the discerning invariant properties in the example. All other invariant properties are the same from domain element to domain element. For example, Invariant properties 3, *size of right ideal*, and 4, *size of left ideal*, always equal to the size of the domain *D* because the operation table of a quasigroup is a Latin square. This highlights the need for multiple invariant properties targeting different areas of algebraic structures to increase their collective discriminating powers. In fact, our algorithm depends more on the orthogonality of the invariants than on the splitting power of any one individual invariant. See Table 2 for the top invariants in different algebras.

It should be easy to see that models *A* and *B* have the same sorted invariant vectors, and thus are possibly isomorphic to each other. They are indeed isomorphic to each other because applying the permutation (1, 2) to model *B* will give model *A*. However, invariant vectors alone cannot prove that they are isomorphic models. It is also easy to see that the invariant vectors of model *C* are different from those of the other 2 models, and from this fact alone, we can conclude that model *C* is not isomorphic to any of *A* and *B*.

Finally, for ease of comparison and hashing, we concatenate the sorted invariant vectors into a single string. The string representation of the invariant vectors for the models are:

$$\begin{aligned}
 A, B: & 2,1,4,4,1,4,2,4,4,1,3,1,4,4,0,4,2,4,4,1,5,1,4,4,0,4,0,4,4,1,5,1,4,4,0,4,0,4,4,1 \\
 C: & 2,1,4,4,1,4,4,4,4,1,3,1,4,4,0,4,0,4,4,1,3,1,4,4,0,4,0,4,4,1,3,1,4,4,0,4,0,4,4,1
 \end{aligned}$$

Since we are to extract all non-isomorphic models from this list of models, we use the string representations of the invariant vectors as the keys for the hash map. Both models *A* and *B* will therefore go to the same block in the hash map, but *C* will go to a different block. Now that all 3 models are deposited in their blocks in the hash map, each block can be processed separately in parallel as we only need to compare models in the same block for isomorphism. This step can be performed by many existing programs such as Mace4's isomorphism filters (see Section 4).

Finally, if the models have multiple binary operations, we compute the unsorted invariant vectors for each binary operation as described above, then concatenate the invariant vectors of the same domain element into one combo invariant vector, sort these combo invariant vectors in lexicographical order, and finally concatenate the sorted invariant vectors into their string format.

It is important to note that the hash map in our algorithm obviates the need to compare invariant vectors among the models during the partitioning process. If we do pairwise comparison of models by their invariant vectors in any step, we would end up with a $O(n^2)$ worst-case scenario.

4 Experimental Results

We have implemented an invariant-based pre-processor to the Mace4's isomorphic models filters. We run the experiments on a 6-core Intel®Core™ i7-9850H CPU computer. We shall show results of tests on 3 algebraic structures, namely, quasigroup, inverse semigroup, and quandle [1]. They are chosen because of their importance in the mathematical world. Quasigroup is the most prominent non-associative algebra, inverse semigroup is probably the most studied associative algebra with a unary operation, and quandles is probably most important algebra with 2 binary operations.

The results show that when the size of the output models is more than just a few hundreds of thousands, the invariant vectors often give an order or two magnitudes of improvements in the speed of the isomorphism elimination process even without running them in parallel. A very desirable feature of our algorithm is that the improvement increases dramatically as the size of the problem grows. Furthermore, Mace4's *isofilter2* is not able to handle input size beyond a few million quasigroups of order 6 (see Table 2), but our invariant-based algorithm can partition the models into smaller blocks of sizes within Mace4's limits.

■ **Table 2** Isomorphism Eliminations.

	Order	# of Mace4 Outputs	Time (s)	
			With Invariants	Without Invariants
Quasigroups	5	10,944	1	1
	6	11,543,040	1,182	N/A
Inverse Semigroups	5	2,151	<1	<1
	6	38,828	3	2
	7	929,923	73	81
	8	28,947,734	2,873	150,703
Quandles	6	1,833	2	1
	7	22,104	6	374
	8	359,859	450	267,463

We show the results of the non-parallel runs to demonstrate the improvements due solely to the invariant vectors. The performance can be improved further if the blocks are processed in parallel. For example, the processing time for the biggest block for quandles of order 8 is only 20 seconds, so if we have enough processors to process all the blocks in parallel, then the processing time can theoretically be cut down close to $24 + 19.937 \approx 44$ seconds from 450 seconds, more than 90% reduction (see Table 3).

■ **Table 3** Isomorphism Eliminations in Parallel.

	Order	#Blocks	Time (s)	
			Generating Invariants	Processing Biggest Block
Quasigroups	6	1,129,129	265	0.0106132
Inverse Semigroups	8	4,582	1,031	2.807
Quandles	8	1,143	24	19.937

One reason for the dramatic improvement in the run-time by our invariant-based algorithm is that the invariant vectors chosen have great discriminating power as shown by the fact that the average number of non-isomorphic models per block is very close to 1 (see Table 4). The top 4 contributing invariants for the highest order of each class are also listed in Table 4.

■ **Table 4** Discriminating Power of Invariant Vectors.

	Order	#Blocks	Non-isomorphic Models		Top 4 Invariants
			Total	Avg per Block	
Quasigroups	5	1,402	1,411	1.01	6, 1, 8, 10
	6	1,129,129	1,130,531	1.00	
Inverse Semigroups	5	52	52	1.00	9, 3, 2, 1
	6	208	208	1.00	
	7	908	911	1.02	
	8	4,582	4,637	1.01	
Quandles	6	66	73	1.11	8, 3, 6, 10
	7	250	298	1.19	
	8	1,143	1,581	1.38	

5 Related Work

The proposed approach falls into the class of *divide-and-conquer* algorithms; most notably Heule et al. [10] recently applied the *cube-and-conquer* approach [9] to solve the Boolean Pythagorean triples problem.

There are a large number of techniques to break symmetries during the search phase [5], such as the Least Number Heuristic (LNH) [18] and the eXtended LNH (XLNH) [2]. The LNH, for example, is a very popular dynamic symmetry breaker implemented in Mace4, FALCON [18], and SEM [19], etc., to help reduce the number of isomorphic models. However, these techniques do not guarantee isomorph-freeness. Systems that try to generate isomorph-free models, such as SEMK [3, 14] and SEMD [11], are either yet to be complete, or are better off allowing some isomorphic models in the outputs for some problem sets. Thus, post-processing tools such as our invariant-based algorithm have an important role in isomorphism elimination as total elimination of isomorphism in the model search phase may not always be the best option.

Invariants are widely used under different guises in many branches of mathematics. For example, in graph theory, node invariants can be used to help detect isomorphic graphs [15]. Interestingly, similar ideas can be seen in Mace4's isomorphism filters. Indeed, Mace4's *isofilter* uses the numbers of occurrences of domain elements in the operation tables as the lone invariant that serves 2 purposes: First is to do quick checks for non-isomorphism, as models having different occurrences of domain elements cannot be isomorphic. Second is to guide the construction of isomorphic functions between potential isomorphic models, as domain elements can only map to domain elements having the same occurrences in the operation tables. This reduces the number of permutations to try in the search of isomorphic functions. However, the lone invariant in *isofilter* would fail miserably if the models are quasigroups for which each domain element would appear the same of times in the operation table. To mitigate this problem, Mace4 provides another isomorphism filter, *isofilter2*, which

transforms the models to their canonical forms based on the same algorithm [14] given by McKay as mentioned above in SEMK. Compared to *isofilter*, *isofilter2* performs much better for quasigroups, but worse on other algebraic structures such as semigroups due to its high overheads in computing canonical forms. Nevertheless, both filters compare every model against the list of non-isomorphic models found so far, and hence their performances degrade rapidly as the number of models increases. Therefore, both filters benefit immensely from the reduced number of models in the blocks created by our invariant-based algorithm.

The *loops* package [16] in GAP [8] uses invariant vectors of 9 invariants in many of its isomorphism-related functions. Like Mace4's *isofilter*, it uses invariant vectors to check for non-isomorphism, and to help guide the construction of isomorphic function between models using sophisticated algorithms that take advantage of other GAP functions. Their invariant vectors work on only one operation table, and exploit heavily specific properties of quasigroups and loops, which may be ineffective in other kinds of algebras. Our invariant-based algorithm targets different aspects of all algebraic structures including quasigroups, semigroups, and more. It also works with multiple binary operations, and does not rely on any built-in functionality of GAP. Moreover, given a list of models to find non-isomorphic models, the *loops* package would compare the invariant vector of every model against those of the list of all non-isomorphic models found so far to get the list of potential isomorphic models. Our hash map-based organization of models eliminates the need to compare invariant vectors repeatedly because all models having the same invariant vectors are already grouped into the same block in the hash map.

6 Future Work and Conclusions

Currently, we only compute invariants based on binary operations, which are by far the most prevalent operations in algebraic structures [4]. However, unary operations are also quite common, and may be even less expensive to manipulate. The discriminating power of the invariant vectors of the model can be enhanced with the addition of invariant vectors based on unary operations, and will be part of our future focus.

The results of our research open a whole new line of research into using invariant properties to eliminate isomorphism in finite model enumeration:


- Identify more invariant properties and the cases for which each of them may be useful.
- Allow dynamic, and preferably automatic, selection of invariant properties to use in any given algebraic structure because different invariants work best for different algebraic structures (see example in Section 3, and also Table 4), so we need to allow dynamic, and preferably automatic, selection of invariant properties.
- Find the best sets of invariant properties to use for various sizes and types of models. A larger set of algebras (usually of higher orders) may need more invariants in the invariant vectors to provide enough discriminating power to separate the models into smaller blocks, but a smaller set of algebras may incur too much overhead in computing the invariant vectors with many invariant properties.

We observe that the invariant-based algorithm is efficient, scalable, and parallelizable. It is also compatible with most, if not all, existing finite model enumerators. The focus of future research will be on finding more good invariant properties, in binary and in unary operations, to be used as partitioning keys, and on adding the capability of dynamic and automatic selection of invariant properties to use.

References

- 1 João Araújo, David Matos, and João Ramires. Axiomatic library finder. URL: <https://axiomaticlibraryfinder.pythonanywhere.com/definitions> [cited 15.05.2021].
- 2 Gilles Audemard and Laurent Henocque. The extended least number heuristic. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning*, pages 427–442, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
- 3 Thierry Boy de la Tour and Prakash Countcham. An isomorph-free sem-like enumeration of models. *Electronic Notes in Theoretical Computer Science*, 125(2):91–113, 2005. Proceedings of the 5th International Workshop on Strategies in Automated Deduction (Strategies 2004). doi:10.1016/j.entcs.2005.01.003.
- 4 Stanley Burris and Hanamantagouda P. Sankappanavar. *A course in universal algebra*, volume 78 of *Graduate texts in mathematics*. Springer, 1981.
- 5 James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 148–159. Morgan Kaufmann, 1996.
- 6 A. Distler and J. Mitchell. Smallsemi, a library of small semigroups in GAP, Version 0.6.12, 2019. GAP package. URL: <https://gap-packages.github.io/smallsemi/>.
- 7 John D. Dixon and Brian Mortimer. *Permutation Groups*. Springer, 1996.
- 8 The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.11.1*, 2021. URL: <https://www.gap-system.org>.
- 9 Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC, Revised Selected Papers*, volume 7261, pages 50–65. Springer, 2011. doi:10.1007/978-3-642-34188-5_8.
- 10 Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the BooleanPythagorean triples problem via cube-and-conquer. In *Theory and Applications of Satisfiability Testing (SAT)*, 2016. doi:10.1007/978-3-319-40970-2_15.
- 11 Xiangxue Jia and Jian Zhang. A powerful technique to eliminate isomorphism in finite model search. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, pages 318–331, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- 12 David Marker. *Model Theory: An Introduction*. Springer, 2002.
- 13 William McCune. Mace4 reference manual and guide. Technical Report Technical Memorandum No. 264, Argonne National Laboratory, Argonne, IL, August 2003. URL: <https://www.cs.unm.edu/~mccune/prover9/mace4.pdf>.
- 14 Brendan D McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26(2):306–324, 1998. doi:10.1006/jagm.1997.0898.
- 15 Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014. doi:10.1016/j.jsc.2013.09.003.
- 16 Gábor Nagy and Petr Vojtěchovský. LOOPS, computing with quasigroups and loops in GAP, Version 3.4.1, November 2018. Refereed GAP package. URL: <https://gap-packages.github.io/loops/>.
- 17 Neil J. A. Sloane and The OEIS Foundation Inc. The on-line encyclopedia of integer sequences, 2020. URL: <http://oeis.org/?language=english>.
- 18 Jian Zhang. Constructing finite algebras with FALCON. *Journal of Automated Reasoning*, 17:1–22, August 1996. doi:10.1007/BF00247667.
- 19 Jian Zhang and Hantao Zhang. SEM: a system for enumerating models. In *IJCAI*, pages 298–303, 1995. URL: <http://ijcai.org/Proceedings/95-1/Papers/039.pdf>.

Improving Local Search for Structured SAT Formulas via Unit Propagation Based Construct and Cut Initialization

Shaowei Cai ✉ 

State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology,
University of Chinese Academy of Sciences, Beijing, China

Chuan Luo ✉ 

School of Software, Beihang University, Beijing, China

Xindi Zhang ✉ 

State Key Laboratory of Computer Science, Institute of Software,
Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology,
University of Chinese Academy of Sciences, Beijing, China

Jian Zhang ✉ 

State Key Laboratory of Computer Science, Institute of Software,
Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology,
University of Chinese Academy of Sciences, Beijing, China

Abstract

This work is dedicated to improving local search solvers for the Boolean satisfiability (SAT) problem on structured instances. We propose a construct-and-cut (CnC) algorithm based on unit propagation, which is used to produce initial assignments for local search. We integrate our CnC initialization procedure within several state-of-the-art local search SAT solvers, and obtain the improved solvers. Experiments are carried out with a benchmark encoded from a spectrum repacking project as well as benchmarks encoded from two important mathematical problems namely Boolean Pythagorean Triple and Schur Number Five. The experiments show that the CnC initialization improves the local search solvers, leading to better performance than state-of-the-art SAT solvers based on Conflict Driven Clause Learning (CDCL) solvers.

2012 ACM Subject Classification Theory of computation → Randomized local search

Keywords and phrases Satisfiability, Local Search, Unit Propagation, Mathematical Problems

Digital Object Identifier 10.4230/LIPIcs.CP.2021.5

Category Short Paper

Supplementary Material *Software (Source Code)*: <https://github.com/caiswgroup/CnC-LS>
archived at `swb:1:dir:f7ef44ee596e5f008dea01ef7e3c1ee47c8b93dc`

Funding This work was supported by Beijing Academy of Artificial Intelligence (BAAI), and Youth Innovation Promotion Association, Chinese Academy of Sciences (No. 2017150), as well as the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (Grant No. QYZDJ-SSW-JSC036).



© Shaowei Cai, Chuan Luo, Xindi Zhang, and Jian Zhang;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 5; pp. 5:1–5:10



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Given a Boolean formula, the Boolean Satisfiability problem (SAT) determines whether the variables of the formula can be assigned in such a way as to make the formula evaluate to TRUE. In the SAT problem, Boolean formulas are usually presented in Conjunctive Normal Form (CNF), i.e., $F = \bigwedge_i \bigvee_j \ell_{ij}$. SAT is the first NP-complete problem. Besides, SAT solvers have shown great success in many applications [29], including bounded model checking [10], program verification [11], and mathematical theorem proving [17].

Two popular methods for SAT are conflict driven clause learning (CDCL) [33] and local search. The CDCL based solvers evolve from the DPLL backtracking procedure [13] and combine reasoning techniques. The reasoning techniques in CDCL solvers, particularly unit propagation (UP) and clause learning, play a critical role in the good performance of CDCL solvers on application instances. Local search is an incomplete method and its process can be viewed as a random walk in the search space [19, 27]. Local search SAT solvers begin with an initial complete assignment and iteratively modify the assignment, until a model is found or a resource limit (usually the time limit) is reached [19, 28, 26]. Local search solvers are usually much simpler and lighter than CDCL ones. Indeed, they are probably the most lightweight SAT solvers. Local search has proved very effective for solving many NP-hard combinatorial problems. However, it is known that local search solvers are not effective as CDCL solvers on solving structured SAT instances, particularly those from real-world applications.

This work aims to improve local search solvers for structured SAT instances. Specifically, we propose a construct-and-cut (CnC) method for generating initial assignments for local search, which aims to produce diverse complete assignments as *consistent* as possible. The CnC method iteratively performs assigning procedures, which are also called construction tries, based on unit propagation and heuristics. In each construction try, the algorithm starts from an empty assignment and extends it to a complete assignment. Also, the algorithm records the best solution found (with fewest empty clauses) so far and its number of empty clauses which serves as an upper bound. In the subsequent tries, once the number of empty clauses reaches the upper bound, the try is cut off.

We use this CnC method to improve three state-of-the-art local search SAT solvers, by replacing the original initialization method with the CnC method. We conduct experiments with three important benchmarks, one of which arises from a recent real-world project about spectrum repacking [32] and the others consist of instances encoded from two important mathematical problems namely Boolean Pythagorean Triple [17] and Schur Number Five [16]. Experiment results show that, the CnC method brings obvious improvements to the local search solvers. Particularly, one of the CnC-enhanced local search solver outperforms modern SAT solvers based on CDCL approach on the three benchmarks.

2 Technical Background

2.1 Preliminary Definitions and Notations

Given a set of Boolean *variables* $\{x_1, x_2, \dots, x_n\}$, a *literal* is either a variable x_i or its negation \bar{x}_i . A conjunctive normal form (CNF) formula F is a conjunction of clauses (i.e., $F = C_1 \wedge C_2 \wedge \dots \wedge C_m$), where a *clause* is a disjunction of literals (i.e., $C_i = \ell_{i1} \vee \ell_{i2} \vee \dots \vee \ell_{ij}$). Alternatively, a CNF formula can be viewed as a set of clauses, and a clause can be viewed as a set of literals. For a formula F , we denote the set of variables in F by $Var(F)$, and the number of literals whose corresponding variable is x_i is denoted by $\Delta_F(x_i)$.

For a literal ℓ , its corresponding variable is denoted by $\ell.var$, and its *phase*, denoted by $\ell.phase$, is 1 if ℓ is positive and 0 if ℓ is negative. A literal can be viewed as an ordered pair of a variable and its phase, i.e., $\ell = (\ell.var, \ell.phase)$. For a literal ℓ , we denote by $\bar{\ell}$ the literal of opposite phase. A clause containing only one literal is a *unit clause*. We denote $\ell \in C_i$ if ℓ is a literal in clause C_i .

For a formula F , an *assignment* α is a mapping $Var(F) \rightarrow \{0, 1\}$. If α maps all variables to a Boolean value, we say it is a *complete* assignment. For a variable $x_i \in Var(F)$ and an assignment α , $\alpha[x_i]$ is the value of variable x_i under α . Given an assignment α , we say that a literal ℓ is true if $\alpha[\ell.var]$ is equal to $\ell.phase$. A clause is *satisfied* if it has at least one true literal, and *unsatisfied* if all the literals in the clause are false literals. By convention the empty clause \square is always unsatisfiable, and represents a conflict. SAT is the problem of deciding whether a given CNF formula is satisfiable.

The process of conditioning a CNF formula F on a literal ℓ amounts to replacing every occurrence of literal ℓ by the constant true, replacing $\bar{\ell}$ by the constant false, and simplifying accordingly. The result of conditioning F on ℓ is denoted by $F|_\ell$ and can be described succinctly as follows: $F|_\ell = \{c/\{\bar{\ell}\} | c \in F, \ell \notin c\}$. Note that $F|_\ell$ does not contain any literal ℓ or $\bar{\ell}$. When we assign a variable x with a value v , we can simplify the formula accordingly, and the simplified formula is denoted as $F|_{(x,v)}$.

Unit propagation on a CNF formula ϕ works as follows: First, we collect all unit clauses in ϕ , and then assume that variables are set to satisfy these unit clauses. If the unit clause $\{x_i\}$ appears in the formula, we set x_i to true. Also, if the unit clause $\{\bar{x}_i\}$ appears in the formula, we set x_i to false. We then condition the formula on these settings. The iterative application of this rule until no more unit clause remains is called *unit propagation* (UP).

2.2 Local Search for SAT

When solving a SAT formula by local search, the search space is organized as a network, in which each position represents a complete assignment and two positions are adjacent if they are neighbors. A commonly used neighborhood relation N maps assignments to their set of Hamming neighbors, i.e., assignments that differ in exactly one variable. Typically, a local search algorithm for SAT starts from a complete assignment, and flips a variable iteratively to search for a satisfying assignment. In this work, we focus on improving local search for SAT by generating good initial assignments.

3 Related Works and Discussions

This work utilizes a construct-and-cut method based on unit propagation (UP) to produce a good quality initial assignment for local search SAT solvers. Unit propagation is a simple form of reasoning, and has been used to improve local search solver previously.

Some local search solvers use UP to simplify the formula before the search [21, 8]. More complicated preprocessors have also been developed [31]. These preprocessing techniques are used to simplify the formula. If the formula cannot be simplified, they just do nothing.

Some algorithms use UP during local search. *UnitWalk* [18] prefers to perform UP if possible in each local search step, and only when UP is not applicable a normal local search step is executed. *QingTing* [22] is an improved version of *UnitWalk* with more efficient implementation and also switches between *UnitWalk* and a normal local search algorithm. *EagleUP* [15] also exploits UP during local search, where UP is performed only when the algorithm is stuck in local optima.

Although UP has been previously combined with local search, these previous works either use UP only as preprocessor, or use UP too heavily. These solvers usually improve local search on crafted and random instances, but no good result is reported on solving instances from real-world applications. Most previous local search solvers, including *CCAnr* [8], *Sattime* [21] and *ProbSAT* [4], generate the initial assignment randomly, while a recent local search solver *YalSAT* [5] also utilizes information such as the best found assignment in the last round to produce the initial assignment. On the other hand, UP-based initialization has been used in local search for MaxSAT [7, 9, 25]. However, in these works, the initialization does not use pruning techniques.

Another relevant direction is using CDCL to boost local search solvers. An incomplete hybrid solver *hybridGM* [3] calls CDCL search around local minima with only one unsatisfied clause. *SATHYS* [1] performs local search and calls a CDCL solver when it is stuck in local optima. However, these methods do not show improvement over the CDCL solvers on application benchmarks, although they show better performance than local search on crafted instances and better performance than CDCL solvers on random instances.

4 A Novel Initialization Method for Local Search SAT Solvers

This section presents the construct-and-cut (CnC) method, which can be used to produce good quality assignments for local search SAT solvers. The CnC algorithm consists of individual construction procedures, each of which constructs a complete assignment by assigning variables one by one.

4.1 The Construct-and-Cut Method

Before presenting the details of the CnC algorithm, we first introduce the key data structures used in the algorithm.

Set \mathcal{U} : it stores all unit clauses, noting that a unit clause has only one literal. \mathcal{U} is updated during the search. Newly generated unit clauses are put into \mathcal{U} , and a unit clause is removed from \mathcal{U} after it is picked to perform unit propagation.

Vector *value*: this vector records the assigned value for each variable. For each variable x , $value(x)$ has 4 possible values $\{-2, -1, 0, 1\}$, as explained below:

- $value(x) = -2$ means unit clauses x and \bar{x} appear in F simultaneously (may be due to different UP operations).
- $value(x) = -1$ means x is unassigned.
- $value(x) = 0$ means x is assigned the value 0 (false).
- $value(x) = 1$ means x is assigned the value 1 (true).

The CnC method is depicted in Algorithm 1. The algorithm consists of individual construction procedures (also called tries), and the number of tries to be executed is controlled by a parameter *cnc_times*. We use $\#(\square)$ to denote the number of empty clauses in the formula that the CnC algorithm is currently dealing with, which is the cost of the current assignment. The cost of the best assignment found (e.g. the minimum cost) in previous construction procedures is denoted as $cost^*$. In the beginning, CnC initializes $cost^*$ as the number of clauses in the input formula, and stores all unit clauses (if any) in \mathcal{U} .

In each try, the algorithm works on a copy of the input formula ϕ , which is denoted as F . In the beginning of each try, $value(x)$ is initialized as -1 for each variable (line 5), indicating that all variables are unassigned. Then, a loop is executed until there is no unassigned variable; moreover, the loop is terminated if $\#(\square)$ reaches $cost^*$.

Algorithm 1 $\text{CnC}(\phi, \text{cnc_times})$.

Input: A CNF formula ϕ , cnc_times
Output: An assignment α^* of variables in ϕ

```

1  $\text{cost}^* \leftarrow +$  the number of clauses in  $F$ ;
2 for  $i \leftarrow 1$  to  $\text{cnc\_times}$  do
3    $F \leftarrow \phi$ ;
4    $\mathcal{U} \leftarrow \{\text{all unit clauses in } \phi\}$ ;
5    $\forall x \in \text{Var}(F), \text{value}(x) \leftarrow -1$ ;
6   while  $\exists$  unassigned variables do
7     if  $\mathcal{U} \neq \emptyset$  then
8        $\ell \leftarrow \text{GetUL}(\mathcal{U})$ ;
9        $x \leftarrow \ell.\text{var}$ ;
10      if  $\text{value}(x) = -1$  then
11         $\text{value}(x) \leftarrow \ell.\text{phase}$ ;
12      else
13         $\text{value}(x) \leftarrow$  a random value from  $\{0,1\}$ ;
14      else
15         $x \leftarrow \text{GetUnassignedVar}()$ ;
16         $\text{value}(x) \leftarrow$  a random value from  $\{0,1\}$ ;
17      Simplify  $F$  accordingly;
18      foreach newly generated unit clause  $r$  do
19        if  $r \notin \mathcal{U} \ \& \ \bar{r} \notin \mathcal{U}$  then
20           $\mathcal{U} \leftarrow \mathcal{U} \cup \{r\}$ ;
21        else if  $\bar{r} \in \mathcal{U}$  then
22           $\text{value}(r.\text{var}) \leftarrow -2$ ;
23      if  $\#(\square) \geq \text{cost}^*$  then break;
24  if  $\#(\square) < \text{cost}^*$  then
25     $\alpha^* \leftarrow \text{value}$ ;  $\text{cost}^* \leftarrow \#(\square)$ ;
26 return  $\alpha^*$ ;

```

If \mathcal{U} is not empty, one literal ℓ is extracted from \mathcal{U} via the function **GetUL** to do unit propagation. Let us denote $x = \ell.\text{var}$. We know that x could not have been assigned (either to 0 or 1). This is because if x is assigned, literals of x would not appear in the formula and \mathcal{U} . Thus, $\text{value}(x)$ is either -1 or -2. If $\text{value}(x) = -1$ (e.g., x is unassigned), then x is assigned the value of $\ell.\text{phase}$ to satisfy the unit clause ℓ ; if $\text{value}(x) = -2$, x is assigned randomly. We would like to mention that, most variables are assigned by UP in the CnC algorithm.

If \mathcal{U} is empty, then an unassigned variable is chosen by the **GetUnassignedVar** function, and is assigned a random value.

Whenever a variable x is assigned a value v , the formula F is simplified accordingly. The result of simplifying F on a literal ℓ can be described succinctly as $F|_{\ell} = \{c/\{\bar{\ell}\} | c \in F, \ell \notin c\}$ [12]. Moreover, for any newly generated unit clause r , if neither r nor \bar{r} is in \mathcal{U} , then r is added into \mathcal{U} ; if \bar{r} is already in \mathcal{U} , we set $\text{value}(r.\text{var})$ to -2 to indicate the conflicting status.

4.2 Main Functions

There are two functions that need to be specified in the CnC algorithm, and they are presented below.

Algorithm 2 Local Search with CnC.

Input: A CNF formula ϕ
Output: A satisfying assignment of ϕ if found

```

1 while not reach time limit do
2    $\alpha_0 \leftarrow \text{CnC}(\phi, \text{cnc\_times});$ 
3   if  $\alpha_0$  satisfies  $\phi$  then return  $\alpha_0$ ;
4    $\alpha \leftarrow \text{LocalSearch}(\alpha_0, \text{StepLimit});$ 
5   if  $\alpha$  satisfies  $\phi$  then return  $\alpha$ ;
6 return "UNKNOWN";

```

GetUL: the function picks a unit clause in \mathcal{U} to perform UP. In the first construction procedure, the function simply picks a random unit clause to perform UP. For the following construction procedures, the function utilizes a strategy as follows. The idea is to employ assigning orders as distant as possible in different tries, so as to exploit diverse reason chains, among which a good one may be touched. Our heuristic is based on a diversification property. For a variable, we use $\text{prev_assign_step}(x)$ to denote the step number in which it was assigned in the previous try of CnC. Our heuristic prefers to pick a variable with the largest prev_assign_step value.

GetUnassignedVar: the function picks an unassigned variable to assign value. We use the same heuristic as in **GetUL**. In the first construction procedure, a randomized strategy is used, and in other procedures a diversification strategy is employed to pick the one with the largest prev_assign_step values.

An important implementation detail is that we use a sampling method for approximately implementing the heuristic of picking a variable with the largest prev_assign_step value. We randomly pick a certain number (which is fixed to 10 according to the preliminary experiments) of candidate variables and pick the one with the largest prev_assign_step value. So, we do not need to sort the variables or scan all of them in each iteration. This allows the linear complexity of our method, as picking a variable to assign can always be done in $O(1)$ time, and the unit propagation in one iteration can be done in $\Delta_\phi(x_i)$ in the worst case, where x_i is the chosen variable. Since $\sum_{i=1}^n \Delta_\phi(x_i) = L(\phi)$, the worst case complexity of one CnC try is bounded by $O(L(\phi))$, where $L(\phi)$ is the length of the formula ϕ .

5 Integrating CnC to Local Search SAT Solvers

In this section, we apply the CnC method to improve local search SAT solvers. The framework of a local search SAT solver equipped with CnC is depicted in Algorithm 2. As it shows, the solver calls CnC to produce an initial assignment, which is handed to a local search algorithm for further improvements, trying to find a satisfying assignment. Local search SAT solvers may have different restart criterion, which is based on a limit on the steps. So, for each time the solver restarts, an initial assignment is produced by CnC and then modified by a local search process.

We apply the CnC method to three state-of-the-art local search SAT solvers for structured formulas, including *Sattime* [20], *ProbSAT* [4] and *CCAnr* [8]. *Sattime* is the only example that a local search solver beats all CDCL solvers in the crafted track of a SAT competition (in 2011) [20]. *ProbSAT* is a local search solver based on probability distribution and won the random track in SAT Competition 2013; it is an improved version of another local search solver *Sparrow* [2], which also uses probability distribution functions. *CCAnr* is a local search designed with the purpose of solving non-random (structured) SAT instances, and has been found effective on some application benchmarks [14].

We also note that a recent local search solver *YalSAT* performs well on a wide range of benchmarks, winning the random track of SAT Competition 2017, and is able to solve some hard crafted and application instances in SAT competitions [5]. Nevertheless, *YalSAT* utilizes the Luby restarting scheme [24] and has very frequent restart in the early stage. This makes it ineffective to integrate CnC into *YalSAT*, due to the heavy overhead.

6 Experiments

To evaluate the effectiveness of our CnC method, we compare state-of-the-art local search solvers with their CnC enhanced versions on three important benchmarks of structured SAT formulas. Also, we compare the best CnC-enhanced local search solver against state-of-the-art CDCL solvers.

6.1 Benchmarks

Our experiments are conducted with three important benchmarks, including instances encoded from a real-world project and two important mathematical problems.

FCC: Recently, SAT solvers have been used by the US Federal Communication Commission (FCC) for spectrum repacking in the context of bandwidth auction which resulted in about 7 billion dollar revenue [32]. The SAT instances from this project are available on line ¹ [32]. This benchmark contains 10000 instances, 9482 of which are known to be satisfiable and 121 unsatisfiable, while the satisfiability of the remaining 397 instances are unknown. As local search solvers such as UPLS are unable to prove unsatisfiability, we discard the unsatisfiable instances, leading to 9879 instances in this benchmark.

PTN: This benchmark consists of instances encoded from a mathematical problem named Boolean Pythagorean Triples. This problem used to be a long-term open mathematical problem and recently has been solved by SAT techniques, resulting in the currently largest-sized mathematical proof [17]. Marijn et al. proved the answer to Boolean Pythagorean Triples (PTN) problem is NO, by encoding PTN into SAT instances, including both satisfiable and unsatisfiable ones, and solving them. Our PTN benchmark contains only the satisfiable instances.² There are 23 instances in this benchmark.

SN5: The instances in this benchmark are encoded from a mathematical problem called Schur Number Five (SN5) and its variants. [16] proved the solution by encoding the century-old problem into SAT instances, and the proof of the solution is about two petabytes in size. Our SN5 benchmark contains 6 satisfiable instances.³

6.2 Solvers

The CnC method is implemented into the local search solvers in C++. For a local search solver *A*, the solver which integrates CnC is denoted as *A+cnc* in our experiments. The *cnc_times* parameter is set to 20, which is tuned on a training set consisting of 100 random FCC instances, half PTN instances and all SN5 instances.

¹ https://www.cs.ubc.ca/labs/beta/www-projects/SATFC/cacm_cnfs.tar.gz

² <https://www.cs.utexas.edu/~marijn/ptn/>

³ <https://www.cs.utexas.edu/~marijn/Schur/>

■ **Table 1** Results of local search solvers and CnC-enhanced local search solvers on all benchmarks.

Benchmark	<i>CCAnr</i>		<i>CCAnr+cnc</i>		<i>ProbSAT</i>		<i>ProbSAT+cnc</i>		<i>Sattime</i>		<i>Sattime+cnc</i>		<i>YalSAT</i>	
	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2
FCC (9879)	7878	2091.6	8110	1868.2	5407	4577.7	5477	4506.5	7054	2911.8	7078	2900.0	7136	2881.1
PTN (23)	13	4718.0	23	127.0	5	7885.0	20	2161.7	9	6790.7	18	2945.3	14	4490.3
SN5 (6)	2	7364.5	4	4969.5	0	10000.0	0	10000.0	0	10000.0	1	8708.7	0	10000.0

■ **Table 2** Results of *CCAnr+cnc* and its CDCL competitors on all benchmarks.

	<i>CCAnr+cnc</i>		<i>CaDiCaL</i>		<i>CaDiCaL_sat</i>		<i>Maple_LCM_Dist</i>		<i>MapleCOMSPS</i>		<i>Kissat</i>		<i>Kissat_sat</i>	
	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2	#SAT	PAR2
FCC (9879)	8110	1868.2	7674	2326.9	7783	2211.9	7788	2183.2	7783	2183.0	7949	2042.8	8163	1819.1
PTN (23)	23	127.0	17	3274.2	17	3007.4	0	10000.0	1	9639.0	19	2215.7	21	1402.5
SN5 (6)	4	4969.5	0	10000.0	0	10000.0	0	10000.0	0	10000.0	0	10000.0	1	9130.7

The solvers *Sattime* and *ProbSAT* are downloaded from the website of SAT Competition 2013. For *CCAnr*, we used the latest version which is available online⁴. We include *YalSAT* in our experiment, which is downloaded from the website of SAT Competition 2017.⁵ Additionally, we tested *UnitWalk* [18] – a typical local search solver using unit propagation.⁶

We also compare the best local search solver obtained by CnC (namely *CCAnr+cnc*) against four state-of-the-art CDCL solvers, including *MapleCOMSPS* [23], *Maple_LCM_Dist* [30], *CaDiCaL* [5] and *Kissat* (including *Kissat_default* and *Kissat_sat*) [6]. *MapleCOMSPS* won the gold medal of Main Track of SAT Competition 2016 and the silver medal of Main Track of SAT Competition 2017, while *Maple_LCM_Dist* won the gold medal of Main Track of SAT Competition 2017 and the winner of the main track of SAT Competition 2018 is also a version of *Maple_LCM_Dist*. *CaDiCaL* solved the most instances in the Main Track of SAT Competition 2019. Particularly, *CaDiCaL* solved the most satisfiable instances in the track. Also, *Kissat_sat* won the gold medal of Main Track of SAT Competition 2020. All these CDCL solvers are downloaded from the website of SAT Competitions.

6.3 Experiment Results

All experiments were conducted on a cluster of computers with 2.10GHz Intel Xeon CPUs and 94GB RAM under the operating system CentOS. For each instance, each solver was performed one run, with 5000 CPU seconds as cutoff. For each solver for each benchmark, we report the number of solved SAT instances denoted “#SAT” and the penalized run time denoted “PAR2” (as used in SAT Competitions), where the run time of a failed run is penalized as twice the cutoff time. The results in **bold** indicates the best performance for a benchmark.

Table 1 presents the results of the local search solvers on the three benchmarks. *UnitWalk* performs much worse than other solvers (solving 4597 FCC instances and none of the other two benchmarks) and is not listed in the table. The CnC method improves local search solvers, particularly on the PTN instances. *CCAnr+cnc* gives the best performance on all the benchmarks. It solves 8110 out of 9879 FCC instances, 4 out of 6 SN5 instances and all PTN instances, showing significantly superiority over all other local search solvers.

⁴ <https://lcs.ios.ac.cn/~caisw/Code/CCAnr-1.1.zip>

⁵ <https://baldur.itk.kit.edu/sat-competition-2017/solvers/>

⁶ <https://logic.pdmi.ras.ru/~arist/UnitWalk/unitwalk3.tar.gz>

We compare *CCAnr+cnc* with state-of-the-art CDCL solvers. Table 2 shows the results of *CCAnr+cnc* and its CDCL competitors. The best CDCL solver is *Kissat_sat*, which outperforms other CDCL solvers on all the benchmarks. Encouragingly, *CCAnr+cnc* is able to solve more instances than the CDCL solvers on all the benchmarks, with only one exception – *CCAnr+cnc* performs a bit fewer FCC instances than *Kissat_sat*. Particularly, *CCAnr+cnc* solves four SN5 instances, while *Kissat_sat* solves only one SN5 instance and other CDCL solvers fail to solve any of them. Note that these benchmarks are encoded from real-world applications or mathematical problems of importance. Our results show that local search solvers can be complementary to CDCL solvers in applications.

We also calculate the overhead of CnC in SLS+CnC solvers. Averaging over all instances, the run time of CnC occupies about 1% run time of the whole process.

7 Conclusions

This work presented an effective method named construct-and-cut (CnC for short) for generating initial assignments for local search SAT solvers. Our experiments on three benchmarks from real-world project and mathematical problems showed that, the CnC method can significantly improve the performance of local search SAT solvers on the benchmarks. More encouragingly, one CnC-enhanced local search solver *CCAnr+cnc* outperformed state-of-the-art CDCL solvers on these benchmarks. The source code of *CCAnr+cnc* is available at <https://github.com/caiswgroup/CnC-LS>.

References

- 1 Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Sais. Boosting local search thanks to CDCL. In *Proceedings of LPAR 2010*, pages 474–488, 2010.
- 2 Adrian Balint and Andreas Fröhlich. Improving stochastic local search for SAT with a new probability distribution. In *Proceedings of SAT 2010*, pages 10–15, 2010.
- 3 Adrian Balint, Michael Henn, and Oliver Gableske. A novel approach to combine a SLS- and a DPLL-solver for the satisfiability problem. In *Proceedings of SAT 2009*, pages 284–297, 2009.
- 4 Adrian Balint and Uwe Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. In *Proceedings of SAT 2012*, pages 16–29, 2012.
- 5 Armin Biere. Splatz, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT competition 2016. In *Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions*, pages 44–45, 2016.
- 6 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. In *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*, pages 50–53, 2020.
- 7 Shaowei Cai, Chuan Luo, Jinkun Lin, and Kaile Su. New local search methods for partial MaxSAT. *Artificial Intelligence*, 240:1–18, 2016.
- 8 Shaowei Cai, Chuan Luo, and Kaile Su. CCAnr: A configuration checking based local search solver for non-random satisfiability. In *Proceedings of SAT 2015*, pages 1–8, 2015.
- 9 Shaowei Cai, Chuan Luo, and Haochen Zhang. From decimation to local search and back: A new approach to MaxSAT. In *Proceedings of IJCAI 2017*, pages 571–577, 2017.
- 10 Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
- 11 Byron Cook, Daniel Kroening, and Natasha Sharygina. Cogent: Accurate theorem proving for program verification. In *Proceedings of CAV 2005*, pages 296–300, 2005.
- 12 Adnan Darwiche and Knot Pipatsrisawat. Complete algorithms. In *Handbook of Satisfiability*, pages 99–130. IOS Press, 2009.
- 13 Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

- 14 Andreas Fröhlich, Armin Biere, Christoph M. Wintersteiger, and Youssef Hamadi. Stochastic local search for satisfiability modulo theories. In *Proceedings of AAAI 2015*, pages 1136–1143, 2015.
- 15 Oliver Gableske and Marijn Heule. EagleUP: Solving random 3-SAT using SLS with unit propagation. In *Proceedings of SAT 2011*, pages 367–368, 2011.
- 16 Marijn J. H. Heule. Schur number five. In *Proceedings AAAI 2018*, pages 6598–6606, 2018.
- 17 Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. In *Proceedings of SAT 2016*, pages 228–245, 2016.
- 18 Edward A. Hirsch and Arist Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, 43(1):91–111, 2005.
- 19 Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.
- 20 Chu Min Li and Yu Li. Satisfying versus falsifying in local search for satisfiability. In *Proceedings of SAT 2012*, pages 477–478, 2012.
- 21 Chu Min Li and Yu Li. Description of Sattime 2013. In *Proceedings of SAT Competition 2013 : Solver and Benchmark Descriptions*, pages 77–78, 2013.
- 22 Xiao Yu Li, Matthias F. M. Stallmann, and Franc Brglez. A local search SAT solver using an effective switching strategy and an efficient unit propagation. In *Proceedings of SAT 2003*, pages 53–68, 2003.
- 23 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *Proceedings of SAT 2016*, pages 123–140, 2016.
- 24 Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.
- 25 Chuan Luo, Shaowei Cai, Kaile Su, and Wenxuan Huang. CCEHC: An efficient local search algorithm for weighted partial maximum satisfiability. *Artificial Intelligence*, 243:26–44, 2017.
- 26 Chuan Luo, Shaowei Cai, Kaile Su, and Wei Wu. Clause states based configuration checking in local search for satisfiability. *IEEE Transactions on Cybernetics*, 45(5):1014–1027, 2015.
- 27 Chuan Luo, Shaowei Cai, Wei Wu, Zhong Jie, and Kaile Su. CCLS: An efficient local search algorithm for weighted maximum satisfiability. *IEEE Transactions on Computers*, 64(7):1830–1843, 2015.
- 28 Chuan Luo, Shaowei Cai, Wei Wu, and Kaile Su. Double configuration checking in stochastic local search for satisfiability. In *Proceedings of AAAI 2014*, pages 2703–2709, 2014.
- 29 Chuan Luo, Holger H. Hoos, and Shaowei Cai. PbO-CCSAT: Boosting local search for satisfiability using programming by optimisation. In *Proceedings of PPSN 2020*, pages 373–389, 2020.
- 30 Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In *Proceedings of IJCAI 2017*, pages 703–711, 2017.
- 31 Norbert Manthey. Coprocessor 2.0 – A flexible CNF simplifier – (tool presentation). In *Proceedings of SAT 2012*, pages 436–441, 2012.
- 32 Neil Newman, Alexandre Fréchette, and Kevin Leyton-Brown. Deep optimization for spectrum repacking. *Communications of the ACM*, 61(1):97–104, 2018.
- 33 João P. Marques Silva and Karem A. Sakallah. GRASP – A new search algorithm for satisfiability. In *Proceedings of ICCAD 1996*, pages 220–227, 1996.

Unit Propagation with Stable Watches

Markus Iser 

Karlsruhe Institute of Technology (KIT), Germany

Tomáš Balyo 

CAS Software AG, Karlsruhe, Germany

Abstract

Unit propagation is the hottest path in CDCL SAT solvers, therefore the related data-structures, algorithms and implementation details are well studied and highly optimized. State-of-the-art implementations are based on reduced occurrence tracking with two watched literals per clause and one blocking literal per watcher in order to further reduce the number of clause accesses. In this paper, we show that using runtime statistics for watched literal selection can improve the performance of state-of-the-art SAT solvers. We present a method for efficiently keeping track of spans during which literals are satisfied and using this statistic to improve watcher selection. An implementation of our method in the SAT solver CaDiCaL can solve more instances of the SAT Competition 2019 and 2020 benchmark sets and is specifically strong on satisfiable cryptographic instances.

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

Keywords and phrases Unit Propagation, Two-Watched Literals, Literal Stability

Digital Object Identifier 10.4230/LIPIcs.CP.2021.6

Category Short Paper

Supplementary Material *Software:* https://github.com/sat-clique/cadical_stability

1 Introduction

Boolean satisfiability (SAT) solvers are used in a large variety of applications, e.g., software and hardware verification [2], automated planning [11], or cryptography [10]. Complete state-of-the-art SAT solvers are based on the Conflict-Driven Clause Learning (CDCL) algorithm [8, 9]. CDCL conducts a series of decisions with subsequent unit propagation and conflict resolution. The runtime of CDCL is dominated by the runtime of unit propagation [5]. Unit propagation is the process of inferring a new assignment from a current partial assignment and the given set of clauses. That requires to map literals which are not satisfied by the current partial assignment to the clauses in which they occur. One can effectively reduce the number of clause accesses for unit-clause and conflict detection by watching only two literals per clause [9, 3].

In this paper we propose a new method for selecting the two literals to be watched for each clause. We introduce the notion of stable literals, i.e., literals that tend to be satisfied for long time periods during the CDCL search. In our method, such stable literals are preferred when selecting new watched and blocking literals.

We implemented our method by modifying the well-known state-of-the-art SAT solver CaDiCaL [1]. Compared to the original CaDiCaL, our modified version can solve more benchmark instances of the Main tracks of SAT Competitions 2019 and 2020 and performs specifically well on a set of satisfiable cryptographic instances. Additionally, the two versions are rather orthogonal in the sense that they perform well on different subsets of the benchmark instances.



© Markus Iser and Tomáš Balyo;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 6; pp. 6:1–6:8

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Preliminaries and Related Work

A *Boolean variable* can take on two possible values: *True* and *False*. A *literal* is a Boolean variable (positive literal) or a negation of a Boolean variable (negative literal). A *clause* is a disjunction (\vee) of literals and a formula is a conjunction of clauses. A clause with only one literal is called a *unit clause*. A positive (resp. negative) literal is satisfied if the corresponding variable is assigned the value *True* (resp. *False*). A clause is satisfied, if at least one of its literals is satisfied and a formula is satisfied, if all its clauses are satisfied.

The satisfiability (SAT) problem is to determine whether a given formula has a satisfying assignment, and if so, also find it. A key component of the CDCL algorithm is unit propagation, which is the following process. Given a unit clause $C = \{l\}$, l has to be satisfied in each possible model of the formula, therefore we can immediately assign l 's variable such that l is satisfied. Next we remove each clause that contains l from the formula (since these clauses are already satisfied) and remove \bar{l} from each clause that contains \bar{l} (since these clauses cannot be satisfied by this literal anymore). Removing literals from clauses may produce new unit clauses or an empty clause. The process is repeated until no more unit clauses are found or until an empty clause is generated. In the latter case, the algorithm has uncovered a conflict between the partial assignment and the formula.

Unit propagation dominates the runtime of the CDCL algorithm [5], therefore it is of paramount importance to implement this procedure as efficiently as possible. Currently the best known implementations of unit propagation are based on the idea of two-watched literals [9, 3]. As long as both watched literals are unassigned or satisfied, the clause can be ignored. When a watched literal is falsified, we must check if some of the clauses where it is watched became unit or empty, otherwise we find a new literal to watch for those clauses.

Competitive implementations of literal watch lists store a so-called *blocking literal* next to each clause pointer. The blocking literal is an arbitrary literal from the clause. If the blocking literal is satisfied, the clause access can be skipped. Another optimization which is used in state-of-the-art implementations is that the search for a new literal to watch does not start at the beginning of the clause. Instead, the position of the last found watching literal is stored in the clause and search starts from there (cycling through the beginning when we pass by the end) [4].

Epochs in CDCL SAT solvers can be measured in several ways. The naive approach is to measure an epoch in terms of real time, which is usually not a good idea as it hurts the reproducibility of runtime results [7]. Epochs are better measured in terms of assignment phases, i.e., the number of decisions, conflicts, or propagations. Recently, Biere came up with the dedicated epoch *ticks* which approximates the number of accessed cache lines during propagations [1]. In the following, an epoch n denotes the solver state in which we process the n -th decision – the *number of decisions*.

3 Selecting Stable Literals to Watch

The *stability* of a literal l is the number of epochs in which l has been satisfied. In Section 3.1, we show how to efficiently maintain and calculate literal stability. Our modified unit propagation periodically prioritizes literals of high stability as watched and blocking literals. We call this concept *stable watches* and explain it in Section 3.2.

■ **Procedure** `limp`(Literal l).

Data: Stability $S : \text{Literals} \rightarrow \mathbb{N}$

Data: Number of Decisions N

$S[l] \leftarrow N - S[l]$

3.1 Literal Stability

The *stability* of a literal l denotes the total number of epochs (as specified by the total number of decisions) in which l has been satisfied. Those epochs are given by a set of tuples $\{(T_i(l), U_i(l)) \mid 0 < i \leq n\}$ of start- and stop-epochs, in which $T_i(l)$ denotes the i -th epoch in which l becomes True (satisfied) and $U_i(l)$ denotes the i -th epoch in which l becomes unassigned or unsatisfied. It is easy to see that $U_i(l) \geq T_i(l) \geq U_{i-1}(l) \geq \dots \geq T_1(l)$. The stability $S_n(l)$ of a literal l is given by Definition 1.

► **Definition 1** (Literal Stability). *Given a literal l which is satisfied in epochs $\{(T_1(l), U_1(l)), \dots, (T_n(l), U_n(l))\}$, its stability $S_n(l)$ is defined as follows.*

$$S_n(l) := \sum_{i=1}^n (U_i(l) - T_i(l))$$

We can incrementally update literal stability during backtracking in an epoch $U_i(l)$ by using the recursive form $S_n(l) = (U_i(l) - T_i(l)) + S_{n-1}(l)$. However, this requires to additionally keep track of epochs $T_i(l)$ in which l gets assigned to true. By reformulating the recursive form like in Equation 1, we can store $T_i(l)$ as an intermediate state of $S_n(l)$ in order to save some cache on a hot path in the solver.

$$S_n(l) := U_i(l) - (T_i(l) - S_{n-1}(l)) \quad (1)$$

In our implementation, we update $S_n(l)$ with the dirty intermediate value $S_n^d(l)$ (Equation 2) when it is assigned to true and then use $S_n^d(l)$ to calculate the new literal stability $S_n(l)$ when l is backtracked (Equation 3).

$$S_n^d(l) = T_i(l) - S_{n-1}(l) \quad (2)$$

$$S_n(l) = U_i(l) - S_n^d(l) \quad (3)$$

Our method for interval accumulation is specified by Procedure `limp`. Procedure `limp` is called twice per interval, first when a literal l becomes satisfied by a taken assignment in epoch T and again when that assignment is undone by backtracking in epoch $U \geq T$. Between T and U , literal stability $S[l]$ is in its dirty state. After each second call to `limp`(l, U), $S[l]$ is a valid sum of intervals.

3.2 Selecting Stable Watches

Traditional implementations of the two watched literal algorithm start by watching the first two literals in the clause and also blocking literals are initialized accordingly. During search new watchers are found according to the order in which literals are stored. Also the blocking literal is updated lazily during propagation.

Our approach exploits the fact that watched and blocking literal initialization starts with the first literals in the clause and then progresses along the order of literals in the clause. Watcher (re-)initialization takes place regularly in *cleanup phases* for learned clause

■ **Procedure** StableWatches(Assignment A , Clauses C).

```

Data: Stability  $S : \text{Literals} \rightarrow \mathbb{N}$ 
Data: Value  $v : \text{Literals} \rightarrow \{0, 1, 2\}$ 

// Cleanup Stability Values
1 for Literal  $l \in A$  do limp( $l$ )

// Apply Stability-Induced Priorities
2 for Clause  $c \in C$  do
3   if  $c$  is not Reason Clause then
4      $\lfloor$  stable_sort( $c.\text{literals}$ ,  $l_0 < l_1 \iff v(l_0)S(l_0) > v(l_1)S(l_1)$ )

// Revert to Dirty State
5 for Literal  $l \in A$  do limp( $l$ )

```

forgetting and memory defragmentation. By reordering the literals in each clause, we control the order in which literals are considered as watching and blocking literals. For each clause, we sort literals in a descending order according to their *literal stability*.

Procedure StableWatches outlines our method. It is called after cleanup and before reattaching clauses to the watcher data-structure. Before we can use the accumulated literal stabilities, we have to fix those values which are currently in their dirty state, and revert their values after sorting (Lines 1 and 5). In order to protect the relative order of literals with the same sorting value, we use stable sorting.

Since there exists a partial assignment that can falsify even the most stable literal of a clause, we must also be careful not to watch a falsified literal. Therefore, we multiply the stability of currently false literals by zero. We place additional weight on the stability of a literal that is also currently satisfied by multiplying its stability by two. This was very easy to implement based on the value function already available in CaDiCaL. The factors are given by the value function $v(l)$, which is as follows.

$$v(l) = \begin{cases} 0 & \text{if } l \text{ is false} \\ 1 & \text{if } l \text{ is unassigned} \\ 2 & \text{if } l \text{ is true} \end{cases}$$

Then, for each clause (Line 2) which is not a reason clause (Line 3)¹, we (stable-) sort its literals in a descending order according to the value of the product of their stability and the value function (Line 4).

4 Evaluation

We experimentally investigated the effectiveness and efficiency of our methods. Our experiments were executed on a cluster of 20 compute nodes, each equipped with 32 GiB RAM and 2×2.66 GHz Intel Xeon E5430 CPU. The operating system is *Ubuntu 18.04.4 LTS*, *Linux Kernel 5.4.0-66*. We ran 2 processes per node and used a time limit of 5000 seconds and a memory limit of 16 GiB per benchmark instance.

¹ A clause can be reason for propagation in the current partial assignment, in which case literal order carries additional semantics.

■ **Table 1** PAR-2 score and number of solved instances for several variants of watched literal prioritization in **Candy**.

Method	PAR-2 Score	Solved Instances
Literal Stability	6747	152
Literal Constrainedness (Desc.)	6844	151
Variable Constrainedness (Desc.)	6920	149
Default Performance	6952	145
Variable Constrainedness (Asc.)	6954	143
Literal Constrainedness (Asc.)	7201	132

We experimented with three sets of instances. The instance sets **Main-2020** (400 instances) and **Main-2019** (399 instances) correspond to the benchmark sets which were used in the Main tracks of the respective SAT Competitions. By projecting on the instance families represented in **Main-2020**, we found that our method seems specifically well suited for cryptographic instances. Our third benchmark set **Crypto** is a collection of 409 cryptographic instances of previous SAT competitions.² Both the number of instances solved and the average runtime with a penalty factor of two (PAR-2 score) are used to compare performance.

We also ran initial experiments with our SAT solver **Candy** on the instances in **Main-2020**. We report on those in Section 4.1. Later, we were able to reproduce and further analyze our results with the well-known state-of-the-art SAT solver **CaDiCaL** (Version 1.1.4) by Armin Biere. We report on results for **CaDiCaL** with all instances in Section 4.2.

4.1 Initial Results

Literal Stability emerged as a possible explanation for what happens in our initial experiments with (trivial) constrainedness-based watcher-priorities. Table 1 displays the preliminary results for several types of literal priorities, which we used for establishing watched literal priorities through recurrent watcher reinitialization in the clause forgetting intervals of our solver **Candy**. In our initial experiments, we sorted clauses by variable and literal constrainedness, both in ascending and descending order. To calculate constrainedness, we use the well-known Jeroslow-Wang score [6].

Our experimental data shows that prioritizing watched literals by low constrainedness leads to fundamentally worse performance than prioritizing those of high constrainedness. Prioritizing by high variable and literal constrainedness both outperform the original implementation. Prioritizing by literal stability however, shows the best performance, solving seven more instances than the default approach.

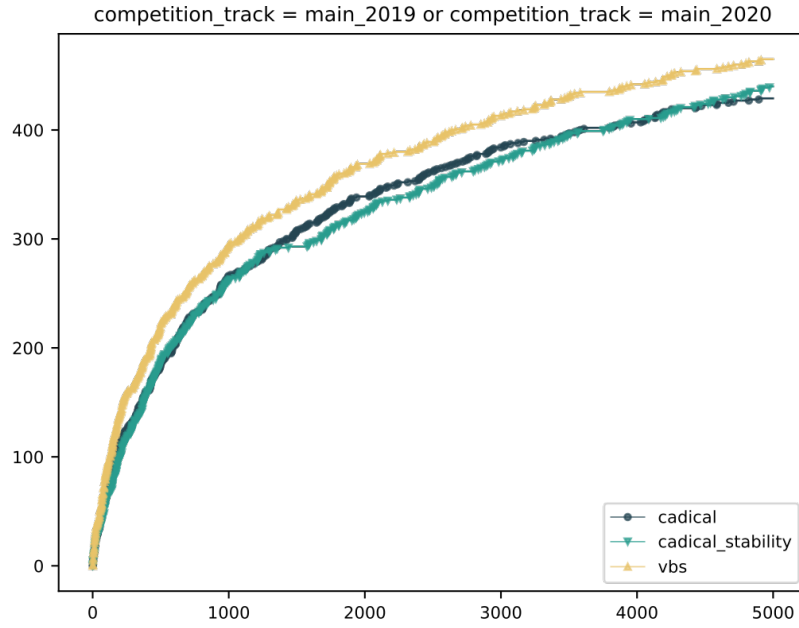
4.2 Experimental Results

A summary of the results of our experiments with **CaDiCaL** and our modified version **CaDiCaL Stability** is displayed in Table 2. We also included a combination of both versions that takes the best result for each benchmark – a virtual best solver (VBS). **CaDiCaL Stability** solves 6 instances more in **Main-2019** and 4 instances more in **Main-2020**. Figure 1 shows that our approach is stronger in long solver runs.

In **Main-2020**, our approach is particularly strong on the **station-repacking** and **cryptographic** families of instances. Of the 12 **station-repacking** instances, **CaDiCaL** solves 4 instances, while **CaDiCaL Stability** solves 10 instances. Of the 35 **cryptographic** instances, **CaDiCaL** solves

² Query for family = cryptography at <https://gbd.iti.kit.edu/>

6:6 Unit Propagation with Stable Watches



■ **Figure 1** Cumulated runtimes of CaDiCaL with and without Stable Watches, and their VBS on the benchmarks of SAT Competitions 2019 and 2020.

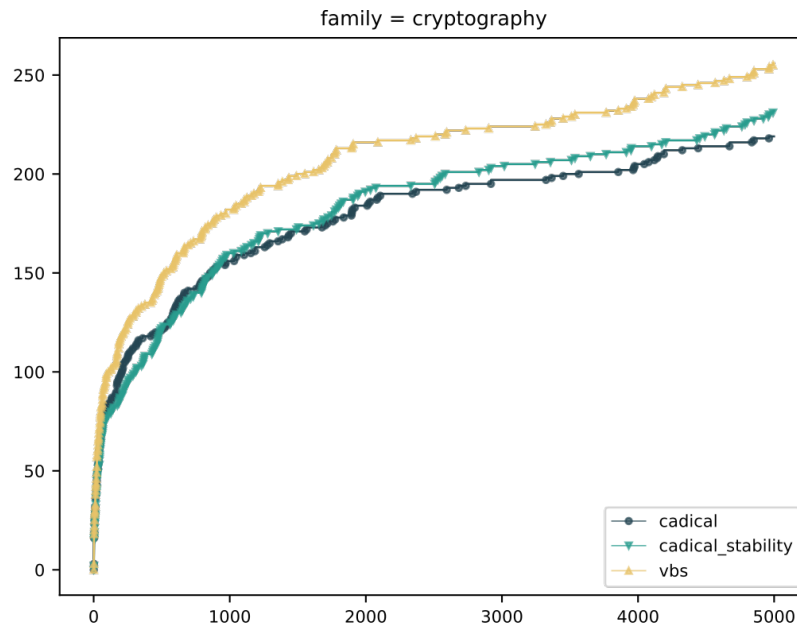
16 instances, while CaDiCaL Stability solves 20. Table 2 and Figure 2 show, that our approach performs significantly better on the 409 instances in **Crypto**. CaDiCaL Stability solves 13 instances more in **Crypto** with a significantly better PAR-2 score.

The clear winner is however VBS, a theoretical solver combining both approaches with a perfect oracle that selects for each instance the fastest approach. This suggests that the two CaDiCaL versions are orthogonal and would work well together in a portfolio.

Since our approach comes with the overhead of maintaining literal stability statistics, which is done once per value assignment and again during backtracking, we measured whether we actually speed-up unit propagation, i.e., watcher iteration. The average number of propagations per second (PPS) over all instances in Main-2019, Main-2020 and **Crypto** goes down from 1.06 million PPS for CaDiCaL to 0.99 million PPS for CaDiCaL Stability. Also the total number of propagations goes down from 2.23 billion propagations for CaDiCaL to 2.16 billion propagations for CaDiCaL Stability. So on average, our approach does less propagations per second, but it also needs a lower total number of propagations to solve the benchmark instances.

■ **Table 2** PAR-2 score and number of solved instances of CaDiCaL, CaDiCaL Stability and their VBS on several benchmarks.

		CaDiCaL	CaDiCaL Stability	VBS
Main-2019	Solved	229	235	243
	PAR-2	4937.9	4890.3	4581.0
Main-2020	Solved	215	219	237
	PAR-2	5212.2	5221.2	4732.5
Crypto	Solved	222	235	259
	PAR-2	5343.5	5146.7	4593.9



■ **Figure 2** Cumulated Runtimes of CaDiCaL with and without Stable Watches, and their VBS on a set of cryptographic instances aggregated from several SAT Competition benchmarks.

5 Conclusion

We showed that we can afford the overhead of maintaining literal stability values on the assignment level (which is a hot path). Using stability values to establish priorities for watched literals leads to improved SAT solver performance, particularly on *satisfiable cryptographic* instances. We also showed that the observed performance gain is *not* due to an increased number of propagations per second but by requiring less total propagations to solve the benchmark instances.

The internal state of the watcher data-structure determines *propagation order*. A partial assignment can be conflicting for several reasons. With stable watches we break ties differently such that we analyze *different conflicts*. In the presented approach, propagation-ties are resolved in favor of clauses which are less stable (or more rarely satisfied). We could empirically show that this helps finding solutions for hard satisfiable instances more quickly.

In the future, we expect other effective tie-breakers to be discovered and analyzed. Future work should focus on how exactly resolution space navigation is affected by propagation order for several types of instances. In the recent SAT Competition 2021, Kaiser and Clausecker won a special price with their solver **CaDiCaL_PriPro**, which performs a different kind of prioritized propagation.³ This is an additional indication that propagation order is important.

Our modified CaDiCaL is kind-of orthogonal to the original CaDiCaL in the sense that it performs well on a different subset of the benchmark instances. This suggests, that combining our approach with the standard approach to select literals to watch could be a promising topic for future work. That might include research on hybrid heuristics or instance-specific heuristic selection.

³ <https://satcompetition.github.io/2021/downloads.html>

References

- 1 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proceedings of SAT Competition 2020*, pages 50–53. Department of Computer Science, University of Helsinki, 2020.
- 2 Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. *J. Satisf. Boolean Model. Comput.*, 6(1-3):165–201, 2009.
- 3 Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT Competition 2020. *Artificial Intelligence*, Accepted for Publication, 2021.
- 4 Ian P. Gent. Optimal implementation of watched literals and more general techniques. *J. Artif. Intell. Res.*, 48:231–251, 2013.
- 5 Steffen Hölldobler, Norbert Manthey, and Ari Saptawijaya. Improving resource-unaware SAT solvers. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 519–534. Springer, 2010.
- 6 Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.*, 1:167–187, 1990.
- 7 Stepan Kochemazov. F2TRC: deterministic modifications of SC2018-SR2019 winners. In Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*, pages 21–22. Department of Computer Science, University of Helsinki, 2020.
- 8 Joao P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
- 9 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, 2001.
- 10 Saeed Nejati, Jan Horáček, Catherine H. Gebotys, and Vijay Ganesh. Algebraic fault attack on SHA hash functions using programmatic SAT solvers. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 737–754, 2018.
- 11 Dominik Schreiber, Damien Pellier, Humbert Fiorino, and Tomáš Balyo. Efficient SAT encodings for hierarchical planning. In Ana Paula Rocha, Luc Steels, and H. Jaap van den Herik, editors, *Proceedings of the 11th International Conference on Agents and Artificial Intelligence, ICAART 2019, Volume 2, Prague, Czech Republic, February 19-21, 2019*, pages 531–538. SciTePress, 2019.

Towards Better Heuristics for Solving Bounded Model Checking Problems

Anissa Kheireddine ✉ 

EPITA, LRDE, Kremlin-Bicêtre, France
Sorbonne Université, UMR 7606 LIP6, Paris, France

Etienne Renault ✉ 

EPITA, LRDE, Kremlin-Bicêtre, France

Souheib Baarir ✉

Sorbonne Université, CNRS UMR 7606 LIP6, France
Université Paris Nanterre, France

Abstract

This paper presents a new way to improve the performance of the SAT-based bounded model checking problem by exploiting relevant information identified through the characteristics of the original problem. This led us to design a new way of building interesting heuristics based on the structure of the underlying problem. The proposed methodology is generic and can be applied for any SAT problem. This paper compares the state-of-the-art approach with two new heuristics: Structure-based and Linear Programming heuristics and show promising results.

2012 ACM Subject Classification Hardware → Theorem proving and SAT solving; Hardware → Model checking

Keywords and phrases Bounded model checking, SAT, Structural information, Linear Programming

Digital Object Identifier 10.4230/LIPIcs.CP.2021.7

Category Short Paper

Supplementary Material *Software (Source Code)*: <https://gitlab.lrde.epita.fr/akheireddine/bmctool>; archived at `swb:1:dir:ef063522105b17855a9f7d76382848f5e7f2a150`

1 Introduction

Computer systems are omnipresent in our daily life. These range from the simple program that runs a microwave to the very complex software driving a nuclear power plant, passing by our smartphones and cars. Ensuring the reliability and robustness of these systems is an absolute necessity. Model-Checking [10] is one of the approaches devoted to this purpose. Its goal is to prove the absence of failure, or to show a possible one.

Model-Checking is declined into several techniques [8, 6, 19]. Among all, those called Bounded Model Checking (BMC) [5], based on Boolean satisfiability (SAT). BMC is very used for hardware formal verification in the context of electronic design automation¹, but is also applied to many other domains. The idea is to verify that a model, restricted to executions bounded by some integer k , satisfies its specification, given as a set of terms in a temporal logic. In this approach, behaviors are described as a SAT problem. The memory usage in SAT solving does not usually suffer from the well-known space explosion problem and can handle problems with thousands of variables and constraints. The complexity here is shifted to the solving time: SAT problems are NP-complete problems in general [28].

¹ <http://fmv.jku.at/hwmc20/index.html>



These last decades, many improvements have been developed in the context of sequential SAT solving² [31, 2, 25, 22, 21], to name but a few. These approaches are quite generic and are based on exploiting either dynamic information, obtained from the progress of the solving algorithm itself (e.g., lbd [31]), or static information, derived from the underlying structure of the SAT problem (e.g., community [2]). Little attention has been given to structural information that can be extracted and exploited from the original problem (e.g., planning, scheduling, cryptography, BMC, etc.).

Indeed, when reducing a BMC problem to SAT, crucial information is lost. As we will highlight in this work, when reintegrated, this information can be a booster for the solving process. To the best of our knowledge, this paper is the first one to exploit such insights: existing approaches working on improving SAT-based BMC [14, 17, 33, 15, 32] either focus on improving existing (generic) heuristics or on dividing efficiently the SAT problem.

This paper aims to propose a methodology (Section 4) to build new heuristics (Section 5). This methodology is generic and can improve SAT-solvers for any problem with its specific characterization. Here, we apply the proposed techniques to build efficient SAT-solvers dedicated for BMC problems. Our results (Section 6) are promising and demonstrate the interest of exploiting the information provided by the underlying problem.

2 Preliminaries

2.1 SAT problem

A propositional variable can have two possible values \top (True) or \perp (False). A literal is a propositional variable (x) or its negation ($\neg x$). A clause ω is a finite disjunction of literals. For a given clause ω , $V(\omega)$ denotes the set of variables composing ω . A clause with a single literal is called unit clause. A conjunctive normal form (CNF) formula F is a finite conjunction of clauses (by abuse of notation, $F = \{\omega_1, \omega_2, \dots\}$). For a given F , the set of its variables is noted V . An assignment \mathbf{A} of variables of F is a function $\mathbf{A} : V \rightarrow \{\top, \perp\}$. \mathbf{A} is total (complete) when all elements of V have an image by \mathbf{A} , otherwise it is partial. For a given formula F and an assignment \mathbf{A} , a clause of F is satisfied when it contains at least one literal evaluating to true regarding \mathbf{A} . The formula F is satisfied by \mathbf{A} iff $\forall \omega \in F, \omega$ is satisfied. F is said to be SAT if there is at least one assignment that makes it satisfiable. It is defined as UNSAT otherwise.

Conflict Driven Clause Learning [34]. Conflict-Driven Clause Learning algorithm (CDCL) is one of the main methods used to solve Satisfiability problems and is an enhancement of the DPLL algorithm [12]. CDCL algorithm performs a backtrack search; selecting at each node of the search tree, a decision literal which is set to a Boolean value. This assignment is followed by an inference step that deduces and propagates some forced unit literal assignments (procedure called *unit propagation*). This branching process is repeated until finding a model or reaching a conflict. In the first case, the formula is answered to be satisfiable, and the model is reported, whereas in the second case, a **learned clause** is generated (by resolution), following a bottom-up traversal of the implication graph [30] (it is called *conflict analysis*).

² Our focus here is on CDCL-like complete algorithms [34].

2.2 SAT-based Bounded Model Checking

Model checking [10] aims at checking whether a model satisfies a property. The model is usually given as a program, defined in a formal language, while the property is given as formula expressed in temporal logic (e.g., LTL [27]). A property is said to be verified if no execution in the model can invalidate it, otherwise it is violated. To achieve this verification a full traversal of the state-space, representing the behaviours of the model, is required.

An LTL property refers to atomic propositions that express a relation between some variables of the model. The model checking approach usually represents the model as a finite-state automaton called a Kripke structure [4]. Such a structure is defined by a 4-uple $K = \langle S, s_0, T, L \rangle$ with: S a finite set of states, $s_0 \in S$ an initial state, $T \subseteq S \times S$ a transition relation, and L a labelling function that provides, for each state $s \in S$, an interpretation of an atomic proposition a denoted by $L(a)$. $L(a)$ is true iff a is satisfied in s .

Bounded Model Checking (BMC) [5, 9] refers to a model checking approach where the verification of the property is performed using a bounded traversal, i.e., a traversal of symbolic representation of the state-space that is bounded by some integer k . Such an approach does not require storing state space and hence, is found to be more scalable and useful [33, 16].

In SAT-based BMC, the BMC approach is reduced to solving a SAT problem. Given a model M , an LTL property p , and a bound k , it builds a propositional formula such that the formula is said to be satisfiable iff there exists a violation of the property (counterexample) of maximum length k . Otherwise, it is unsatisfiable and the property is verified up to length k . The encoding of this formula requires multiple steps.

First, it translates the model into a Boolean formula. The set of variables of this SAT formula can be decomposed in two disjoint subsets: \mathcal{M} and \mathcal{J} , where \mathcal{M} is a Boolean representation of the original variables of the model, while \mathcal{J} is a set of fresh variables (junction variables) used to finalize the conversion into a Boolean formula³. Second, the property p is also translated into a SAT formula. This conversion involves \mathcal{M} and \mathcal{J} and introduces new fresh variables \mathcal{F} . Let us denote by \mathcal{M}_p the set of variables of \mathcal{M} involved in p , \mathcal{J}_p the set of variables of \mathcal{J} involved in p . With these definitions we can build $\mathcal{P} = \mathcal{J}_p \cup \mathcal{M}_p \cup \mathcal{F}$, the set of the variables used to encode the property. Finally, the two previous steps are combined in the following formula:

$$\underbrace{I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-1}, s_k)}_{Model} \wedge \underbrace{P_k}_{Property} \quad (1)$$

It can be observed that both the transition relation of the model and the property have been unrolled up to the bound k . The left-side denotes the model constraints while the right-side is related to the property constraints. $I(s_0)$ are the initialization constraints that verify if s_0 is the the initial state of K , s_i represents the reachable states (in K) in i steps using the transition relation T .

3 Related work

Most of the works on improving SAT solving focus on building heuristics to detect and exploit relevant information during the solving process. Usually CDCL-like solvers maintain a database of interesting learnt clauses in order to speedup the solving. Good performances of

³ For instance, a 32 bits variable will be represented as 32 Boolean variables, and the logical operators (\wedge , \vee , \implies , ...) will rely on fresh variables for their representation.

these solvers are associated to their ability to preserve interesting clauses while maintaining a reasonable size for the database. So, the issue here is to find the best trade-off between what is considered to be a relevant information and how much of this information must be kept. Some of the state-of-the-art heuristics that are used in the best solvers of the world⁴ are described below:

Size bounded learning [13]. This approach protects learnt clauses that are sized less than a certain threshold.

Relevant bounded learning [18]. This approach discards learnt clauses when they are no longer relevant according to some metric. For instance, a learnt clause is considered as not relevant if the number of its literals that are assigned (w.r.t the current global assignment) exceeds predefined threshold.

Literal block distance (LBD) [31]. LBD is a positive integer, that is used as a learnt clause quality metric in almost all competitive sequential CDCL-like SAT-solvers. The LBD of a clause is the number of different decision levels on which variables of the clause have been assigned. Hence, the LBD of a clause can change overtime and it can be (re)computed each time the clause is fully assigned. If $LBD(\omega)=n$, then the clause ω spans on n propagation blocks, where each block has been propagated within the same decision level. Intuitively, variables in a block are closely related. Learnt clauses with lower LBD score tend to have higher quality: Glue Clauses [31] have LBD score of 2 and are the most important type of learnt clauses.

Community structure [2]. In this approach, the formula at hand is represented as a graph. The shape of this graph is then analyzed to extract community structure: roughly speaking, variables belonging to the same community are more densely interconnected than variables in different communities. Existing studies [2, 3] showed that using the community structure to detect new learnt clauses results in an improvement of the performance of the solver.

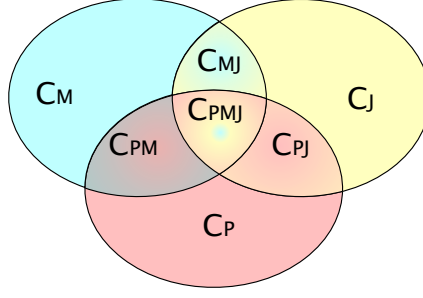
Symmetries [11]. SAT problems often exhibit symmetries, and not taking them into account forces solvers to needlessly explore isomorphic parts of the search space. Symmetries can help learning interesting clauses that the classical learning approaches fail to capture [25, 1]. Despite the generic character of these heuristics, they have been tuned by some research works in the case of the BMC problem. We can cite [29, 33, 17, 32] that present a variety of optimizations such as: variable ordering heuristics, branching heuristics, studying the symmetry structure of the BMC formula (1). Other works went for a decomposition of the BMC formula into simpler and independent subproblems showing promising results [14, 15].

4 Studying the characteristics of BMC problem

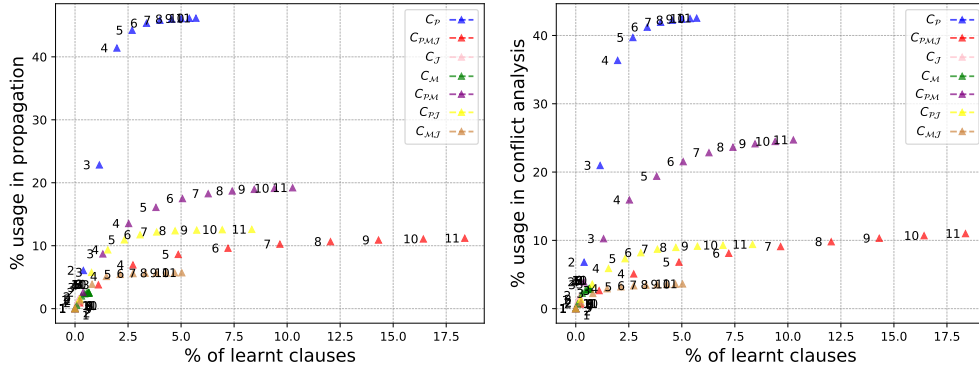
4.1 Intuition

The notion of what is a relevant information is quite unclear for SAT procedures. Most of the existing techniques are generic and try to perform well on any studied formula, without taking a real care of its origin (see Section 3). However, taking the structural information of the original problem into account will eventually lead to an improvement of the solving process. In this paper we explore this idea in the particular case of the BMC problem.

⁴ According to the results of the SAT competitions (<https://satcompetition.github.io/2020/results.html>).



■ **Figure 1** The seven disjunctive classes of clauses according to the combination of variables they handle: model variables (blue), fresh/junction variables (yellow) and property variables (red).



■ **Figure 2** Measures on the *training benchmark* showing learnt clauses usage in propagation (left) and conflict analysis (right) phases. Each class of clauses is colored and annotated by its LBD value.

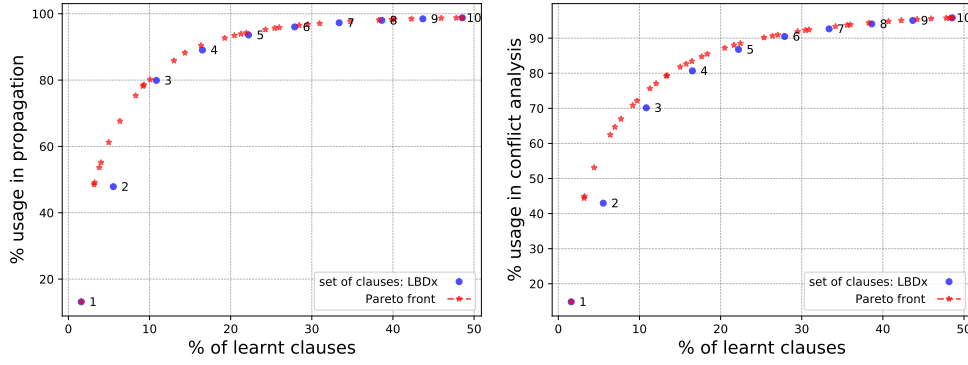
The starting point is to study the characteristics of the BMC problem. As a first insight, one can observe that the BMC problem can be trivially divided in two parts: the model and the property. However, when studying the learnt clauses w.r.t. this partitioning no relevant information could be inferred. Indeed, a learnt clause usually spans on variables belonging to both the model and the LTL property at the same time. So, we suggest here a sharper classification based on the clause variables.

A clause can be composed of variables belonging to \mathcal{M} , \mathcal{P} or \mathcal{J} . Let us denote by $C_X = \{\omega \in L \mid \forall v \in V(\omega), v \in X\}$ the classes highlighted in Figure 1, where X is either \mathcal{P} (the property), \mathcal{M} (the model), \mathcal{J} (the fresh variables for the model), \mathcal{PJ} (property and fresh variables), \mathcal{PM} (property and model variables), \mathcal{MJ} (model and fresh variables) or \mathcal{PMJ} (property, model and fresh variables). We can now study the usefulness of each of the above classes of clauses in the solving process.

4.2 Measures

Let us first precise our setup: all experiments of the paper were conducted on a benchmark of 400 SMV instances. The instances came from the SMV hardware verification problems [7], the BEEM [26] and the RERS Challenge benchmarks⁵. The SMV instances were translated into DIMACS format for various bound values $k = \{20, 40, 60, \dots, 4000, 6000\}$ [20]. Each instance includes an LTL property provided with the model (46% of Safety property, 30% Guarantee, 14% Persistence, and 10% Recurrence according to the hierarchy of Manna&Pnueli [24]).

⁵ <https://tinyurl.com/29a4jcme>



■ **Figure 3** Measures of learnt clauses usage during propagation (left) and conflict analysis (right) phases. Blue dots denote LBD while red points depict the Pareto front of H_{LP} strategy.

To perform our analysis, we developed a tool called BMC-tool⁶ that integrates NuSMV tool [7] as a front-end and MapleCOMSPS [23] SAT-solver as a back-end. This solver is the winner of the main track of the SAT competition 2016 and was used as core engine for the best solvers in the last 5 years. The success of this solver relies on the management of the learnt clauses with three different databases according to the LBD value of the clauses: **core** ($LBD \leq 3$) for the really important ones (never deleted), **tier-2** ($LBD \leq 6$) for not-yet-decided clauses, and **local** database for the remaining clauses. Clauses in **tier-2** can be promoted to the core database or downgraded to local database while those of **local** database can either be promoted to **tier-2** or permanently deleted.

We run our tool on 25% of the whole benchmark (100 instances), the so-called *training benchmark*. For all instances, we logged the information related to each learnt clauses when used in the *unit propagation* and *conflict analysis*. These information are the LBD of the clause and its class (C_X). The results are depicted in Figure 2.

The x-axis reports the cumulative mean percentage of learnt clauses for the *training benchmark* and the y-axis corresponds to the cumulative mean usage percentage (left-side for *unit propagation* and right-side for *conflict analysis*). Each point represents the used percentage of learnt clauses of a certain LBD (from 1 to 10) for a certain class. For example, the purple triangle with left annotation 4 shows 2.5% of learnt clauses of class \mathcal{PM} have an $LBD \leq 4$ and are used in 13% of the *unit propagation* time (resp. 16% on *conflict analysis*).

We observe that C_P have a significant usage (around 45%) with a total coverage of around 5% in both *propagation* and *conflict analysis*. Therefore, these clauses seem to be good candidates for being considered as a relevant information.

Consider now Figure 3 while ignoring momentarily the red points. This figure depicts the same information as Figure 2 but without clause classification. Here, we observe that the default strategy for characterizing relevant information in MapleCOMSPS, i.e., $LBD \leq 3$ (identified by the blue point) covers 75% (mean value between *propagation* and *conflict analysis* curves) of utilization for a total of 11% of the learnt clauses. It appears then that more than half of what is considered as a relevant information came from C_P .

This measure comforts our thoughts that the performances of the SAT-solver are conditioned by a certain class of clauses. Our fine grained classification reveals that property clauses seem to be the more pertinent ones.

⁶ <https://gitlab.lrde.epita.fr/akheireddine/bmctool>

5 Heuristics for BMC

Based on the previous study, we present our ideas for improving the solving of SAT-based BMC problem. Our proposal is to identify and protect (from deletion) new sets of clauses that are relevant for the solving of a BMC problem. We introduce for this, two new heuristics:

Structural heuristic (H_S). The intuition behind this heuristic is to encourage the solver to focus on C_P clauses (probably these are used to falsify the property). To achieve this, we augment the **core** database of MapleCOMSPS by a subset of C_P : we take all clauses of C_P that have an $LBD \leq 5$. Indeed, after this threshold, the curves of Figure 2 seem to initiate an inflection that suggests that no more relevant information is captured.

Linear programming heuristic (H_{LP}). This approach aims at predicting the usefulness of each learnt clause mathematically. It determines the adequate LBD value for each class of clauses by maximizing the total usage of learnt clauses while minimizing their number. This is achieved by solving a linear programming system, that will provide multiple solutions specifying the suitable value of LBD for each class.

The linear program is written such that the objective is an aggregation function of the two above criteria. The constraints restrict the search-space to select at most one LBD value per class of clauses (input information is captured from Figure 2). The description of the linear system needs the introduction of the following notations:

- u_i^j : the percentage of learnt clauses (x-axis) with $LBD \leq i$ of class j .
- v_i^j : the percentage usage of learnt clauses (y-axis) with $LBD \leq i$ of class j .
- x_i^j : a Boolean variable representing the decision variable of the linear system. It takes the value 1 if the $LBD \leq i$ is chosen for the class of clauses j , 0 if not.
- $C = \{\mathcal{P}, \mathcal{M}, \mathcal{J}, \mathcal{PM}, \mathcal{PJ}, \mathcal{MJ}, \mathcal{PMJ}\}$ denotes the set of classes.

Hence, our modeling of the optimisation problem is as follows:

$$\begin{aligned} \text{maximize } \mathbf{f}_\mu &= -\mu \overbrace{\sum_{i=1}^{10} \sum_{j \in C} u_i^j x_i^j}^{\mathbf{O}_1} + (1 - \mu) \overbrace{\sum_{i=1}^{10} \sum_{j \in C} v_i^j x_i^j}^{\mathbf{O}_2} \\ \text{subject to } \left\{ \begin{array}{ll} \sum_{i=1}^{10} x_i^j \leq 1 & \forall j \in C \quad // \text{At most one LBD value per class} \\ x_i^j \in \{0, 1\} & \forall i \in \llbracket 1; 10 \rrbracket, \forall j \in C \end{array} \right. \end{aligned}$$

\mathbf{f}_μ is the aggregation function, defined as a weighted sum⁷, and parameterized with μ ($0 \leq \mu \leq 1$): the term \mathbf{O}_1 represents the number of learnt clauses that should be minimized and the term \mathbf{O}_2 is the used percentage that should be maximized. Then, this bi-objective optimization problem is converted to a single maximization problem using the parameter μ as described above. Solving this system (using data collected on the training benchmark) with various values for μ allows to draw a Pareto front (possibly not optimal).

The Pareto front highlighted in Figure 3 (red points) is obtained by solving this system using an increment of 0.01 for the parameter μ : each of these points corresponds to a configuration of the form: class C_P with LBD x , class C_J with LBD y , etc.

Our first observation is that the red points dominate the blue ones (representing the LBD-based approach of MapleCOMSPS) on both graphics of Figure 3. It appears then that we can improve the performance of the standard approach by choosing one of this

⁷ Other aggregation functions can be used, for example: Ordered Weighted Average, Choquet integral,...

point as a basis for detecting new relevant information: red points located between blue points tagged 3 and 4 (i.e., those with $LBD \leq 3$ and $LBD \leq 4$, respectively) are the best candidates. They are located at the inflection on both *propagation* and *conflict analysis* curves. Among these points, we found that the best promising one covers 83% on *unit propagation* (resp. 81% on *conflict analysis*) for a total of 15% of learnt clauses. This point characterizes the clauses with the following properties: $LBD \leq 3$ for all classes but C_P and C_J . These latter have the configurations $LBD \leq 4$ and $LBD \leq 9$, respectively. Therefore, this confirms the usefulness of clauses with $LBD \leq 3$ but also identifies new interesting ones.

6 Experimental results

All the experiments have been executed on the full benchmark presented in Section 4 on an Intel Xeon@2.40GHz machine with 12 processors and 64 Go of memory and a time limit of 6000 seconds⁸. Table 1 details the results of our experiments using MapleCOMSPS with H_S or H_{LP} heuristics. The table displays, the number of UNSAT and SAT solved instances, the total number of solved instances, the PAR-2 metric⁹ used in SAT competitions, the CTI metric¹⁰ and the cumulated time. H_S and H_{LP} don't include pre-processing time (took 44h27) and the Pareto front computation in H_{LP} doesn't take more than one second.

We observe that MapleCOMSPS solves 289 instances with a PAR-2 of 423h58. Besides, augmenting the **core** database to protect learnt clauses with $LBD \leq 4$ (MapleCOMSPS- $LBD \leq 4$) seems to deteriorate the performances: 2 instances less with a PAR-2 of 429h33 (5 hours slower than the original solver). This result shows that increasing the number of relevant clauses based entirely on the LBD cannot bring better performances.

The two next lines display the results of our heuristics. It appears that both of these strategies perform better than state-of-the-art: MapleCOMSPS- H_S solves 1 UNSAT and 2 SAT more while MapleCOMSPS- H_{LP} solves 4 UNSAT and 2 SAT more. The PAR-2 of these two heuristics shows a significant improvement with a gain of (at least) 6 hours.

Thus, the two presented heuristics demonstrate the importance of the information captured by C_P , since it is used by both of them: when performing the model-checking approach, a synchronous product between the Kripke structure and the (automaton of the) property is executed. Forcing the SAT procedure to consider property clauses will eliminate invalid paths in the property automaton, leading to a smaller synchronized product, i.e the state-space size is reduced efficiently. Also, it appears that H_{LP} captures another important information with C_J clauses: it is composed of fresh variables that make the connection between the property and the model. Consequently, they also help to compute information related to the synchronous product.

7 Conclusion and future work

Our journey towards building new heuristics for SAT procedures started with the observation that the relevant information used by SAT-solvers can be refined. We proposed a generic methodology to classify learnt clauses and we applied it to the special case of BMC. These learnt clauses have been classified according to their meaning in the original problem which

⁸ For a description of our setup, detailed results and code, see <https://akheiredine.github.io/>

⁹ PAR-k is the penalised average runtime, counting each timeout as k times the running time cutoff.

¹⁰ Cumulated execution Time of the Intersection for instances solved by all solvers

■ **Table 1** Comparison between state-of-the-art MapleCOMSPS solver and H_S and H_{LP} heuristics. MapleCOMSPS-LBD ≤ 4 uses a strategy where learnt clauses with LBD ≤ 4 are considered as relevant.

Solver	UNSAT	SAT	TOTAL	PAR-2	CTI (279)	Cumulated time
MapleCOMSPS	173	116	289	423h58	44h08	238h59
MapleCOMSPS-LBD ≤ 4	169	118	287	429h33	43h12	241h13
MapleCOMSPS- H_S	174	118	292	418h10	43h24	238h10
MapleCOMSPS- H_{LP}	177	118	295	413h53	45h02	238h53

helped us to suggest two heuristics (H_S and H_{LP}) based on the information carried by the LTL property. The two heuristics improve the state-of-the-art approach, with the particularity of H_S to have a structural reasoning behind. In the other hand, the procedure used to build H_{LP} relies on a mathematical reasoning.

Future work aims to refine the proposed classification by exploiting the specification of the property or the synchronicity of the model. Moreover, we would like to propagate this idea to offer new sharing strategies on parallel SAT-solvers. And finally, building a SAT-solver to exploit exclusively structural information of the original problem is in our perspectives.

References

- 1 F.A. Aloul, K.A. Sakallah, and I.L. Markov. Efficient symmetry breaking for boolean satisfiability. *IEEE Transactions on Computers*, 55(5):549–558, 2006. doi:10.1109/TC.2006.75.
- 2 Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The community structure of sat formulas. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 410–423, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- 3 Carlos Ansótegui, Maria Luisa Bonet, Jesús Giráldez-Cru, Jordi Levy, and Laurent Simon. Community structure in industrial sat instances, 2019. arXiv:1606.03329.
- 4 Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.
- 5 Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking, December 2003. doi:10.1016/S0065-2458(03)58003-2.
- 6 J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142–170, 1992. doi:10.1016/0890-5401(92)90017-A.
- 7 A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.
- 8 E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, pages 419–422, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
- 9 Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, 2001. doi:10.1023/A:1011276507260.
- 10 Edmund Clarke, E. Emerson, and Joseph Sifakis. Model checking. *Communications of the ACM*, 52, November 2009. doi:10.1145/1592761.1592781.
- 11 James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, KR’96, page 148–159, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

- 12 Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962. doi:10.1145/368273.368557.
- 13 Daniel Frost and Rina Dechter. Dead-end driven learning. *Proceedings of the National Conference on Artificial Intelligence*, 1, August 2000.
- 14 Malay Ganai and Aarti Gupta. Tunneling and slicing: Towards scalable bmc. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, page 137–142, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1391469.1391507.
- 15 Malay Ganai, Aarti Gupta, Zijiang Yang, and Pranav Ashar. Efficient distributed sat and sat-based distributed bounded model checking. *International Journal on Software Tools for Technology Transfer*, 8:387–396, August 2006. doi:10.1007/s10009-005-0203-z.
- 16 Malay K. Ganai. Sat-based scalable formal verification solutions. In *Series on Integrated Circuits and Systems*, Springer-Verlag New York, 2007.
- 17 Malay K. Ganai. Propelling SAT and sat-based BMC using careset. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 231–238. IEEE, 2010. URL: <http://ieeexplore.ieee.org/document/5770954/>.
- 18 Matthew L. Ginsberg and David A. McAllester. Gsat and dynamic backtracking. In Alan Born-ing, editor, *PPCP*, volume 874 of *Lecture Notes in Computer Science*, pages 243–265. Springer, 1994. URL: <http://dblp.uni-trier.de/db/conf/ppcp/ppcp94-lncs.html#GinsbergM94>.
- 19 Gerard J. Holzmann. Explicit-state model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 153–171, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-10575-8_5.
- 20 Paul Jackson and Daniel Sheridan. Clause form conversions for boolean circuits. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing*, pages 183–198, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- 21 Sima Jamali and David Mitchell. Centrality-based improvements to cdel heuristics. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 122–131, Cham, 2018. Springer International Publishing.
- 22 George Katsirelos and Laurent Simon. Eigenvector centrality in industrial sat instances. In Michela Milano, editor, *Principles and Practice of Constraint Programming*, pages 348–356, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- 23 J. Liang, Vijay Ganesh, P. Poupart, and K. Czarnecki. Learning rate based branching heuristic for sat solvers. In *SAT*, 2016.
- 24 Z. Manna and A. Pnueli. A hierarchy of temporal properties (invited paper, 1989). In *PODC '90*, 1990.
- 25 Hakan Metin, Souheib Baarir, Maximilien Colange, and Fabrice Kordon. Cdclsym: Introducing effective symmetry breaking in sat solving. In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18)*, volume 10805 of *Lecture Notes in Computer Science*, pages 99–114, Thessaloniki, Greece, 2018. Springer.
- 26 Radek Pelánek. Beem: Benchmarks for explicit model checkers. In Dragan Bošnački and Stefan Edelkamp, editors, *Model Checking Software*, pages 263–267, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- 27 Kristin Y. Rozier. Survey: Linear temporal logic symbolic model checking. *Comput. Sci. Rev.*, 5(2):163–203, 2011. doi:10.1016/j.cosrev.2010.06.002.
- 28 Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, page 216–226, New York, NY, USA, 1978. Association for Computing Machinery. doi:10.1145/800133.804350.
- 29 Ofer Shtrichman. Tuning sat checkers for bounded model checking. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 480–494, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

- 30 João P. Marques Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '96, page 220–227, USA, 1997. IEEE Computer Society.
- 31 Laurent Simon and Gilles Audemard. Predicting Learnt Clauses Quality in Modern SAT Solver. In *Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*, Pasadena, United States, 2009. URL: <https://hal.inria.fr/inria-00433805>.
- 32 Chao Wang, HoonSang Jin, Gary D. Hachtel, and Fabio Somenzi. Refining the sat decision ordering for bounded model checking. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, page 535–538, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/996566.996713.
- 33 Emmanuel Zarpas. Simple yet efficient improvements of sat based bounded model checking. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design*, pages 174–185, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- 34 Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '01, page 279–285. IEEE Press, 2001.

Integrating Tree Decompositions into Decision Heuristics of Propositional Model Counters

Tuukka Korhonen   

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

Matti Järvisalo   

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

Abstract

Propositional model counting ($\#SAT$), the problem of determining the number of satisfying assignments of a propositional formula, is the archetypical $\#P$ -complete problem with a wide range of applications in AI. In this paper, we show that integrating tree decompositions of low width into the decision heuristics of a reference exact model counter (SharpSAT) significantly improves its runtime performance. In particular, our modifications to SharpSAT (and its derivant GANAK) lift the runtime efficiency of SharpSAT to the extent that it outperforms state-of-the-art exact model counters, including earlier-developed model counters that exploit tree decompositions.

2012 ACM Subject Classification Theory of computation \rightarrow Constraint and logic programming

Keywords and phrases propositional model counting, decision heuristics, tree decompositions, empirical evaluation

Digital Object Identifier 10.4230/LIPIcs.CP.2021.8

Category Short Paper

Supplementary Material *Software (Source code and experimental data):* <https://github.com/Laakeri/modelcounting-cp21>

archived at `swb:1:dir:fd1a6eaa9d3ba301b7151f077f51e1da29801ffe`

Funding Work financially supported by Academy of Finland under grants 322869 and 328718.

1 Introduction

Propositional model counting ($\#SAT$), the problem of determining the number of satisfying assignments of a propositional formula, is the archetypical $\#P$ -complete problem [34]. Improving the scalability of state-of-the-art model counters is a challenging task, motivated by a wide range of applications in AI, including probabilistic reasoning, planning, quantified information flow analysis, differential cryptanalysis, and model checking [29, 5, 25, 20, 2].

Many current exact model counters rely heavily on search techniques adapted from Boolean satisfiability (SAT) solving and employ component caching to avoid repeatedly counting over the same residual formulas seen during the counting process. In particular, these techniques are applied both by “search-based” exact model counters (such as Cachet, SharpSAT and GANAK [28, 33, 30]) and “compilation-based” counters (such as c2d, minic2d, and D4 [7, 24, 21]) in which the compilation process is based on SAT solver traces. Hence improvements to decision heuristics in the underlying model counters have the promise of speeding up various state-of-the-art model counters.

In this work, we propose and evaluate the effects of integrating information on tree decompositions of CNF formulas to guide the decision heuristics in search-based exact propositional model counters. In theory, it is known that $\#SAT$ can be solved in time $poly(|\phi|)2^w$, where $|\phi|$ is the size of the formula and w the width of a given tree decomposition of the primal graph of the formula ϕ . If clause learning is not employed, search-based counters



© Tuukka Korhonen and Matti Järvisalo;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 8; pp. 8:1–8:11

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

achieve this time complexity if they employ component caching and a variable selection algorithm based on the tree decomposition [1, 6, 9]. Tree decompositions have recently been employed in dynamic programming based model counters [12, 15, 17], and recent exact model counters have adapted alternative graph-based techniques, including heuristic graph partitioning algorithms [8, 21, 24] and graph centrality measures [3], for deciding variable orderings and decision heuristics. (For more discussion, see section on Related Work.) However, we are not aware of earlier work on integrating tree decompositions directly as a decision heuristic component in the context of search-based propositional model counters.

In this paper, we show that, in practice, exploiting tree decompositions of low width is easy and effective in speeding up state-of-the-art search-based exact model counters SharpSAT and GANAK on instances with treewidth as high as 150 (or even higher). In particular, motivating the approach through theoretical observations, we describe how to integrate tree decomposition guidance to the decision heuristics of these model counters. We show through extensive empirical evaluation that the tree decomposition guided modifications of SharpSAT and GANAK noticeably outperform other state-of-the-art exact model counters, including the counters themselves in their default settings. Beyond the empirical evidence provided in this paper, we note that our SharpSAT-based model counter **SharpSAT-TD**, implementing the ideas presented in this work, ranked first in tracks 1, 2, and 4 of Model Counting Competition 2021 (see <https://mccompetition.org/>).

2 Preliminaries

We consider the problem of counting the number of satisfying truth assignments (or models) of a conjunctive normal form (CNF) propositional formula, i.e., #SAT. A CNF formula is denoted by ϕ , its variables by $V(\phi)$, clauses by $cls(\phi)$, and variables of a clause c by $V(c)$. The size of a formula ϕ is $|\phi| = |V(\phi)| + |cls(\phi)|$. We denote by $\phi_{|x=1}$ the formula obtained from ϕ by assigning a variable $x \in V(\phi)$ to 1 (true), i.e., the formula ϕ with x removed from the variable set, each clause containing literal x removed, and each occurrence of $\neg x$ in any clause removed. The formula $\phi_{|x=0}$ is defined analogously. The formula obtained by applying unit propagation, i.e., setting $\phi \leftarrow \phi_{|x=0}$ whenever there is a clause $(\neg x)$ and $\phi \leftarrow \phi_{|x=1}$ whenever there is a clause (x) , is denoted by $UP(\phi)$. The number of models of ϕ is $\#(\phi)$. For any variable x it holds that $\#(\phi) = \#(\phi_{|x=0}) + \#(\phi_{|x=1})$. Note also that $\#(\phi) = \#(UP(\phi))$. We denote the union of two formulas ϕ_1 and ϕ_2 with disjoint variable sets by $\phi_1 \sqcup \phi_2$. The fact that $\#(\phi_1 \sqcup \phi_2) = \#(\phi_1) \cdot \#(\phi_2)$ allows for separately counting the number of models in the variable-disjoint formulas ϕ_1 and ϕ_2 to obtain the model count of $\phi_1 \sqcup \phi_2$ [19].

We consider tree decompositions of primal graphs of CNF formulas (aka Gaifman graphs). A graph G has a set of vertices $V(G)$ and a set of edges $E(G)$. For a vertex set $X \subseteq V(G)$ we denote by X^2 the set of all possible edges within X . The primal graph $G(\phi)$ of a formula ϕ is a graph with $V(G(\phi)) = V(\phi)$ and $E(G(\phi)) = \bigcup_{c \in cls(\phi)} V(c)^2$. In words, the vertices of the primal graph are the variables and the edges are created by inducing a clique on the variables of each clause.

► **Example 1.** Consider the CNF formula ϕ with variables $V(\phi) = \{x_1, \dots, x_6\}$ and clauses $cls(\phi)$ as shown in Figure 1 (left). The primal graph $G(\phi)$ is in Figure 1 (middle). The vertices of $G(\phi)$ are the variables of ϕ and the edges of $G(\phi)$ are defined by the clauses of ϕ . For example, $G(\phi)$ contains the edge $\{x_1, x_2\}$ because ϕ contains the clause $(x_1 \vee \neg x_2 \vee x_5)$.

A tree is a connected graph T with $|E(T)| = |V(T)| - 1$. A tree decomposition [26, 4] of a graph G is a tree T whose each node t corresponds to a bag $T[t] \subseteq V(G)$ containing vertices of G and which satisfies the properties

1. $V(G) \subseteq \bigcup_{t \in V(T)} T[t]$,
2. $E(G) \subseteq \bigcup_{t \in V(T)} T[t]^2$, and
3. for each $v \in V(G)$, the nodes $\{t \in V(T) \mid v \in T[t]\}$ form a connected subtree of T .

The width of a tree decomposition T is $w(T) = \max_{t \in V(T)} |T[t]| - 1$, and the treewidth of a graph G is the minimum width over all tree decompositions of G . We use the convention that one of the nodes of the tree decomposition is chosen as the root of the tree decomposition. The root can be chosen arbitrarily. We denote by $d_T(t)$ the distance from the root to the node t in the tree decomposition T , i.e., the depth of the node t .

► **Example 2.** Consider the CNF formula ϕ with variables $V(\phi) = \{x_1, \dots, x_6\}$ and clauses $cls(\phi)$ as shown in Figure 1 (left). The primal graph $G(\phi)$ is shown in Figure 1 (middle), and a tree decomposition T of $G(\phi)$ in Figure 1 (right). The bags of T are $\{x_2, x_3, x_5\}$, $\{x_1, x_2, x_5\}$, $\{x_3, x_5, x_6\}$, and $\{x_1, x_4\}$. The width of T is 2 because the largest bag has size 3, and thus the treewidth of $G(\phi)$ is at most 2. Let t_1 denote the node of T with the bag $T[t_1] = \{x_2, x_3, x_5\}$ and t_2 the node with the bag $T[t_2] = \{x_1, x_4\}$. If t_1 is the root, then $d_T(t_1) = 0$ and $d_T(t_2) = 2$.

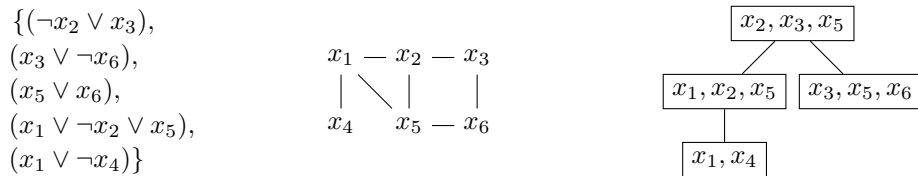
3 Tree Decomposition Guided Model Counting

Consider the basic DPLL-style algorithm with component caching for model counting [1] presented as Algorithm 1, consisting of unit propagation (Line 1), detection of disconnected components (Line 4), component caching (cache check on Line 6, caching on Line 9), and making decisions by selecting and assigning currently unassigned variables (Line 7).

Our focus in this work is on the decision heuristics, i.e., implementation of Line 7. Algorithm 2 specifies the tree decomposition guided variable selection algorithm. By using Algorithm 2 as the variable selection procedure in Algorithm 1, we obtain a DPLL-style tree decomposition guided model counter.

► **Example 3.** Consider the run of Algorithm 1 on the formula ϕ of Figure 1 (left) using Algorithm 2 with the tree decomposition T of Figure 1 (right), rooted on the node t_1 with the bag $T[t_1] = \{x_2, x_3, x_5\}$. In the first recursive call, the variable selected is x_2 because it is the lowest index variable in the bag of the root node. Consider a recursive call after variable decisions $x_2 = 1$, $x_3 = 1$ by unit propagation, and $x_5 = 1$. The remaining formula has variables x_1, x_4 , and x_6 , and only the clause $(x_1 \vee \neg x_4)$. On Line 4 it is partitioned to two formulas, one with variable set $\{x_1, x_4\}$, and one with variable set $\{x_6\}$. On a recursive call on the former formula, the variable x_1 is selected by Algorithm 2, because it is the only variable left in the lowest-depth bag $\{x_1, x_2, x_5\}$ intersecting the variable set of the formula.

The time complexity of Algorithm 1 equipped with Algorithm 2 for variable selection is $\text{poly}(|\phi|)2^{w(T)}$, where T is the tree decomposition given as input. This time complexity and similar observations have been already made earlier [1, 6, 9, 27].



■ **Figure 1** An example formula (left), its primal graph (middle), and one of tree decompositions of the primal graph (right).

■ **Algorithm 1** DPLL-style model counter.

Input : Formula ϕ
Output : The number of satisfying assignments of ϕ

```

1  $\phi \leftarrow \text{UP}(\phi)$ 
2 if  $\emptyset \in \text{cls}(\phi)$  then return 0
3 if  $V(\phi) = \emptyset$  then return 1
4 if  $\phi = \phi_1 \sqcup \phi_2$  then
5   | return  $\text{Count}(\phi_1) \cdot \text{Count}(\phi_2)$ 
6 if  $\phi$  in cache then return cache[ $\phi$ ]
7  $x \leftarrow \text{VariableSelect}(\phi)$ 
8  $R \leftarrow \text{Count}(\phi|_{x=0}) + \text{Count}(\phi|_{x=1})$ 
9 cache[ $\phi$ ]  $\leftarrow R$ 
10 return  $R$ 
```

■ **Algorithm 2** Tree decomposition guided variable selection.

Input : Formula ϕ and tree decomposition T of $G(\phi)$
Output : Variable $x \in V(\phi)$

```

1  $t \leftarrow$  The lowest depth node of  $T$  with  $|T[t] \cap V(\phi)| \geq 1$ 
2 return The variable in  $T[t] \cap V(\phi)$  with the lowest index
```

► **Proposition 4** ([6]). *If Algorithm 1 implements the variable selection of Algorithm 2, then the number of cache entries created during Algorithm 1 is at most $|V(T)|(w(T) + 1)2^{w(T)}$.*

Proof. Suppose that the execution of Algorithm 1 is at Line 7. We show that there can be at most $2^{w(T)}$ different formulas ϕ for a fixed node t of T determined on Line 1 of Algorithm 2 and a fixed variable x returned by Algorithm 2. This implies the proposition because there are at most $|V(T)|$ choices for t and at most $(w(T) + 1)$ choices for x .

Let p be the parent node of t in T . The formula ϕ can be obtained from the original input formula by assigning all variables in $T[p] \cap T[t]$ and the variables in $T[t]$ with lower index than x , then applying unit propagation, and then selecting the component containing x . There are at most $w(T)$ such variables, so the number of choices is $2^{w(T)}$. ◀

As each recursive call of Algorithm 1 is polynomial-time, time complexity $\text{poly}(|\phi|)2^{w(T)}$ follows from Proposition 4. Although Proposition 4 does not necessarily hold when equipping Algorithm 1 with clause learning, we will show that tree decomposition guidance provides significant performance improvements in practice also when clause learning is employed.

4 Integrating Tree Decompositions into Model Counters

In SharpSAT [33] and GANAK [30] (a SharpSAT derivative), variable selection is based on variable scores, maintained as an array `score` mapping variables to floating point numbers. The variable selection algorithm works by selecting the variable x with the highest `score`(x). The score of each variable is based on two components: it is the sum of the frequency score of the variable and the activity score of the variable. The frequency score is the number of occurrences of the variable in the current formula, and an activity score similar to VSIDS in SAT solvers [23]. The resulting heuristic, `score`(x) = `act`(x) + `freq`(x), with both frequency and activity is called VSADS. Further, GANAK makes use of another score

called CacheScore for prioritizing variables whose components were not recently added to the cache. The resulting heuristic is called CSVSADS. We implement tree decomposition based variable selection by modifying the `score` array in both SharpSAT and GANAK. In principle, implementing tree decomposition based variable selection with the `score` array amounts to just setting the score of a variable x to $-\min_{\{t|x \in T[t]\}} d_T(t)$, where $d_T(t)$ is the distance from the root of T to the node t . However, as we show in our experiments it is sometimes beneficial to use hybrid scores, even though the theoretical bound will not hold in that case. In particular, we propose the following integration of tree decomposition guidance as a modification of VSADS into both SharpSAT and GANAK:

$$\text{score}(x) = \text{act}(x) + \text{freq}(x) - C \min_{\{t|x \in T[t]\}} d_T(t) \quad (1)$$

where C is a per-instance chosen positive constant and $d_T(t)$ is normalized to take values between 0 and 1. As default we use $C = 100 \exp(n/w)/n$, where n is the number of variables and w the width of the tree decomposition. We empirically justify this choice in Section 5.

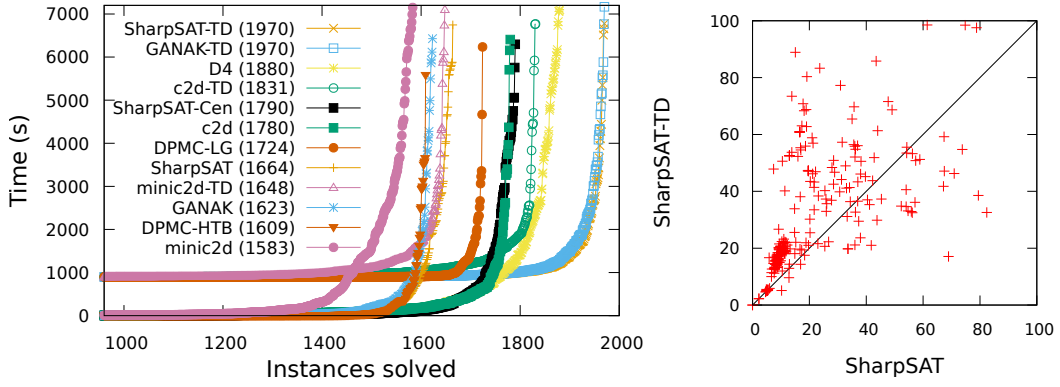
For computing tree decompositions of low width in practice, we use FlowCutter [16, 32] FlowCutter was ranked second in the 2nd Parameterized Algorithms and Computational Experiments Challenge (PACE 2017) heuristic treewidth track, and was observed to outperform the winning implementation on large graphs [10]. It is also used in the recent DPMC model counter [12]. FlowCutter is an anytime algorithm, meaning that we can terminate it anytime to get the best tree decomposition computed thus far. As the root of the tree decomposition we choose a centroid node, i.e., a node t such that each component of $G(\phi) \setminus T[t]$ has at most $|V(G(\phi))|/2$ vertices. Before computing the tree decomposition we preprocess the formula with the standard techniques of unit propagation and failed literal elimination.

Finally, although not on the level of internal decision heuristics, we note that both c2d [7] and minic2d [24] can take as an input a structure to control the variable ordering inside the compiler. In particular, c2d can take a decision tree (dtree) as an input and minic2d a variable tree (vtree) as an input. Both of these structures can be constructed from a tree decomposition so that the variable selection algorithm of the compiler implements Algorithm 2. For empirically evaluating the impact of tree decompositions obtained with FlowCutter on c2d and minic2d, we construct both of these structures from a tree decomposition by placing all variables of the root bag to the top of the tree, and recursing to the subtrees.

5 Empirical Evaluation

We provide results from an extensive empirical evaluation, comparing the impact of integrating tree decomposition based heuristics on the runtime performance of SharpSAT, GANAK, c2d, and minic2d. We also compare to the extend possible the performance of these four model counters with tree decomposition heuristics to the performance of the recent model counters D4 [21], DPMC [12], gpusat [15] and NestHDB [17]; and SharpSAT with the recently proposed centrality-based heuristics [3]. DPMC has both a tensor and a decision diagram based implementation. We compare to the decision diagram based implementation, as it has been reported to perform significantly better [12]. The decision diagram based implementation has two versions, DPMC-LG which exploits tree decompositions and DPMC-HTB, and we compare to both of them. (DPMC-HTB is equivalent to ADDMC [11].)

As benchmarks we used 2424 instances from recent empirical evaluations of model counters. In particular, we merged an instance set of 1952 instances from <http://www.cril.univ-artois.fr/KC/benchmarks.html> used in e.g. [21, 3, 12, 14] with an instance set of 1619 instances from <https://github.com/dfremont/counting-benchmarks> used



■ **Figure 2** Left: Empirical runtime comparison of different model counters. Right: average number of variables in component cache hits, SharpSAT vs SharpSAT-TD.

■ **Table 1** Pairwise comparison of original versions of SharpSAT, GANAK, c2d and minic2d against their tree decomposition guided versions.

Family	#Ins	VBS	VBS-O	VBS-TD	SharpSAT	SharpSAT-TD	GANAK	GANAK-TD	c2d	c2d-TD	minic2d	minic2d-TD
BN-Ratio	389	387	333	387	177	385	163	386	329	345	289	330
BN-DQMR	660	629	627	629	627	629	620	629	577	593	598	584
BN-Ace	31	22	22	22	22	22	22	22	22	22	19	16
BN-other	5	4	3	4	1	2	1	2	3	4	2	2
Plan recognition	11	11	11	11	11	11	11	11	10	11	10	11
Planning-pddl	529	458	451	447	417	446	405	446	426	428	415	390
Planning-other	17	16	16	16	15	16	15	16	13	13	11	13
Circuit-iscas	132	117	117	116	112	116	109	116	110	109	109	104
Circuit-other	17	10	10	10	10	10	10	10	9	9	10	9
BMC-symb-markov	130	118	112	118	52	118	53	114	94	108	20	20
BMC-other	18	14	13	11	12	11	12	11	7	8	3	7
Symbolic-sygyus	138	22	21	17	19	15	20	16	1	0	0	0
QIF-maxcount-qif	127	12	11	12	11	12	6	12	10	10	4	4
QIF-other	7	5	4	5	4	5	4	5	4	5	2	3
Handmade	68	36	36	36	36	34	35	36	31	33	33	35
Configuration	35	35	35	35	35	35	34	35	33	32	21	21
Random	104	103	103	103	103	103	103	103	101	101	37	99
Scheduling	6	0	0	0	0	0	0	0	0	0	0	0
Total	2424	1999	1925	1979	1664	1970	1623	1970	1780	1831	1583	1648

in e.g. [15, 14, 17], removing duplicates and instances found unsatisfiable using a SAT solver. The benchmark set divides into 18 families from applications in e.g. probabilistic reasoning, planning, model checking, synthesis [29, 25, 21]. The experiments were run single-threaded on computers with 2.6-GHz Intel Xeon E5-2670 processors. A time limit of 2 hours and memory limit of 16 GB was used. Please consult <https://github.com/Laakeri/modelcounting-cp21> for source code and detailed data.

Figure 2(left) overviews the relative performance of the model counters (apart from gpusat and NestHDB). SharpSAT and GANAK using the tree decomposition heuristics (*-TD) solved the greatest number of instances (1970), resulting in state-of-the-art performance over all the considered counters. After SharpSAT-TD and GANAK-TD, the best-performing counters are D4, c2d-TD (i.e., the tree decomposition guided c2d), and SharpSAT using centrality-based heuristics, solving 1880, 1831, and 1790 instances, respectively. Note that here we allowed a fixed 900 seconds for tree decomposition computation using FlowCutter on each instance for SharpSAT-TD, GANAK-TD, c2d-TD, and minic2d-TD (as well as DPMC-LG; see Related Work section). This 900-second runtime is included in the results, as can be clearly seen in Figure 2(left). However, using this relatively high number of seconds is not necessary: when using 5, 60, 900, and 1800 seconds, respectively, the numbers of

■ **Table 2** Pairwise comparison grouped by width of tree decompositions used by SharpSAT-TD.

Width	#Ins	VBS	VBS-O	VBS-TD	SharpSAT	SharpSAT-TD	GANAK	GANAK-TD	c2d	c2d-TD	minic2d	minic2d-TD
≤ 20	810	810	809	810	798	810	791	810	809	810	809	810
21...30	526	525	509	525	405	524	385	524	467	489	461	483
31...50	378	307	286	303	173	302	164	302	254	266	165	185
51...100	259	164	131	155	101	152	95	153	106	117	60	71
101...150	57	27	26	27	25	26	25	27	18	23	8	13
151...200	128	115	114	115	114	115	114	115	112	110	44	108
201...300	43	31	31	27	31	26	31	26	11	13	11	7
301 \leq	223	20	19	17	17	15	18	13	3	3	3	3
Total	2424	1999	1925	1979	1664	1970	1623	1970	1780	1831	1583	1648

■ **Table 3** Comparison of gpusat, NestHDB, and SharpSAT-TD, grouped by width of the tree decomposition used by SharpSAT-TD.

Width	#Ins	VBS	gpusat	NestHDB	SharpSAT-TD
≤ 30	1232	1232	1232	1232	1232
31...50	21	14	1	10	14
51...100	15	10	0	7	9
101...200	18	16	0	16	16
201...266	21	11	0	8	10
267 \leq	187	0	0	0	0
Total	1494	1283	1233	1273	1281

instances solved by SharpSAT-TD are, respectively, 1962, 1971, 1970, and 1967. In particular, using the much lower time limit of 5 seconds would result in very much the same overall performance for SharpSAT-TD.

Table 1 gives a per benchmark family comparison of the impact of tree decomposition based heuristics on the number of instances solved by SharpSAT, GANAK, c2d and minic2d. SharpSAT-TD improves significantly on SharpSAT (1970 vs 1664 solved), and similarly GANAK-TD improves significantly on GANAK (1970 vs 1623). Furthermore, SharpSAT-TD and GANAK-TD solve only 9 instances less than the virtual best solver VBS-TD, which is considered to solve an instance if at least one of SharpSAT-TD, GANAK-TD, c2d-TD, and minic2d-TD solves the instance. VBS-TD also outperforms the virtual best solver VBS-O over the original four model counters which evidently are more different from each other than their modifications, each using the same tree decomposition to guide the counting process; Indeed, the difference between VBS-O and the best original model counter is 145 instances, in contrast to the difference of 9 instance between VBS-TD and SharpSAT-TD.

The number of instances solved, with instances grouped by the width of the tree decomposition found with FlowCutter in 900 seconds, is shown in Table 2. We observe to a great extent consistent performance improvement for each of the four model counters up to width 150 and at times even up to width 200. For instances of width ≤ 20 , SharpSAT-TD, GANAK-TD, c2d-TD, and minic2d-TD each solve all instances, while the original SharpSAT, GANAK, c2d, and minic2d each fail to solve some instances.

Due to the techniques gpusat and NestHDB implement – gpusat relies on certain GPU hardware, and NestHDB relies on a database management system – we were unable to run them ourselves. Hence we are forced to resort to comparing our runtimes with the empirical results provided for gpusat and NestHDB in their respective papers [15, 17] using the benchmark instances used therein. For this indirect comparison, following [17], we enforced a per-instance time limit of 900 s, memory limit of 16 GB, and tree decomposition computation time limit of 60 s on SharpSAT-TD. Table 3 provides the indirect comparison with instances grouped by the width of the tree decomposition used by SharpSAT-TD. On this set of 1494 instances, gpusat solves 1233 instances and NestHDB solves 1273, while SharpSAT-TD solves 1281 instances. Note that in [17] NestHDB was found to be the best

against a range of other model counters on these benchmarks, and minic2d second-best solving 1243 instances. Here SharpSAT-TD outperforms gpusat and NestHDB on all ranges of width apart from ≤ 30 and $[101..200]$, where it solves the same instances as VBS.

Finally, we shortly overview further observation on the impact of the tree decomposition based heuristics in SharpSAT. We considered modifications of the variable selection heuristics (Equation 1) for SharpSAT-TD. Recall that SharpSAT-TD solved 1970 instances using the heuristic with default activity and frequency components and C determined as $C = 100 \exp(n/w)/n$. When selecting C as 10^3 , 10^7 , and $100 \exp(n/w)$, SharpSAT-TD solves 1922, 1964, and 1960 instances, respectively. We note that the choice 10^7 leads to the tree decomposition based component always dominating in the equation, with activity and frequency serving only as tiebreakers. When $C = 100 \exp(n/w)/n$ and the activity component is removed, SharpSAT-TD solves 1965 instances, while when the frequency component is removed SharpSAT-TD solves 1962 instances. Hence the impact of each of these two components on their own, when including the tree decomposition component, is relatively small. However, when both the activity and the frequency component are removed, SharpSAT-TD solves only 1855 instances. Putting all of these observations together, we believe that the activity and frequency components act mainly as a secondary tiebreaking mechanism for choosing between variables in the same bag of the decomposition. Furthermore, the impact of the choice between using activity vs frequency as the tiebreaking mechanism appears to be small, and the primary heuristic component leading to the observed performance improvements is indeed the tree decomposition component.

The tree decomposition based heuristics appears to have a positive impact on average cache hit size, i.e., the number of variables of the components found to be in cache during checks to the component cache. Intuitively, the larger the cache hits, the earlier SharpSAT can determine the number of models in the current search branch, thereby saving time due to the component cache. Figure 2 (right) shows average cache hit sizes reported by SharpSAT and SharpSAT-TD on instances which both of them solved using at least 60 seconds on search (267 instances). The tree decomposition guided variable selection increases average cache hit size for most of the instances. (We did not observe clear effects on cache hit rates; cache hit rates do not distinguish hits on small components from hits on large component.)

6 Related Work

The idea of exploiting low-width tree decompositions in model counters has recently gained popularity with the model counters gpusat [15], NestHDB [17] and DPMC-LG [12] explicitly exploiting low-width tree decompositions. In contrast to our work, gpusat, NestHDB, and the tensor implementation of DPMC-LG exploit tree decompositions by manipulating dense dynamic programming tables. The model counters gpusat and the tensor implementation of DPMC-LG are “pure” dynamic programming implementations that suffer from best-case time complexity exponential in treewidth, while NestHDB also incorporates hybrid techniques, including falling back to SharpSAT in subproblems with high treewidth. The decision diagram implementation of DPMC-LG uses tree decompositions via project join trees to build an algebraic decision diagram using the CUDD package [31].

In the context of #CSP, tree decompositions have been exploited in the #BTD [13] and #EBTD[18] backtracking algorithms. The method of exploiting tree decompositions in #BTD and #EBTD is similar to SharpSAT-TD when selecting a high value of the constant C , although many techniques exploited in these counters are CSP-specific.

Instead of tree decompositions, heuristic graph partitioning is used in compilation-based model counters: D4 uses the PaToH graph partitioner [21], c2d uses Hmetis [8], and minic2d uses the min-fill heuristic for variable ordering [24]. GANAK introduced a variable selection heuristic CSVSADS aiming to increase the cache hit rate by discouraging branching from variables whose components were recently cached [30]. In the context of constraint networks, heuristics aiming to promote decomposition into components have been evaluated in [22].

7 Conclusion

We proposed a simple approach for integrating tree decomposition guidance into the decision heuristics of exact model counters. As a decision heuristic, the approach is directly applicable to both unweighted and weighted model counting. The empirical results suggest that tree decomposition guided SharpSAT dominates in performance standard exact model counters. and provides significant performance improvements in practice.

References

- 1 Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *44th Symposium on Foundations of Computer Science, FOCS 2003*, pages 340–351. IEEE Computer Society, 2003. doi:10.1109/SFCS.2003.1238208.
- 2 Fabrizio Biondi, Michael A. Enescu, Annelie Heuser, Axel Legay, Kuldeep S. Meel, and Jean Quilbeuf. Scalable approximation of quantitative information flow in programs. In Isil Dillig and Jens Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation – 19th International Conference, VMCAI 2018*, volume 10747 of *Lecture Notes in Computer Science*, pages 71–93. Springer, 2018. doi:10.1007/978-3-319-73721-8_4.
- 3 Bernhard Bliem and Matti Järvisalo. Centrality heuristics for exact model counting. In *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019*, pages 59–63. IEEE, 2019. doi:10.1109/ICTAI.2019.00017.
- 4 Hans L. Bodlaender. Discovering treewidth. In Peter Vojtás, Mária Bielíková, Bernadette Charron-Bost, and Ondrej Šýkora, editors, *SOFSEM 2005: Theory and Practice of Computer Science, 31st Conference on Current Trends in Theory and Practice of Computer Science*, volume 3381 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005. doi:10.1007/978-3-540-30577-4_1.
- 5 Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799, 2008. doi:10.1016/j.artint.2007.11.002.
- 6 Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001. doi:10.1145/502090.502091.
- 7 Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI’2004*, pages 328–332. IOS Press, 2004.
- 8 Adnan Darwiche. The C2D compiler user manual. Technical Report D-147, UCLA Department of Computer Science, 2005.
- 9 Rina Dechter and Robert Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171(2-3):73–106, 2007. doi:10.1016/j.artint.2006.11.003.
- 10 Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The PACE 2017 parameterized algorithms and computational experiments challenge: The second iteration. In Daniel Lokshtanov and Naomi Nishimura, editors, *12th International Symposium on Parameterized and Exact Computation, IPEC 2017*, volume 89 of *LIPIcs*, pages 30:1–30:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.IPEC.2017.30.

- 11 Jeffrey M. Dudek, Vu Phan, and Moshe Y. Vardi. ADDMC: Weighted model counting with algebraic decision diagrams. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020*, pages 1468–1476. AAAI Press, 2020. URL: <https://aaai.org/ojs/index.php/AAAI/article/view/5505>.
- 12 Jeffrey M. Dudek, Vu H. N. Phan, and Moshe Y. Vardi. DPMC: Weighted model counting by dynamic programming on project-join trees. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming – 26th International Conference, CP 2020*, volume 12333 of *Lecture Notes in Computer Science*, pages 211–230. Springer, 2020. doi:10.1007/978-3-030-58475-7_13.
- 13 Aurélie Favier, Simon de Givry, and Philippe Jégou. Exploiting problem structure for solution counting. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming – CP 2009, 15th International Conference, CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 335–343. Springer, 2009. doi:10.1007/978-3-642-04244-7_27.
- 14 Johannes K. Fichte, Markus Hecher, and Florim Hamiti. The Model Counting Competition 2020. *CoRR*, abs/2012.01323, 2020. arXiv:2012.01323.
- 15 Johannes Klaus Fichte, Markus Hecher, and Markus Zisser. An improved GPU-based SAT model counter. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming – 25th International Conference, CP 2019*, volume 11802 of *Lecture Notes in Computer Science*, pages 491–509. Springer, 2019. doi:10.1007/978-3-030-30048-7_29.
- 16 Michael Hamann and Ben Strasser. Graph bisection with pareto optimization. *ACM Journal of Experimental Algorithmics*, 23, 2018. doi:10.1145/3173045.
- 17 Markus Hecher, Patrick Thier, and Stefan Woltran. Taming high treewidth with abstraction, nested dynamic programming, and database technology. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020 – 23rd International Conference*, volume 12178 of *Lecture Notes in Computer Science*, pages 343–360. Springer, 2020. doi:10.1007/978-3-030-51825-7_25.
- 18 Philippe Jégou, Hanan Kanso, and Cyril Terrioux. Improving exact solution counting for decomposition methods. In *28th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2016*, pages 327–334. IEEE Computer Society, 2016. doi:10.1109/ICTAI.2016.0057.
- 19 Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. Counting models using connected components. In Henry A. Kautz and Bruce W. Porter, editors, *Proceedings of the Seventeenth National Conference on Artificial Intelligence, AAAI 2000*, pages 157–162. AAAI Press / The MIT Press, 2000. URL: <http://www.aaai.org/Library/AAAI/2000/aaai00-024.php>.
- 20 Stefan Kölbl, Gregor Leander, and Tyge Tiessen. Observations on the SIMON block cipher family. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015 – 35th Annual Cryptology Conference*, volume 9215 of *Lecture Notes in Computer Science*, pages 161–185. Springer, 2015. doi:10.1007/978-3-662-47989-6_8.
- 21 Jean-Marie Lagniez and Pierre Marquis. An improved decision-DNNF compiler. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017*, pages 667–673. ijcai.org, 2017. doi:10.24963/ijcai.2017/93.
- 22 Jean-Marie Lagniez, Pierre Marquis, and Anastasia Paparrizou. Defining and evaluating heuristics for the compilation of constraint networks. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming – 23rd International Conference, CP 2017*, volume 10416 of *Lecture Notes in Computer Science*, pages 172–188. Springer, 2017. doi:10.1007/978-3-319-66158-2_12.
- 23 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*, pages 530–535. ACM, 2001. doi:10.1145/378239.379017.
- 24 Umut Oztok and Adnan Darwiche. An exhaustive DPLL algorithm for model counting. *Journal of Artificial Intelligence Research*, 62:1–32, 2018. doi:10.1613/jair.1.11201.

- 25 Markus N. Rabe, Christoph M. Wintersteiger, Hillel Kugler, Boyan Yordanov, and Youssef Hamadi. Symbolic approximation of the bounded reachability probability in large Markov chains. In Gethin Norman and William H. Sanders, editors, *Quantitative Evaluation of Systems – 11th International Conference, QEST 2014*, volume 8657 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 2014. doi:10.1007/978-3-319-10696-0_30.
- 26 Neil Robertson and Paul D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986. doi:10.1016/0196-6774(86)90023-4.
- 27 Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50–64, 2010. doi:10.1016/j.jda.2009.06.002.
- 28 Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *SAT 2004 – The Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004. URL: <http://www.satisfiability.org/SAT04/programme/21.pdf>.
- 29 Tian Sang, Paul Beame, and Henry A. Kautz. Performing Bayesian inference by weighted model counting. In Manuela M. Veloso and Subbarao Kambhampati, editors, *The Twentieth National Conference on Artificial Intelligence, AAAI 2005*, pages 475–482. AAAI Press / The MIT Press, 2005. URL: <http://www.aaai.org/Library/AAAI/2005/aaai05-075.php>.
- 30 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, pages 1169–1176. ijcai.org, 2019. doi:10.24963/ijcai.2019/163.
- 31 Fabio Somenzi. CUDD: CU decision diagram package—release 3.0.0, 2015. URL: <https://github.com/ivmai/cudd>.
- 32 Ben Strasser. Computing tree decompositions with FlowCutter: PACE 2017 submission. *CoRR*, abs/1709.08949, 2017. arXiv:1709.08949.
- 33 Marc Thurley. sharpSAT – counting models with advanced component caching and implicit BCP. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing – SAT 2006, 9th International Conference*, volume 4121 of *Lecture Notes in Computer Science*, pages 424–429. Springer, 2006. doi:10.1007/11814948_38.
- 34 Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979. doi:10.1137/0208032.

Failure Based Variable Ordering Heuristics for Solving CSPs

Hongbo Li ✉ 

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 Li¹ ✉

College of Computer Science and Technology, Jilin University, Changchun, China

Abstract

Variable ordering heuristics play a central role in solving constraint satisfaction problems. In this paper, we propose failure based variable ordering heuristics. Following the fail first principle, the new heuristics use two aspects of failure information collected during search. The failure rate heuristics consider the failure proportion after the propagations of assignments of variables and the failure length heuristics consider the length of failures, which is the number of fixed variables composing a failure. We performed a vast experiments in 41 problems with 1876 MiniZinc instances. The results show that the failure based heuristics outperform the existing ones including activity-based search, conflict history search, the refined weighted degree and correlation-based search. They can be new candidates of general purpose variable ordering heuristics for black-box CSP solvers.

2012 ACM Subject Classification Computing methodologies

Keywords and phrases Constraint Satisfaction Problem, Variable Ordering Heuristic, Failure Rate, Failure Length

Digital Object Identifier 10.4230/LIPIcs.CP.2021.9

Category Short Paper

Supplementary Material *Software (Source Code)*: <https://github.com/lihb905/fbs/>
archived at `swb:1:dir:2c2d84dba840b8f5299bf2bc2edc8df70031c82a`

Funding *Hongbo Li*: The National Natural Science Foundation of China under Grant NO. 61802056.

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

Constraint satisfaction problems (CSP) are a powerful framework to model and solve combinatorial search problems occurring in various fields. The challenge in a CSP is to determine an assignment of values to all variables that satisfies all the constraints, or otherwise, to prove there is no such an assignment. Backtracking search is a complete method that has been used to solve general 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 algorithms for solving CSPs. It is a computationally difficult task to find an optimal ordering that results in a search tree exploring the fewest number of nodes [11], thus, the ordering is determined by variable ordering heuristics (VOH) in practice.

¹ Minghao Yin and Zhanshan Li are corresponding authors.



© Hongbo Li, Minghao Yin, and Zhanshan Li;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 9; pp. 9:1–9:10



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In the past decades, much effort has been done in developing efficient variable ordering heuristics. Many VOHs have been designed according to the fail first principle that “to succeed, try first where you are likely to fail” [6]. The aim is to first process the variables that belong to the most difficult part of a CSP. Modern VOHs are adaptive, which learn during search to find the variables that are most likely to cause a failure. To select the next variable to assign a value at a search tree node, the VOHs estimate how likely a variable causing a failure from various aspects. For instances, the minimum domain size VOH considers the variables with the smallest domain sizes [6], the weighted degree VOH considers the variables involved in the constraints causing more failures have a larger chance to cause a failure [2, 20], some VOHs consider the variables involved in the constraints with higher tightness have a larger chance to cause a failure [3, 10], the conflict history search considers the variables involved in the constraints causing recent failures have a larger chance to cause a failure [5].

In this paper, we propose failure based variable ordering heuristics pursuing the fail first principle from new aspects. The new VOHs do not consider which constraint leads to a failure but which variable causes the failure. They consider the straightforward information between an assignment of a variable and the propagation result of the assignment, failure or success. To estimate how likely a variable causing a failure, the failure rate based heuristic collects the information of proportion of failures caused by assignments of a variable, and the failure length based heuristic collects the information of length of failures caused by assignments of a variable, i.e., the depth of the search tree when a failure occurs. There is no parameter to set in the failure based heuristics. We employ the decaying strategy of the conflict history search to make the new VOHs favor the variables causing recent failures. The failure based VOHs behave like the last conflict based reasoning [9], so the difference between them is discussed. We perform experiments in the benchmark set of MiniZinc containing 1876 instances of 41 problems. Besides the naive VOHs, we compare the VOHs equipped with a geometric fast restart strategy and last conflict based reasoning. The results show that the failure rate based VOH with the decaying strategy solves the largest number of instances. It outperforms the state of the arts VOHs including ABS [12], CHS [5] and *dom/wdeg^{ca.cd}* [20] and CRBS [18] and performs best in general.

The paper is organized as follows. Section 2 provides the background of CSPs. Some related works are mentioned in Section 3. The failure based VOHs are introduced in Section 4. Section 5 presents the experimental results. Finally, the conclusion is in Section 6.

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 $\mathcal{X} = \{x_1, x_2 \dots x_n\}$, \mathcal{D} is a set of domains $\mathcal{D} = \{dom(x_1), dom(x_2) \dots dom(x_n)\}$, where $dom(x_i)$ is a finite set of possible values for variable x_i , and \mathcal{C} is a set of e constraints $\mathcal{C} = \{c_1, c_2 \dots c_e\}$. Each constraint c consists of two parts, an ordered set of variables $scp(c) = \{x_{i1}, x_{i2} \dots x_{ir}\}$ and a subset of the Cartesian product $dom(x_{i1}) \times dom(x_{i2}) \times \dots \times dom(x_{ir})$ that specifies the disallowed (or allowed) combinations of values for the variables $\{x_{i1}, x_{i2} \dots x_{ir}\}$. An assignment of a variable x is in the form $(x = v)$ where v is a value in $dom(x)$.

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. Backtracking search performs a depth-first traversal of a search tree to solve general CSPs. In the context of a search tree, each edge is associated with an assignment, a node at level k is associated with a set of k assignments which are attached to the path from the root to this node. The root node at level 0 is an empty set.

We use *PastVar* to denote the set of fixed variables which have been assigned and *FutVar* to denote the set of future variables which have not been assigned. At each search tree node, a future variable is selected by a VOH and a new node is generated after the assignment to this variable, then a propagation algorithm filtering those inconsistent values from the domains of variables is applied. 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.

3 Related Works

Following the fail first principle, the most popular variable ordering heuristic, minimum domain size, selects the variable with smallest domain size [6]. It has been combined with many efficient VOHs. The *dom/deg* [1] and *dom/ddeg* [16] combine minimum domain size with largest variable degree. The weighted degree associates a weight with each constraint, which records the number of failures caused by the constraint [2]. It identifies the variables involved in difficult parts of problems. Combined with minimum domain size, the *dom/wdeg* has been one of the most efficient general purpose VOHs and has been used as default VOH by some solvers, such as Choco [14]. Its variants use different strategies to update the weights of constraints, such as constraint tightness [10] and the explanation information of a failure [7]. Its recent refinement, *wdeg^{ca.cd}*, combines current arity and current domains to update the constraint weights, has been shown to outperform the classic weighted degree heuristic [20]. The impact-based search (IBS) estimates the search space reduction after the propagation of an assignment of a variable [15]. It prefers the variables which may lead to the greatest search space reduction. The count-based search (CBS) considers solution densities of constraints [13]. It prefers the variable-value pair with the largest solution density. The activity-based search (ABS) estimates how active a variable is, e.g., how often the variable is affected by the assignments of other variables [12]. It prefers the most active variables. The correlation based heuristic (CRBS) measures the possibility of having conflict between each pair of variables and estimates the degree of conflicts when choosing a variable [18]. It prefers the variable which is estimated to causing more conflicts. The conflict history search (CHS) considers the history of constraint failures [5]. It prefers the variables involved in those constraints causing recent failures. Given a set of candidate VOHs, the Multi-Armed Bandit(MAB) techniques have been used to estimate the best VOH on a CSP instance. It has been shown that the MAB-based methods are more efficient than any single candidate VOH [21, 19]. The last-conflict based reasoning can be combined with any VOH, called underlying VOH. Whenever an assignment of a variable x is canceled, such as the propagation of the assignment leads to a failure, x is stored as a last conflict variable. The strategy always selects the last conflict variable until its assignment succeeds. It makes the next selection by the underlying VOH if there is no last conflict variable stored.

4 Failure Based Search

The failure based variable ordering heuristics

Given a CSP, a CP solver applies a propagation algorithm F after an assignment of a variable x . If the propagation of the assignment of a variable x leads to a failure (a domain wipeout), then we say the failure is caused by x . Although the actual reason of the failure may contain other assignments, we consider only the last assignment as the reason here, because only the last assignment must be one of the reasons of the failure, whereas some of the previous assignments may not be the reasons.

► **Definition 1 (Failure Rate).** *In the context of backtracking search, the Failure Rate of a variable x $FR(x)$ is $\frac{failNum(x)}{assignNum(x)}$ where the $failNum(x)$ is the total number of failures caused by x and the $assignNum(x)$ is the total number of x being assigned (or selected) since the beginning of search.*

The failure rate based (FRB) variable ordering heuristic collects the information of $failNum(x)$ and $assignNum(x)$ to calculate the failure rate for each variable x . It selects the variable with the largest $\frac{FR(x)}{|dom(x)|}$. For each variable x , $failNum(x)$ and $assignNum(x)$ are initialized to 0.5 and 1 respectively. The initialization indicates that the default failure rate of each variable is 50% before the failure information is collected. After the searching starts, if a variable x leads to a failure soon, $FR(x)$ will increase. If the variable leads to a failure after being assigned several times, $FR(x)$ will decrease.

► **Definition 2 (Failure Length).** *If the propagation of an assignment of a variable x leads to a failure, then the length of the failure is $|PastVar| + 1$, where $PastVar$ is the set of fixed variables before the assignment.*

Failure length is the depth of the search tree when a failure occurs. The length of a failure caused by a variable x is denoted by $|failure(x)|$. The failure length based (FLB) variable ordering heuristic associates an accumulated failure length with each variable x , denoted by $AFL(x)$. For each variable, $AFL(x)$ is initialized to 0. If a failure is caused by a variable x during search, then $AFL(x)$ is updated as follows.

$$AFL(x) = AFL(x) + \frac{1}{|failure(x)|} \quad (1)$$

The FLB variable ordering heuristic selects the variable with the largest $\frac{AFL(x)}{|dom(x)|}$. The heuristic prefers the variables causing more shorter failures, i.e., the variables causing failures at higher levels of the search tree.

Modern VOHs usually use some decaying strategies to give the recent information more priority, such as CHS and ABS. We employ the strategy of CHS to make the failure based variable ordering heuristics prefer the variables causing recent failures. For each variable x , the factor of the decaying strategy is defined as,

$$A(x) = \frac{1}{\#TotalFailure - LastFailure(x) + 1} \quad (2)$$

where $\#TotalFailure$ is the total number of failures detected since the beginning of search, and $LastFailure(x)$ stores the $\#TotalFailure$ value of the last failure caused by x .

Based on the $A(x)$ factor, we propose a new VOH combining the FRB with $A(x)$, namely FRBA. We use addition to combine them here. The FRBA variable ordering heuristic selects the variable with largest $FRBA(x)$ defined as,

$$FRBA(x) = \frac{FR(x) + A(x)}{|dom(x)|} \quad (3)$$

Similarly, we propose a new VOH combining the FLB with $A(x)$, namely FLBA. The $AFL(x)$ may be much larger than $A(x)$ and the former may obscure the latter if we use addition here, so we use multiplication to combine them. The FLBA variable ordering heuristic selects the one with largest $FLBA(x)$ defined as,

$$FLBA(x) = \frac{AFL(x) \times A(x)}{|dom(x)|} \quad (4)$$

In backtracking search of CSPs, CP solvers usually use the binary branching (i.e., 2-way branching) strategy which has been shown to be more efficient than the non-binary branching strategy [8]. With binary branching, if the propagation of an assignment ($x = v$, a left branch) leads to a failure, a backtracking occurs and a propagation of the refutation ($x \neq v$, a right branch) is performed. Note that the failure based VOHs collect only the failure information of left branches, so the failures of right branches do not affect $FR(x)$ and $AFL(x)$. For $A(x)$, although $\#TotalFailure$ counts the failures of both left branches and right branches, $LastFailure(x)$ is updated only after a failure of left branch is detected.

Discussion

The failure based VOHs behave like the last conflict based reasoning that tend to select the variable causing the last failure, but the new VOHs do not strictly select the last one. We have mentioned $FR(x)$, $AFL(x)$, $A(x)$ and $|dom(x)|$ as scores to design the new heuristics. Given a variable x causing the last failure, we discuss how these scores change after the last failure detected.

- $FR(x)$: $FR(x)$ increases because $\frac{failNum(x)+1}{assignNum(x)+1}$ is always larger than $\frac{failNum(x)}{assignNum(x)}$.
- $AFL(x)$: $AFL(x)$ is increased by $\frac{1}{|failure(x)|}$.
- $A(x)$: If x causes the last failure, then $\#TotalFailure$ equals to $LastFailure(x)$. $A(x)$ will be the largest one.
- $|dom(x)|$: After the failure is detected, the value of the assignment of x will be removed from $dom(x)$, so $|dom(x)|$ decreases.

From the analysis, we can see if x causes the last failure, then all the four scoring function give x a score better than that before the failure. Its score was the best one, so the better score has a large chance to be selected. Thus, both the failure rate based VOHs and the failure length based VOHs tend to select the variable causing the last failure.

In binary branching strategy, a failure of a left branch is followed by a propagation of a refutation. Although the propagation of the refutation does not affect $FR(x)$ and $AFL(x)$, it may reduce the domains of other variables, so some other variables may have much smaller domain sizes. In this case, the variable x gets a better score after the backtracking, but it may not be the best one due to the propagation of the refutation. In addition, if the propagation of the refutation leads to a failure, the search will backtrack to a higher level, then some fixed variables with a score better than that of x may become available for branching. Thus, the failure based VOHs do not strictly select the variable causing the last failure.

5 Experiments

The experiments were run in Choco 4.10.6 [14]. The environment is JDK8 under CentOS 6.4 with Intel Xeon CPU E7-4820@2.00GHz processor and 58 GB RAM.

To examine the robustness of the proposed VOHs, we tested the VOHs with MiniZinc benchmark from <https://github.com/MiniZinc/MiniZinc-benchmarks>. Solving a problem with different modelling may get different performances, so we tested all the MiniZinc models of CSPs. The instances are flattened offline. After eliminating some large instances which cannot be flattened in 1 hour and the problems where infeasibility is proved at the root node, we include 41 problems with 1876 instances of 46 MiniZinc models in the experiments.

The performance of searching for the first solution or proving unsatisfiable are measured by CPU time in seconds. Timeout is set to 1200 seconds. We have used a random seed 0 throughout the experiments. Besides the naive version of the VOHs, we compare these VOHs equipped with a geometric restart strategy [4, 17] and last conflict based reasoning

strategies [9] storing one (LC-1) and five (LC-5) conflict variables respectively. The restart strategy uses 10 as the initial cutoff of failure count and 1.1 as the growing factor. The ABS uses its default value selector and all other VOHs use lexicographic ordering as the value selector.

In the following tables, we present the number of instances solved (*#solved*) by each VOH, the accumulated CPU time of each VOH solving the instances solved by all the compared VOHs (all solved time, *ast*) and the total CPU time of each VOH solving the instances solved by at least one of the compared VOHs (total time, *tt*). The integer in the brackets after *ast* is the number of all solved instances, so is the one after *tt*. The time cost of a timeout run is count as 1200s and we eliminated the results of the instances where all compared VOHs are timeout. The best one in each row is in bold.

In Table 1, we compare the failure based VOHs. We have three observations from the table. Firstly, FRBA performs better than FRB and FLBA performs better than FLB, so the decaying strategy improves the original failure based VOHs. Secondly, FLBA performs best when the restart strategy is not equipped, and FRBA performs best when the restart strategy is equipped. This is because the effective *FR* scores may be quickly learned after several restarts, so FRBA could make good decisions after some restarts. Finally, the VOHs get better performance when equipped with last conflict based reasoning. In general, the FRBA VOH solves the largest number of instances, so we use it as the representative one of the failure based VOHs in the following experiments.

■ **Table 1** Comparing the failure based VOHs.

			FRB	FRBA	FLB	FLBA
no restart	LC-0	<i>#solved</i>	702	711	818	870
		<i>ast</i> (596)	9,401	9,344	18,740	8,140
		<i>tt</i> (972)	348,961	340,532	224,463	162,034
	LC-1	<i>#solved</i>	738	741	828	879
		<i>ast</i> (618)	11,574	12,558	22,436	15,572
		<i>tt</i> (980)	320,280	325,257	222,613	168,985
	LC-5	<i>#solved</i>	767	763	862	894
		<i>ast</i> (638)	15,494	16,119	15,777	12,316
		<i>tt</i> (1,012)	339,881	344,336	217,003	184,585
	LC-0	<i>#solved</i>	949	985	984	1000
		<i>ast</i> (864)	21,357	11,730	19,218	15,504
		<i>tt</i> (1,087)	201,929	162,862	160,376	145,543
restart	LC-1	<i>#solved</i>	1,024	1,037	969	992
		<i>ast</i> (911)	14,895	11,813	17,361	17,014
		<i>tt</i> (1,094)	132,819	119,706	180,152	165,624
	LC-5	<i>#solved</i>	1,027	1,037	958	973
		<i>ast</i> (910)	14,676	13,990	21,305	29,482
		<i>tt</i> (1,097)	128,496	119,595	200,505	200,291

In Table 2, we compare FRBA with the state of the art VOHs. The ABS, CHS and *dom/wdeg*^{ca.cd} have been implemented in Choco and we implemented the CRBS and the proposed VOHs². We use the default parameters for these VOHs, which are recommended in

² The source code is available at <https://github.com/lihb905/fbs/>.

the literatures [12, 5, 20, 18]. For each MiniZinc model, we calculate the rate of the number of instances solved by each VOH to the number of instances solved by at least one VOH. The aggregated 46 rates are shown in the *solvedRates* rows. The results show that, when the restart strategy is equipped, FRBA gets the best performance. When the restart strategy is not equipped, FRBA is competitive with the existing ones and the CHS with LC-5 solves more instances than the other VOHs. The FRBA equipped with the restart strategy and last conflict based reasoning solves instances of largest number and its total time cost is less than that of the existing ones. The FRBA has the largest aggregated solved rate in most of the rows.

■ **Table 2** Comparing the failure based VOHs with the existing VOHs.

		ABS	CHS	<i>dom/wdeg^{ca.cd}</i>	CRBS	FRBA
no restart	LC-0	<i>#solved</i>	530	665	596	711
		<i>ast</i> (415)	7,104	6,306	16,860	5,603
		<i>tt</i> (801)	350,965	194,178	274,969	135,332
		<i>solvedRates</i>	36.65	35.96	32.31	37.47
	LC-1	<i>#solved</i>	557	744	647	770
		<i>ast</i> (427)	10,073	5,362	11,038	3,964
		<i>tt</i> (961)	528,055	303,136	415,091	274,381
		<i>solvedRates</i>	38.42	36.92	34.09	35.32
	LC-5	<i>#solved</i>	583	796	704	763
		<i>ast</i> (464)	10,279	9,213	18,073	6,998
		<i>tt</i> (998)	530,479	287,299	391,992	327,536
		<i>solvedRates</i>	39.90	38.88	36.19	35.88
restart	LC-0	<i>#solved</i>	630	730	700	985
		<i>ast</i> (443)	9,687	9,276	22,666	8,360
		<i>tt</i> (1,106)	618,393	492,387	536,574	185,662
		<i>solvedRates</i>	39.82	38.49	33.39	40.48
	LC-1	<i>#solved</i>	655	905	883	1,037
		<i>ast</i> (474)	13,022	8,552	14,409	8,493
		<i>tt</i> (1,134)	630,983	362,924	349,635	167,706
		<i>solvedRates</i>	40.37	39.74	36.22	41.32
	LC-5	<i>#solved</i>	674	960	929	1,037
		<i>ast</i> (475)	9,618	5,810	17,482	7,935
		<i>tt</i> (1,162)	661,177	300,729	331,411	197,595
		<i>solvedRates</i>	40.33	40.47	37.37	40.55

It has been shown that the VOHs equipped with restart and LC-5 is the best strategy in general, so we present the detailed results of the strategy in Table 3. The table includes all the 46 MiniZinc models of 41 problems. The integer in the brackets after each problem name is the total number of instances of the problem. In each cell, we present the number of solved instances and the number in the brackets is the total time cost of the instances solved by at least one VOH. The last row shows the numbers of problems where the corresponding VOH performs best. To decide which VOH performs best in a problem, we give a rule that considers the VOH solving instances of largest number as the best one. If a tie exists, we further compare the total time cost. It is shown that, the VOHs get best performance in different problems. Both ABS and FRBA get best performance in 13 problems, which is the largest number.

■ **Table 3** Detailed results of all problems.

Problems	ABS	CHS	$dom/wdeg^{ca.cd}$	CRBS	FRBA
alpha(1)	1(0.13)	1(0.11)	1(0.17)	1(0.31)	1(0.35)
amaze2(47)	10(1,117)	7(4,820)	4(7,693)	7(4,034)	7(4,102)
amaze3(47)	45(391)	44(1,242)	44(2,393)	43(2,727)	43(3,038)
areas(4)	4(0.21)	4(0.06)	4(0.1)	4(0.08)	4(0.07)
bibd(16)	15(805)	13(2,509)	9(7,617)	15(127)	15(15)
black-hole(21)	21(24)	21(51)	21(28)	21(72)	20(1,213)
carseq(79)	23(7,795)	0(27,600)	0(27,600)	0(27,600)	0(27,600)
cars(79)	37(17,480)	10(46,048)	14(42,313)	5(54,652)	18(37,850)
CostasArray(10)	8(1,880)	7(2,683)	8(1,642)	8(1,769)	9(1,270)
step1-aes(7)	2(2,417)	3(1,882)	3(1,739)	3(1,658)	2(2,408)
debruijn-binary(11)	4(3,678)	7(1,088)	7(1,094)	7(1,180)	7(1,013)
elitserien-noseasonal(10)	10(184)	10(56)	10(26)	10(113)	10(48)
eq20(1)	1(0.05)	1(0.17)	1(0.09)	1(0.12)	1(0.04)
fillomino(20)	17(170)	16(2,372)	16(3,976)	16(2,970)	16(2,983)
golfers1(9)	6(794)	6(117)	6(1,089)	6(259)	6(142)
golfers1b(9)	6(223)	6(15)	6(20)	6(47)	6(31)
golfers2(9)	5(2,347)	5(1,338)	2(4,820)	2(5,876)	3(3,775)
kakuro(6)	6(0.18)	6(0.15)	6(0.12)	6(0.11)	6(1.38)
knights(4)	4(2.18)	4(0.61)	4(0.53)	4(1.13)	4(6.79)
langford(25)	20(119)	20(87)	20(88)	20(112)	20(616)
latin-squares-fd(7)	7(24)	6(1,291)	5(2,594)	4(3,893)	5(3,640)
latin-squares-fd2(7)	7(1.66)	7(0.91)	7(0.97)	7(1.90)	7(10)
latin-squares-lp(7)	7(161)	7(1,294)	4(3,626)	4(3,684)	6(1,909)
magicseq(9)	7(2,503)	9(386)	9(191)	7(2,408)	7(2,468)
market-split(60)	39(9,858)	33(10,567)	30(15,614)	33(10,749)	36(11,169)
mknapsack(7)	7(2,035)	5(3,299)	2(6,717)	4(3,924)	5(3,388)
nmseq(20)	9(6,293)	13(3,687)	11(5,407)	11(6,266)	14(2,290)
non(26)	21(11,151)	23(4,783)	15(17,988)	15(16,631)	24(3,847)
nsp-1(200)	56(95,939)	71(66,452)	55(78,245)	18(130,871)	114(10,801)
nsp-2(200)	6(18,424)	0(22,800)	6(19,884)	0(22,800)	13(12,459)
oocsp-racks(6)	6(898)	6(46)	6(29)	6(86)	6(45)
pentominoes-int(7)	7(252)	7(96)	7(181)	7(41)	7(193)
QCP(60)	60(4,385)	59(5,488)	55(8,914)	51(13,722)	53(11,441)
quasigroup7(10)	5(16)	5(11)	5(14)	5(38)	5(40)
queens(7)	7(1.11)	7(129)	7(108)	7(4.10)	7(1.49)
rect-packing(56)	56(56)	56(12)	56(12)	56(10.02)	56(10.24)
rect-packing-mznc2014(56)	56(54)	56(13)	56(9.16)	56(12)	56(10)
rubik(5)	4(2,298)	5(541)	1(4,826)	3(2,771)	3(2,458)
schur(3)	3(0.07)	3(0.05)	3(0.08)	3(0.06)	3(0.06)
search-stress2(1)	1(0.63)	1(0.59)	1(0.56)	1(0.46)	1(0.48)
search-stress(3)	2(3.00)	2(3.97)	2(40)	2(3.30)	2(2.91)
slow-convergence(10)	10(8.57)	10(9.39)	10(519)	10(355)	10(8.30)
solbat(39)	36(6,491)	32(10,813)	28(18,975)	24(22,157)	26(19,404)
tents(3)	3(0.33)	3(0.19)	3(0.24)	3(0.08)	3(0.07)
wwtpp-random(251)	0(168,000)	125(20,864)	133(11,650)	134(13,203)	138(5,298)
wwtpp-real(401)	7(292,879)	218(56,216)	226(33,711)	223(40,806)	232(20,576)
Sum of bests	13	10	5	5	13

In Table 4, we compare each pair of the VOHs according to the previous rule. The number of problems where each VOH performs better is present in the table. We can see that FRBA performs better than the others in general.

■ **Table 4** Comparing FRBA with the existing VOHs by pairs.

		ABS	FRBA	CHS	FRBA	$dom/wdeg^{ca.cd}$	FRBA	CRBS	FRBA
no restart	LC-0	21	25	20	26	17	29	19	27
	LC-1	22	24	21	25	13	33	15	31
	LC-5	26	20	31	15	21	25	16	30
restart	LC-0	24	22	24	22	13	33	18	28
	LC-1	20	26	22	24	13	33	9	37
	LC-5	21	25	23	23	18	28	15	31

The numbers of instances of each MiniZinc model vary greatly. If a VOH works well in some models containing a large number of instances, it may get a good overall performance. To balance the effect of instance set size, we randomly select $\frac{1876}{46} = 41$ instances from the instance set of each MiniZinc model to generate a smaller benchmark set (46 is the number of MiniZinc models). If an instance set contains less than 41 instances, we select them all. The smaller benchmark set contains 832 instances. The results are present in Table 5. We can see that, when the restart strategy is not equipped, FRBA clearly outperforms the others. When the restart strategy is equipped, FRBA is competitive with the existing ones. In general, FRBA gets the best performance in the smaller benchmark set.

■ **Table 5** Comparing FRBA with the existing VOHs in the smaller benchmark set.

			ABS	CHS	$dom/wdeg^{ca.cd}$	CRBS	FRBA
no restart	LC-0	<i>#solved</i>	451	455	420	430	478
		<i>ast</i> (351)	6,504	5,476	15,569	11,961	4,630
		<i>tt</i> (538)	127,769	119,427	167,554	150,743	89,434
	LC-1	<i>#solved</i>	461	467	431	454	490
		<i>ast</i> (360)	9,579	4,594	9,509	9,833	3,219
		<i>tt</i> (552)	142,492	124,518	175,928	143,740	92,591
	LC-5	<i>#solved</i>	484	491	466	461	497
		<i>ast</i> (391)	8,600	7,692	16,355	15,180	6,156
		<i>tt</i> (565)	121,385	110,707	148,158	148,961	104,365
	LC-0	<i>#solved</i>	505	488	427	477	521
		<i>ast</i> (371)	7,116	8,002	22,176	12,087	7,574
		<i>tt</i> (598)	142,153	161,329	239,873	177,064	120,197
restart	LC-1	<i>#solved</i>	508	510	470	482	527
		<i>ast</i> (389)	10,017	5,897	13,847	11,099	8,068
		<i>tt</i> (590)	133,624	126,346	169,283	155,500	106,790
	LC-5	<i>#solved</i>	514	525	489	480	533
		<i>ast</i> (400)	6,316	3,890	16,542	12,876	7,399
		<i>tt</i> (608)	150,353	124,973	176,728	181,416	122,106

6 Conclusion

In this paper, we propose failure based variable ordering heuristics for solving CSPs. The new VOHs consider failure rate and failure length to estimate how likely an assignment of a variable causing a failure. All the heuristic information are collected during search, so the new heuristics are parameter-free. The experiments in the MiniZinc benchmark set show that the failure based VOHs outperform the state of the art VOHs in general. They can be new candidates of general purpose variable ordering heuristics for black-box CSP solvers.

References

- 1 C. Bessière and J. C. R  gin. Mac and combined heuristics: two reasons to forsake fc (and cbj?) on hard problems. In *Proc. CP'96*, pages 61–75. Springer, 1996.
- 2 F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proc. ECAI'04*, pages 146–150, 2004.
- 3 I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proc. CP'96*, pages 179–193. Springer, 1996.
- 4 C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. AAAI'98*, pages 431–437. AAAI, 1998.
- 5 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.
- 6 R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
- 7 E. Hebrard and M. Siala. Explanation-based weighted degree. In *Proc. CPAIOR'17*, pages 167–175. Springer, 2017.
- 8 J. Hwang and D. G. Mitchell. 2-way vs d-way branching for csp. In *Proc. CP'05*, pages 343–357. Springer, 2005.
- 9 C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.
- 10 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.
- 11 P. Liberatore. On the complexity of choosing the branching literal in dpll. *Artificial Intelligence*, 116(1):315–326, 2000.
- 12 L. Michel and P. Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *Proc. CPAIOR'12*, pages 228–243. Springer, 2012.
- 13 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.
- 14 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>.
- 15 P. Refalo. Impact-based search strategies for constraint programming. In *Proc. CP'04*, pages 557–571. Springer, 2004.
- 16 B. M. Smith and S. A. Grant. Trying harder to fail first. In *Proc. ECAI'98*, pages 249–253, 1998.
- 17 T. Walsh. Search in a small world. In *Proc. IJCAI'99*, pages 1172–1177, 1999.
- 18 R. Wang, W. Xia, and R. H. C. Yap. Correlation heuristics for constraint programming. In *Proc. ICTAI'17*, pages 1037–1041. IEEE, 2017.
- 19 H. Watez, 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.
- 20 H. Watez, C. Lecoutre, A. Paparrizou, and S. Tabary. Refining constraint weighting. In *Proc. of ICTAI'19*, pages 71–77. IEEE, 2019.
- 21 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.

Generating Magical Performances with Constraint Programming

Guilherme de Azevedo Silveira   

Alura, São Paulo, Brazil

Abstract

Professional magicians employ the use of interesting properties of a deck of cards to create magical effects. These properties were traditionally discovered through trial and error, the application of heuristics or analytical proofs. We discuss the limitations of relying on humans for such methods and present how professional magicians can use constraint programming as a computer-aided design tool to search for desired properties in a deck of cards. Furthermore, we implement a solution in Python making use of generative magic to design a new effect, demonstrating how this process broadens the level of freedom a magician can decree to their volunteers while retaining control of the outcomes of the magic. Finally, we demonstrate the model can be easily adapted to multiple languages.

2012 ACM Subject Classification Applied computing → Computer-aided design; Theory of computation → Constraint and logic programming

Keywords and phrases Constraint, generative design, computer aided design, constraint programming, generative magic, magical performance

Digital Object Identifier 10.4230/LIPIcs.CP.2021.10

Category Short Paper

Supplementary Material *Software*: <https://doi.org/10.5281/zenodo.5148915> [7]

Audiovisual: <https://doi.org/10.5281/zenodo.5148882> [6]

Acknowledgements We thank Daniela Mikyung Song for assistance providing the card designs and illustrations.

1 Introduction

A central problem in the performing arts of magic concerns designing new effects which are easily reproduced while complex enough so they are not easily figured out by spectators. Magicians traditionally use trial-and-error procedures that take time and are limited to only specific situations such as the creation process behind *Poker Night at the Improv* [14]. Others create heuristics such as the *System for Arranging Cards for Any Spelling Combination* [16], but heuristics might not work under different circumstances.

Mathematically inclined magicians publish proofs of properties on a deck of cards, such as the properties of a *Faro Shuffle* [10, 19, 21, 20]. Programmers create closed source code exploring possibilities on a memorized deck as in the *Poker Formulas* [14].

Even after an effect is published, it might not be replicated by magicians who perform them in other languages since many card effects use characteristics from their own cultures. One such example is the language bias present in many spellings of card effects, such as the value and suit in English which are fundamental parts of the *Spelling a Card* [16]. Other effects use double meanings of words such as Jacks or clubs. Memorized decks such as the *Aronson Stack* [1] are built around card spelling characteristics from the English language.

Moreover, nationality bias diminishes the reach of magical effects beyond the creator's cultural bubble. Poker is a central theme in many magical effects [23, 17, 11, 25, 15], although many countries have their own games [22] which better represent their identities.



© Guilherme de Azevedo Silveira;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 10; pp. 10:1–10:13

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

10:2 Applying Constraint Programming to Magic

Therefore, designing effects based on the properties of a deck of cards is traditionally a time-consuming process, limited by the cultural aspects, biases and experience of a creator.

Mentalism

In *mentalism*, a volunteer makes n choices and the magician reveals a matching prediction [3]. While some effects allow for free choices, others will force the volunteer to a predetermined selection. One such example is the *P.A.T.E.O. force* [18]. The magician displays n items and, along with the volunteer, take turns removing items one by one. The remaining item is always one that the magician determined ahead of time. For the magic inclined, the method is further explained in Appendix A.

There are two hints to the volunteers that the magician is guiding them to choose a predetermined option. First, the magician is always involved in the process of removing an item, even if it is the volunteer's turn. Second, the magician makes the final choice. In this example and in the most common styles of forces, volunteers might perceive that they were forced to make a choice, thereby ruining the entire sensation of the uncontrolled environment the magician is trying to build.

This motivates us to ask whether there are methods to provide free choices to volunteers while retaining control on the final result of the effect.

Our contribution

By defining and limiting the parameter space of the choices of a volunteer, we demonstrate the application of constraint programming to control and infer results from random selections made by such a person. Python code is provided to replicate these findings [7] and a framework [8] was open sourced that can be used to design and generate magic under these premises.

Outline

This paper is organized as follows: Section 1 presents the background information necessary to understand the limitations of traditional creative magical methods. Section 2, then, examines how some magical effects can be generated through constraint programming; following which Section 3 briefly goes through the sample code that designs and creates such effects. Finally, Section 4 reports on the results achieved in performing these generative magic effects and discusses their real life usability before conclusions are presented in Section 5.

2 Controlling outcomes through constraint programming

Several spelling effects are described in the literature, where the magician deals a card as the chosen card name is spelled. A simple version of the spelling of a freely chosen card comprises of the following procedure.

The magician removes one blue and one red case from their pocket and places them on a table. The magician points to the blue case and announces that it contains one prediction. Next, they open the red one and spreads the 52 cards face down over the table, asking a volunteer to remove one card from the spread while hiding it from the magician.

The magician puts away both the remaining 51 cards and the red deck case back into their pocket. From the spectators' perspective, from this moment on, there is nothing the magician can do because both the prediction and the volunteer's free choice have already been made. The volunteer then reveals the chosen card was, for instance, the five of spades.

The performer proceeds to spell the name of the mentioned card, dealing one card from the top of the blue deck for each letter. Since the “five of spades” is spelled with 12 letters, the magician deals 12 cards and reveals the 13th card to be the five of spades. Spreading the other 51 cards face up, the magician reminds the volunteer that they could have chosen any card, yet, they both chose the same five of spades.

This effect consists of two parts, the first one is a free selection from the volunteer which turns out to be a forced card. To achieve this result, one can make use of a force red deck. One of such decks is a one-way deck consisting of 52 cards having the same red back and the same face, in this example, the five of spades. This self-working effect [12] is easy to perform because it only requires the preparation of the blue deck by positioning its five of spades in the 13th position.

Convincers and issues

To make the effect stronger, the magician can use false shuffles on both decks, such as the *Mead and Kennedy false shuffle* [5]. False shuffles temporarily displace some of the cards but end the movement with all cards in their original order prior to the shuffle.

They can follow it with false cuts, which do not change the deck order.

A third method consists in placing the five of spades at position 7 and execute controlled shuffles, such as the *Out-Faro Shuffle* [19] which brings the card to the 13th position. The Faro is described in Appendix A

Even the use of stronger convincers might not be enough for this effect as currently presented, since the magician cannot hand the red cards to the volunteers for further inspection. Depending on the force deck in use, it cannot even be spread face up.

Choices, control and knowledge

While the volunteer made a free choice in the previous effect, the magician controlled its result by giving them no other option but the five of spades. The magician decided beforehand which card would be chosen and placed it in its expected position into the deck.

The selections made by the volunteer can be understood as parameters to the magic effect. Parameters can be randomly chosen from their own parameter space. In the example given, there is one $[1,52]$ space, making it a 1-parameter magic effect.

The question raised is whether magical methods exist that allow volunteers to make N random choices over a N -dimensional parameter space while handing them real control over the card sequence and, yet, allow the magician to force the result.

In this paper, we aim to confirm that it is indeed possible. A magician can make correct predictions about the outcome of N random choices made by a volunteer based on a set of items such as a 52 card deck by controlling other variables through constraint programming.

3 Constraining for freedom

Our desired effect, *Freedom of Spelling*, consists of attributing real freedom over the parameters the volunteer will choose. The magician lays a blue and red case each on the table. The red cards are removed from the deck and fanned, revealing 52 different examinable cards as in figure 1a. They are spread face down on the table. The volunteer chooses 3 cut points resulting in 4 packets as in figure 1c, and decides a permutation that defines the sequence in which the packets will be put back together.

10:4 Applying Constraint Programming to Magic

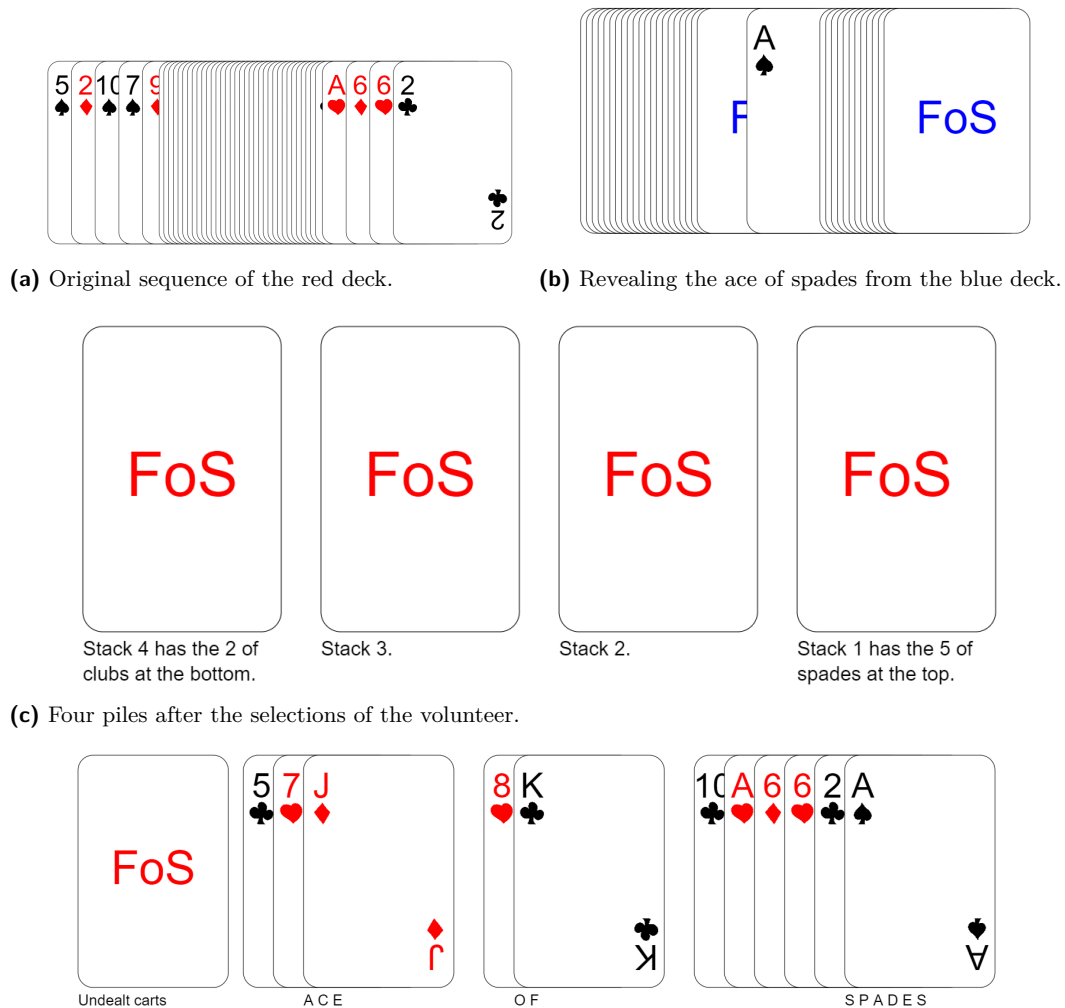


Figure 1 Performance of the Freedom of Spelling.

The volunteer has therefore 6 free choices and real control over the resulting card sequence. The magician emphasizes two points. First, the volunteer has made free choices. Second, there are $52!$ different combinations of a deck of cards. The magician proceeds to reveal one single card turned face down on the blue deck, for example, the ace of spades as in figure 1b. Spelling the “ace of spades”, the card is at the expected 12th position in the red deck which was controlled by the volunteer as in figure 1d.

There are two parts to this effect. In the first part, the volunteer freely makes 6 choices from a 6-dimensional parameter space. The resulting card sequence is one out of many possibilities. The volunteers are fooled to believe that there are $52!$ possible sequences that they might generate in this process while the magician has never explicitly said so. The two phrases are disconnected but the volunteers do not perceive it due to its misdirection and their misunderstanding of the possibilities generated by a 6-dimensional space parameter.

The first cut is made anywhere between position 2 and 49. The second cut between the first one and 50, the third one between the second one and 51. Finally the volunteer chooses any of the 24 permutations of the four stacks to gather the deck back together in one stack, which gives only a subset of all possible card sequence combinations, one in the order of 2 million possibilities.

The magician needs to know what position the volunteer has cut to. They can count the position by spreading the cards and asking the volunteer where to cut. Another method is to use number markings on the back of the cards as in *Card Control by the Numbers* [24].

The second part of the effect is to reveal that there is a card in its expected position. If the ace of spades is in the 12th position, the magician proceeds to use an index method to reveal the card as a supposed prediction. The method described earlier, the *Invisible Deck* [2], would display the ace of spades as the single card face down.

The question we are left with is will there always be at least one card in the expected position no matter the choice of parameters?

An unexpected way to model this problem aids in its solution. The magician has control of two aspects of the effect: the starting deck sequence and the card to be revealed. The first one is a set of 52 variables that start with no constraint. The card to be revealed should be defined by the parameters chosen by the volunteer during the live performance.

Because the first part of the effect defines 6 parameters and a sequence of array operations that are performed on the stack of cards, one can create a set of constraints that are required in order to guarantee the existence of one card in its expected position at the end of the process. In order to achieve it, we define the 6-dimensional parameter spaces. The 3 cut points are defined in closed intervals, and the final sequence is defined by the first 3 values of a permutation.

- (a) $cut_1 \in [2, 49], cut_2 \in [cut_1, 50], cut_3 \in [cut_2, 51]$
- (b) $sequence \in S(\{1, 2, 3, 4\})$

The second step is to extract the length of each card's name in the language that the effect will be performed. For instance, the ace of clubs has the length 10 while the king of diamonds has the length 14.

Given the 6 parameters and starting with a deck numbered $[1, 2, \dots, 52]$, one can simulate the cuts and deck rebuilding, obtaining the final deck order. For example, cutting to the 10th, 20th and 30th card gives $cut_1 = 10, cut_2 = 20$ and $cut_3 = 30$. Using the permutation $[3, 1, 2, 4]$ results in the sequence 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, \dots , 20, 31, 32, 33, 34, \dots , 51, 52.

Note that the card that ends in the 11th position is the 1, while the 2 ends at position 12. If a card spelled with 10 letters begins at position 1 it would end at the expected 11th position.

One can now define a constraint that says the ace of clubs must start at position 1 to end at position 11. Another constraint requires that the king of spades starts at position 3. But those constraints do not need to hold at the same time, only one of the 52 constraints need to be satisfied as in listing 1.

Unfortunately, this is not yet enough to generate a magic effect. Satisfying one of such constraints gives us a set of starting deck position rules that work only for this specific point in the parameter space. The code must explore the entire parameter space, generating a set of 2 million constraints, each one consisting of an Or clause with 52 constraints.

Finally, adding the requirement that all variables are non-repeating integers in the space $[1, 52]$, a solver implementation can be used to check for satisfaction and, if possible, a unique solution that works for any set of parameters the volunteer chooses.

10:6 Applying Constraint Programming to Magic

■ **Listing 1** List of example constraints which must have at least one satisfied.

```
ace of clubs starts at 1 or
two of clubs starts at 1 or
...
king of spades starts at 3
```

■ **Listing 2** A function that simulates the cuts and joins.

```
def simulate(cuts: List[int], deck: List[int]):
    stacks = np.array([range(cuts[0]),
                       range(cuts[0], cuts[1]),
                       range(cuts[1], cuts[2]),
                       range(cuts[2], 53)])
    return np.concatenate(stacks[deck])
```

If a solution exists, the performer can use that specific order for the starting red deck, perform false shuffles and cuts. The volunteer makes their 6 choices. The performer looks up a table, as in *Poker Formulas* [14], or is hinted by a computer on which card is at the expected position in the red deck. The magician proceeds to reveal that card face down in the blue deck and finally spells the card in the correct position in the red one.

4 The Z3Solver solution

Starting with a deck in any order, one needs to simulate the card movements by slicing and joining arrays. Listing 2 creates a sequence and decides where each card in the original order of 1 to 52 ends up.

For example, a number 15 in the second position of the returned NumPy [13] array means that the 15th card from the original deck ends up at the 2nd position of the deck.

Using Z3Solver [9], a SMT solver, 52 integer variables are created representing the starting position of their respective cards as shown in listing 3. The first 13 variables stand for the ace, 2, 3, ..., 10, jack, queen and king of clubs. The next 39 variables represent the same cards in the suits of hearts, spades and diamonds.

Every card must have a constraint limiting its position to [1,52] and no two cards can occupy the same starting position.

The next step to constrain the original deck order to the requirements of a given set of 6 parameters is to simulate the card movements with a sample deck. Then, use its output to generate the required constraints as in listing 4.

The 52 conditions can be generated by going over each card extracting its name and length. By using the number that finishes at the expected position in the deck array, we define where such a card should be placed at the beginning of the effect, as in listing 5.

This allows the exploration of a single point in the parameter space, therefore, it is required to explore the 6-dimensional discrete parameter space, invoking *freedom_of_spelling* and generating a new rule for each execution as in listing 6.

■ **Listing 3** Defining all 52 card variables.

```
names = map(retrieve_card_name, range(1, 53))
all_vars = list(map(z3.Int, names))
```

■ **Listing 4** Simulates the slices, joins and return the proper constraints.

```
def freedom_of_spelling(all_vars:List[z3.Int], cuts:List[int],
                      sequence: List[int]) -> z3.Or:
    deck = simulate(cuts, sequence)
    return spelling_rules(all_vars, deck)
```

■ **Listing 5** Card rules.

```
rules = []
for card in range(1, 53):
    name = retrieve_card_name(card)
    length = len(name.replace("_", ""))
    original_position = deck[length]
    rules.append(all_vars[card-1] == original_position)
return z3.Or(rules)
```

5 Results

The Z3Solver is unable to find a solution in less than 24 hours of runtime. The following optimizations and variations can be implemented to make its runtime faster and this method of magic creation more approachable to magicians without many resources.

Redesigning for a solution: multiple outs

The performer has only *one out* so far. For example, the “ace of spades” must be at position 12 so the magician reveals it when spelling the last letter. Such constraint can be relaxed by allowing the performer *two outs*. if the “ace of spades” is at position 13, the reveal is made after the complete spelling. The volunteers are never aware of the two possible outcomes. The relaxed constraint says that a card with n letters can be at position n or $n + 1$.

Optimizing

All card names in English have length between 10 and 15. Therefore, if different starting parameters end up with the same cards in these positions, the redundant ones are removed.

Also, although all cuts are possible, during performances, this is not true. Magicians know that volunteers do not make their cut on the first few cards even when given a free choice. Therefore, limiting the parameter space as follows achieves the same magical effect.

- (a) $cut_1 \in [7, 21]$
- (b) $cut_2 \in [\max(cut_1, 12), 26]$
- (c) $cut_3 \in [\max(cut_2, 25), 39]$
- (d) $sequence \in S(\{1, 2, 3, 4\})$

■ **Listing 6** Exploring the 6-dimensional parameter space.

```
for cut1 in range(2, 49):
    for cut2 in range(cut1, 50):
        for cut3 in range(cut2, 51):
            for sequence in permutations(range(4)):
                cuts = (cut1, cut2, cut3)
                rule = freedom_of_spelling(all_vars, cuts, sequence)
                rules.append(rule)
```

10:8 Applying Constraint Programming to Magic

■ **Table 1** Stack order for *Freedom of Spelling in English*, from the deck top to its face.

1	5 of hearts	14	8 of clubs	27	7 of diamonds	40	8 of spades
2	2 of diamonds	15	J of spades	28	4 of diamonds	41	K of hearts
3	10 of spades	16	3 of diamonds	29	J of hearts	42	6 of spades
4	7 of spades	17	K of spades	30	J of diamonds	43	J of clubs
5	9 of diamonds	18	A of clubs	31	9 of hearts	44	7 of hearts
6	7 of clubs	19	4 of spades	32	K of diamonds	45	3 of hearts
7	9 of clubs	20	5 of clubs	33	5 of spades	46	10 of hearts
8	8 of hearts	21	9 of spades	34	5 of diamonds	47	Q of spades
9	6 of clubs	22	Q of hearts	35	3 of clubs	48	10 of clubs
10	A of diamonds	23	10 of diamonds	36	4 of clubs	49	A of hearts
11	Q of clubs	24	4 of hearts	37	8 of diamonds	50	6 of diamonds
12	K of clubs	25	2 of hearts	38	3 of spades	51	6 of hearts
13	A of spades	26	2 of spades	39	Q of diamonds	52	2 of clubs

After optimizing, 2,049 constraints for English are created in 8 minutes and analyzed, generating one of many sequences of the deck that satisfies the desired properties.

The entire source code was released [7] and can be found in Appendix B. A new project was released as a library [8], allowing magicians to explore methods, create effects and share contributions to advance the field of generative magic.

Preshow and performance

The resulting stack order, *Freedom of Spelling in English*, is presented in table 1. Other solutions exist for card name length tables using different languages.

During the performance, the simulation function is run once to obtain the matching card for the set of parameters chosen by the volunteer.

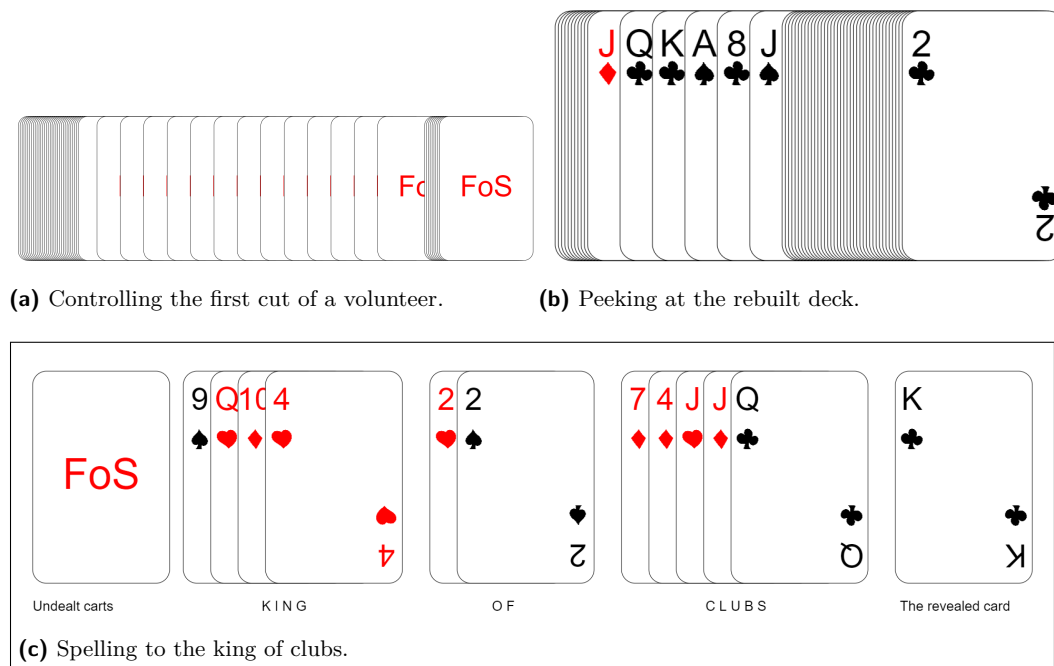
The effect was performed online in English, Korean and Portuguese by a professional magician. In a close up presentation it is preferred to use low tech - such a clicker - to input the position of the cuts. A smartwatch or hidden thumper can notify the magician which the card is at its expected position. These technologies are already in use in close up magic performances.

As a performance example, figure 1a shows the stacked deck from table 1. A volunteer makes the first cut as in figure 2a.

Supposing the cuts are made at positions 10, 20 and 30. When the volunteer rebuilds the stack using the sequence [3, 2, 1, 4], the resulting 10th to 15th cards can be seen in figure 2b: the jack of diamonds, queen of clubs, king of clubs, ace of spades, eight of clubs and jack of spades. In this case, the king of clubs is at position 12, its length plus 1. Therefore, in a second deck, the magician displays a single card face down, the king of clubs. They spell “king of clubs” while dealing 12 cards, revealing the king of clubs as in 2c. The entire performance with the explanation was made available [6].

6 Conclusion and further research

We presented a novel problem formulation to the design process of new effects in magic. Our implementation proves that new magical effects can be devised and implemented through the use of computer-aided design (CAD) frameworks. We devised and applied one effect using constraint programming, giving the volunteers more freedom while keeping control of the results with the magician.



■ **Figure 2** Simulation of a performance with parameters [10, 20, 30, 3, 2, 1, 4].

Because the code is open source, anyone can run it using a different language, therefore, limiting some of the unintentional biases and boundaries a magical effect has due to the identity and experience of its creator.

Further research can be done optimizing the model, using other solvers and trying to remove the constraints from the parameter space. Other effects can be generated such as allowing a volunteer to choose the language to be used only after the deck has been handled by the performer.

Generative magic can be explored with other areas of mathematics and computer science. Its ultimate research challenge is to search and catalog in programming terms the existing effects in magic literature so that an engine can co-design new ones.

References

- 1 Simon Aronson. *A Stack To Remember*. Self-published, 1979.
- 2 J. B. Bobo. *WATCH THIS ONE!* Lloyd E. Jones, 1947.
- 3 Robert Cassidy. *The Art of Mentalism*. Collectors' Workshop, 1984.
- 4 Michael Close. *Workers Number 5*. Self-published, 1996.
- 5 Michael Close. *Closely Guarded Secrets*. MichaelClose.com, 2 edition, 2004.
- 6 Guilherme de Azevedo Silveira. Freedom of spelling demonstration, 2021. doi:10.5281/zenodo.5148882.
- 7 Guilherme de Azevedo Silveira. Freedom of spelling source code, July 2021. doi:10.5281/zenodo.5148915.
- 8 Guilherme de Azevedo Silveira. Generative magic, 2021. URL: <https://github.com/guilhermesilveira/generativemagic/>.

10:10 Applying Constraint Programming to Magic

- 9 Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, C. R. Ramakrishnan, and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-78800-3_24.
- 10 Persi Diaconis, R.L Graham, and William M Kantor. The mathematics of perfect shuffles. *Advances in Applied Mathematics*, 4(2):175–196, 1983. doi:10.1016/0196-8858(83)90009-X.
- 11 Karl Fulves. *Hocus Poker*. Self-published, 1982.
- 12 Glenn Gravatt. *Encyclopedia of Self-working Card Tricks*. Quality Magic Company, 1936.
- 13 Charles R. Harris, K. Jarrod Millman, St’efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern’andez del R’io, Mark Wiebe, Pearu Peterson, Pierre G’erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. doi:10.1038/s41586-020-2649-2.
- 14 Pit Hartling. *In Order To Amaze*. Self-published, 2016.
- 15 Jean Hugard and Frederick Braue. *The royal road to card magic*. Farber and Farber, London, 1979. OCLC: 25036735.
- 16 Jean Hugard, John J. Crimmins, and Glenn G. Gravatt. *Encyclopedia of card tricks*. Dover Publications, New York, 1974.
- 17 Edward Marlo. *The Ten Hand Poker Stack*. Self-published, 1974.
- 18 Hugh Miller. *Baker’s Bonanza*. Supreme Magic Publication, 2 edition, 1978.
- 19 Stephen Minch. *The Collected Works of Alex Elmsley*, volume 2. L & L Publishing, 1994.
- 20 S. Brent Morris. The basic mathematics of the faro shuffle. *Pi Mu Epsilon Journal*, 6(2):85–92, 1975. URL: <http://www.jstor.org/stable/24345166>.
- 21 S.Brent Morris and Robert E. Hartwig. The generalized faro shuffle. *Discrete Mathematics*, 15(4):333–346, 1976. doi:10.1016/0012-365X(76)90047-9.
- 22 Nike Arts. *Enciclopedia de los juegos de cartas*. RobinBook, Barcelona, 1999. OCLC: 807845266.
- 23 Darwin Ortiz. *Darwin Ortiz on casino gambling: the complete guide to playing and winning*. Dodd, Mead & Co, New York, 1st ed edition, 1986.
- 24 McCabe Pete and Michael Close. *The PM Card System*. MichaelClose.com, 2019.
- 25 David Regal. *Close-up & personal*. Hermetic Press, Seattle, Wash., 1999. OCLC: 45272617.

A Mentioned effects

Magic is an uncommon theme to the field of constraint programming. Therefore, this appendix further describes a few of the effects mentioned earlier.

P.A.T.E.O. force

During a dinner event, the performer writes “bottle” in a piece of paper and leaves it folded over the table, in plain sight. Nobody but the performer knows what is written as this is the magician’s prediction. A volunteer is chosen and the performer selects and names 5 items for them to play with, such as a napkin, glass, bottle, fork and knife.

The magician points to two of those items. The volunteer is asked to select one amongst the two to remove. With four elements left, it is the volunteer's turn to point to two items. The magician selects one amongst the two and removes it. This process is repeated until there is only one element, which will always be the prediction element – in this case, the bottle.

For this effect to work, the magician must always select two items which do not include the prediction. During the volunteer's turn, if they point to two items that do not include the prediction, the magician can remove any element. If the volunteer points to one item and the prediction item, the magician must remove the former. Because the magician goes first to show how it works, the number of items must be odd for the effect to work.

The Faro shuffle

Although called a shuffle, the Faro is a movement which controls the entire deck. A typical deck of 52 cards is split into two stacks of 26 cards. Each stack is interwoven resulting in a single stack where all even cards come from the original top and odd ones from the original bottom. The result might be the inverse according to which card is the first one to interweave.

The magician must practice the precise cut and interweave movements as to pretend to be doing an uncontrolled shuffle, while in reality controlling the position of the cards.

The simplest usage of a series of Faro shuffles is to force one card into a specific target position.

Stacked decks

Stacked decks are either partially or completely memorized and used throughout magic performances. The cards at position 10 to 15 in the Aronson Stack are the ace of clubs, ten of spades, five of hearts, two of diamonds, king of diamonds and seven of diamonds. All of them are at their exact spelling position when using the English language.

Invisible deck

The invisible deck is traditionally performed by first displaying a closed deck case. After asking a volunteer to name any card the magician opens the case and shows that there is only one card face down. It happens to be the card named by the volunteer.

The widely used Invisible Deck can not be examined after the effect is performed because its gimmick is easily perceived when the cards are manipulated. Michael Close's variation [4] does not use any gimmick and is fully examinable after the reversed card is revealed.

Invisible decks are used as finishers, to show a matching prediction. However, its usage in *Freedom of Spelling* is innovative as it forces an intermediate outcome.

B Source code

The source code for generating one *Freedom of Spelling* deck sequence can be split into generating the variables in listing 7 and generating the constraints and running the solver in listing 8.

10:12 Applying Constraint Programming to Magic

■ Listing 7 Generating the deck sequence.

```
from itertools import permutations
from typing import List, Tuple
import numpy as np
import z3

def simulate(cuts: Tuple[int], deck: Tuple[int]):
    stacks = np.array([range(cuts[0]),
                        range(cuts[0], cuts[1]),
                        range(cuts[1], cuts[2]),
                        range(cuts[2], 53)])

    return np.concatenate(stacks[[*deck]])

VALUES = ["ace", "two", "three", "four", "five", "six", "seven",
          "eight", "nine", "ten", "jack", "queen", "king"]
SUITS = ["clubs", "hearts", "spades", "diamonds"]

def retrieve_card_name(position):
    value = (position - 1) % 13
    suit = (position - 1) // 13
    return VALUES[value] + " of " + SUITS[suit]

def rules_all_cards_on_deck(all_vars: List[z3.Int]) -> z3.And:
    rules = [z3.And([card >= 1, card <= 52]) for card in all_vars]
    return z3.And(rules)

def freedom_of_spelling(all_vars: List[z3.Int], cuts: Tuple[int],
                        sequence: Tuple[int]) -> z3.Or:
    deck = simulate(cuts, sequence)
    return spelling_rules(all_vars, deck)

def spelling_rules(all_vars: List[z3.Int], deck: List[int]) -> z3.Or:
    rules = []
    for card in range(1, 53):
        name = retrieve_card_name(card)
        length = len(name.replace(" ", ""))
        original_position = int(deck[length])
        var = all_vars[card - 1]
        rules.append(var == original_position)
        rules.append(var == original_position - 1)
    return z3.Or(rules)

rules = set()
names = map(retrieve_card_name, range(1, 53))
all_vars = list(map(z3.Int, names))
```

■ **Listing 8** Generating all optimized constraints.

```
from tqdm import tqdm
from z3 import AtMost

for cut1 in tqdm(range(7, 21)):
    for cut2 in range(max(cut1, 12), 26):
        for cut3 in range(max(cut2, 25), 39):
            for sequence in permutations(range(4)):
                cuts = (cut1, cut2, cut3)
                rule = freedom_of_spelling(all_vars, cuts, sequence)
                rules.add(rule)

for position in range(1, 53):
    at_starting_point = [card == position for card in all_vars]
    rule = AtMost(*at_starting_point, 1)
    rules.add(rule)


rules.add(rules_all_cards_on_deck(all_vars))

print(len(rules))
print(z3.solve(rules))
```


Vehicle Dynamics in Pickup-And-Delivery Problems Using Electric Vehicles

Saman Ahmadi¹  

Department of Data Science and AI, Monash University, Victoria, Australia
CSIRO Data61, Canberra, Australia

Guido Tack 

Department of Data Science and AI, Monash University, Victoria, Australia

Daniel Harabor 

Department of Data Science and AI, Monash University, Victoria, Australia

Philip Kilby 

CSIRO Data61, Canberra, Australia

Abstract

Electric Vehicles (EVs) are set to replace vehicles based on internal combustion engines. Path planning and vehicle routing for EVs need to take their specific characteristics into account, such as reduced range, long charging times, and energy recuperation. This paper investigates the importance of vehicle dynamics parameters in energy models for EV routing, particularly in the Pickup-and-Delivery Problem (PDP). We use Constraint Programming (CP) technology to develop a complete PDP model with different charger technologies. We adapt realistic instances that consider vehicle dynamics parameters such as vehicle mass, road gradient and driving speed to varying degrees. The results of our experiments show that neglecting such fundamental vehicle dynamics parameters can affect the feasibility of planned routes for EVs, and fewer/shorter charging visits will be planned if we use energy-efficient paths instead of conventional shortest paths in the underlying system model.

2012 ACM Subject Classification Computing methodologies → Planning and scheduling

Keywords and phrases Electric vehicle routing, pickup-and-delivery problem, vehicle dynamics

Digital Object Identifier 10.4230/LIPIcs.CP.2021.11

Supplementary Material *Dataset:* https://bitbucket.org/s-ahmadi/pdp_ev
archived at `swb:1:dir:63b287819b65ea75ac1ee8a584e23f2c540c6b38`

Funding Research at Monash University is supported by the Australian Research Council (ARC) under grant numbers DP190100013 and DP200100025 as well as a gift from Amazon.

1 Introduction

The Pickup-and-Delivery Problem (PDP) is a well-studied problem in Constraint Programming (CP) and Operations Research. PDP is a point-to-point transport problem where a fleet of vehicles needs to serve requests for moving loads/passengers between a set of pickup and delivery points. In this problem, transit requests are known and vehicles can start and terminate their trips at particular depots. The solution to the PDP is a set of routes (one route per vehicle) that satisfies both problem objectives (e.g. shortest tours) and transport constraints (such as time windows and energy requirements).

In this study, we are interested in the PDP using Electric Vehicles (EVs). The electrification of transport systems is a well-known and efficient practice to reduce transport emissions. Compared to the conventional combustion-based vehicles, battery-powered vehicles are less

¹ Corresponding Author



© Saman Ahmadi, Guido Tack, Daniel Harabor, and Philip Kilby;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 11; pp. 11:1–11:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

dependant on fossil sources and offer higher energy efficiencies. However, limited driving range and the lengthy charging process in EVs may require substantially different routing decisions compared to conventional vehicles. As an example, in cases where the EV's available energy is not sufficient to plan point-to-point trips, charging detours need to be considered. Hence, the transport model needs to carefully track the EVs' energy levels to ensure the feasibility of planned trips by possibly adding necessary charging stops.

In transport systems with EVs, energy matters can generally be studied from two aspects: energy absorbed at charging points and energy consumed in point-to-point trips (discharging). Depending on the required energy and also the technology used at charging points, charging times may vary from a few minutes to several hours. A common practice to model the charging component of the transport system is establishing a linear relationship between energy and time [8, 16]. However, as EV chargers can adapt their power with the battery's remaining energy (slower charging rate in higher state of charge), linear charging models may not be able to correctly reflect the actual charging time needed for a specific amount of energy. To address this inaccuracy, recent studies have employed more realistic charging profiles for their transport models to accommodate various charging technology types via piece-wise linear approximations [15, 13].

Discharging (while driving) is the other aspect of energy matters in EVs. Similar to charging, transport models require an energy model that appropriately accounts for energy consumption. A common (but inaccurate) method to estimate the energy consumption of point-to-point trips is to use a fixed energy consumption rate (measuring units of energy per unit of distance/time) [11, 19]. This means that basic models assume shortest/fastest point-to-point paths to also be the most energy-efficient paths. However, there are several important parameters in vehicle dynamics that are ignored in linear consumption models, such as driving patterns, vehicle mass and road gradients. Therefore, the estimated energy costs obtained by using basic models are not accurate and there is always a high risk for planned trips to be infeasible in reality. Due to the complexity of realistically estimating the discharging energy in EVs, attempts to add some parameters of vehicle dynamics into energy models may lead to major simplifications in the energy model such as ignoring vehicle acceleration or neglecting changes in ground slope [9].

To better explain to what extent vehicle dynamics can affect the energy consumption in EVs, we solved a PDP with 14 transport demands in San Francisco as depicted in Figure 1 (Right). The trip was planned using the average (fixed rate) energy consumption of the *Peugeot iOn* as an EV with 16 kWh battery capacity and 100% initial energy level. As shown in Figure 1, the EV can transport all of the passengers to their destination using less than 100% of the energy capacity (red line). Therefore, the planned trip seems feasible with the basic (fixed-rate) energy model. But the situation changes when we recalculate the energy requirement of the planned multi-stop trip via three other energy models that take vehicle dynamics into account to varying degrees. As seen in Figure 1 (Left), the trip would require more energy when passenger weights are included (blue line). Energy consumption further increases when the ground slope is added into the energy calculation (green and yellow curves), making the last two pickups (at a distance of 120km) infeasible with more accurate energy models. Therefore, given the fact that trips can be planned for every possible initial energy level, neglecting vehicle dynamics parameters can potentially result in infeasible trips.

This paper investigates the implications of adding vehicle dynamics into the EV route planning. To this end, we establish an energy-based PDP transport model that can handle realistic charging and discharging profiles of EVs. We use CP technology to prototype a solver for the PDP that can deal with the new transport model. This allows us to conduct a

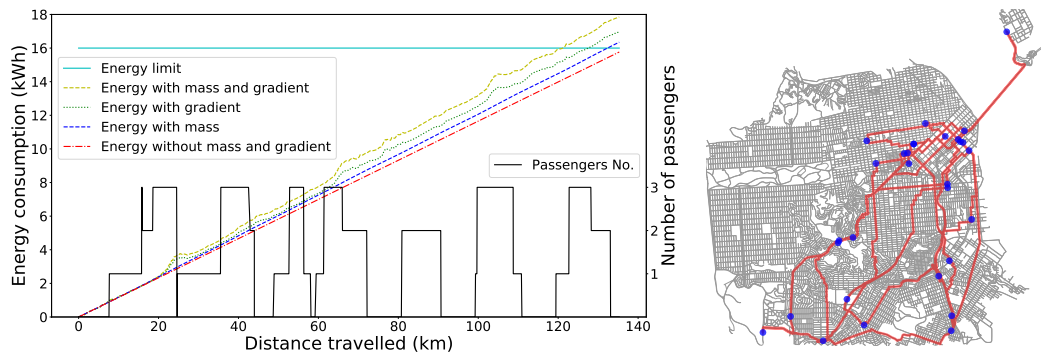


Figure 1 [Coloured] Right: A Sample multi-stop trip in San Francisco, Left: Pickup and drop-off of passengers during the trip, also energy requirement of the trip using different energy models versus the distance travelled.

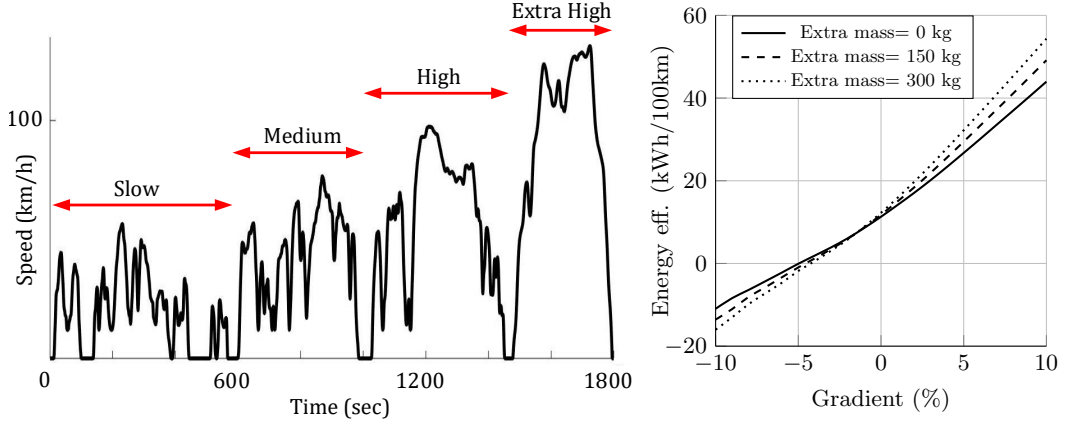
detailed study of the new model and evaluate it with a new set of realistic instances that accounts for challenging requirements of EV energy models in two scenarios: using shortest or energy-efficient point-to-point paths. The results of our experimental study show that it is crucial to use more accurate transport models to avoid the risk of seriously underestimating energy requirements. These results justify an investment into new solving technology that can handle these accurate models efficiently.

2 Energy Consumption Models

When solving routing problems for EVs, the energy requirement of the underlying point-to-point paths are determined based on an *energy model*, which determines how much energy the vehicle will consume (or indeed recuperate) when travelling from point A to point B . Energy models can differ dramatically in their complexity. A very basic model may estimate energy requirements over a path long length d by simply using an average energy consumption rate β (in Wh/100m), resulting in the simple equation $E = \beta \times d$. This simplistic energy model provides a rough approximation and does not fully take into account the main parameters in vehicle dynamics, such as road gradient, vehicle mass or acceleration. To better understand the importance of the vehicle dynamics parameters, Figure 2 (Right) shows changes in energy efficiency with different road gradients and extra mass for the *Peugeot iOn* as a test EV with an initial energy level of 70%. We use the realistic WLTP drive cycle² depicted in Figure 2 (Left). We see a slight but clear non-linear relationship between energy efficiency and road gradient. Furthermore, the figure shows that increasing vehicle mass can either increase or decrease energy efficiency depending on the gradient.

In positive gradients, the energy requirement of links increases with mass, but this is not always the case in negative gradients. EVs can potentially recover part of the kinetic energy via regenerative braking. This means that energy consumption can even be negative on negative slopes as shown in Figure 2 (Right). If the energy requirement of a link is negative (on negative slopes), increasing mass would contribute to recuperating more energy and, therefore, decreased energy consumption. Figure 2 (Right) also highlights that the amount of energy the EV can regenerate on a negative slope is much less than the energy it needs to climb up the same gradient. This difference is mainly due to the total powertrain efficiency and hybrid (mechanical+electric) braking strategy in EVs.

² Worldwide Harmonised Light Vehicles Test Procedure



■ **Figure 2** Left: Speed profiles in the WLTP cycle, Right: Energy efficiency vs. gradient and extra mass over WLTP for the *Peugeot iOn* with the average energy efficiency of 11.6 kWh/100km.

As the exact calculation of the links' energy requirement in the PDP setting is a difficult task, we build our energy models based on three main parameters in vehicle dynamics that change with transport request/location. These parameters are gradient, speed profiles (acceleration/deceleration) and extra mass. In this study, we investigate six different energy models. Our most accurate energy model is given in Eq. (1). We call this full model E_{gv}^m , since it takes gradient g , speed profiles v and extra mass m into account.

$$E_{gv}^m = ((\alpha_{1,i}m + \beta_{1,i})g^2 + (\alpha_{2,i}m + \beta_{2,i})g + (\alpha_{3,i}m + \beta_{3,i}))d \quad (1)$$

In Eq. (1), E_{gv}^m is the energy requirement (in Wh), g is the road angle ($\sin \theta$), d is the link distance (in units of 100m) and m is the extra mass (load/passenger weights in kg). Coefficients α_i (in Wh/(100m*kg)) and β_i (in Wh/100m) are parameters that depend on the selected speed profile i and also vehicle specification. These coefficients can be obtained using the relationship depicted in Figure 2 (Right) for every EV evaluated under a driving pattern. Table 1 shows α_i and β_i values for our test EV *Peugeot iOn* simulated under the speed profiles of the WLTP driving cycle (Slow, Medium, High, Extra-High) after fitting a polynomial of degree two to the operating points obtained for each speed profile (see Figure 2 (Left) for the overall WLTP pattern). We define the energy requirement of a path to be the aggregation of the energy requirements of all of its links.

We first define our *basic energy model* to be the model that just uses the EV's average energy efficiency, i.e. $E_b = \beta_3 d$. If extra mass is to be considered in the basic model, we then have $E_b^m = (\alpha_3 m + \beta_3) d$. Our second model incorporates road gradient g as an additional parameter via $E_g = (\beta_1 g^2 + \beta_2 g + \beta_3) d$. Analogously, adding extra mass to this model yields $E_g^m = ((\alpha_1 m + \beta_1)g^2 + (\alpha_2 m + \beta_2)g + (\alpha_3 m + \beta_3))d$. Given the nonlinear relationship in Figure 2 (Right) for every driving pattern, our last case accounts for both gradient g and speed v impacts on energy consumption via $E_{gv} = (\beta_{1,i}g^2 + \beta_{2,i}g + \beta_{3,i})d$. Finally, adding mass to this model yields our full energy model E_{gv}^m as in Eq. (1). Table 2 shows a summary of our energy models with and without extra mass consideration. Note that for the models with a fixed speed profile such as E_b and E_g , we use the average energy coefficients obtained for the concrete WLTP drive cycle (last profile in Table 1).

■ **Table 1** Energy specification and coefficients of the *Peugeot iOn* based on the profiles in WLTP. α in Wh/(100m*kg) and β in Wh/100m.

Vehicle details	Profile	α_1	α_2	α_3	β_1	β_2	β_3
<i>Peugeot iOn 2017</i>	Slow	0.398	0.244	0.005	315.33	264.69	12.60
Capacity: 16 kWh	Medium	0.451	0.241	0.004	381.85	262.25	10.04
Efficiency*: $\sim 11.6 \frac{\text{Wh}}{100\text{m}}$	High	0.526	0.249	0.004	511.05	259.70	10.36
Kerb weight: 1050 kg	Extra High	0.731	0.262	0.004	734.48	293.05	13.31
Overall (avg.)		0.579	0.251	0.004	536.72	272.77	11.65

■ **Table 2** Summary of energy models versus parameters of vehicle dynamics.

Model	Parameters	Without Mass	Adding Mass (m)
E_b	-	$\beta_3 d$	$+ m \alpha_3 d$
E_g	gradient	$(\beta_1 g^2 + \beta_2 g + \beta_3) d$	$+ m(\alpha_1 g^2 + \alpha_2 g + \alpha_3) d$
E_{gv}	gradient, speed	$(\beta_{1,i} g^2 + \beta_{2,i} g + \beta_{3,i}) d$	$+ m(\alpha_{1,i} g^2 + \alpha_{2,i} g + \alpha_{3,i}) d$

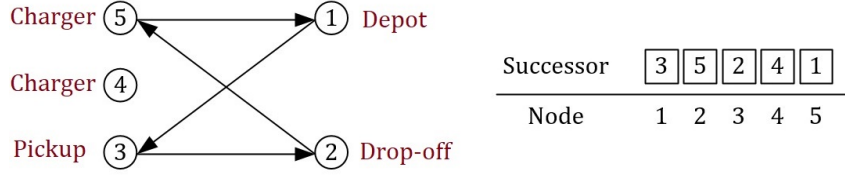
3 EV Routing Problem

This section explores the impacts of vehicle dynamics on the PDP when EVs are operated in the transport system. In this problem, EVs start and terminate trips at particular depots and transit requests are known in advance. A solution to this PDP is a set of routes (one route per vehicle) that satisfies both problem objectives (e.g. fastest/shortest tours) and fundamental constraints such as passenger time windows [18]. Furthermore, there are other considerations such as tracking the EVs' energy levels and charging times to ensure route feasibility.

The PDP model for EVs should respect the correlations between energy matters in EVs and routing constraints in the PDP. Although Mixed-Integer Programming (MIP) is a traditional modelling approach in routing problems, in this study, we use CP to design and develop our PDP model as it provides a greater degree of flexibility in the way our non-linear energy constraints are handled. For this purpose, we develop our energy-based PDP model in MiniZinc [14].

We define each possible origin/destination location in our model to be a node with at most one status from $N = \{\text{pickup}, \text{drop-off}, \text{depot}, \text{charger}\}$. This means that we model every transport request with a $(\text{pickup}, \text{drop-off})$ pair, each EV initially at a *depot* node, and each charger with at least one *charger* node per visit (multiple nodes if more than one visit is allowed). For the CP model of this study, we use the *Successor* representation to encode the EVs' trips. We explain this approach using an example shown in Figure 3. For the simple trip planned in Figure 3 (Left) and the nodes' successors (Right), we can execute the full trip by sequentially looking up each node's successor to obtain the sequence $\{1-3-2-5-1\}$, given node 1 as the depot. Node 4 is not part of the trip.

Problem objective. Following the traditional objective definition in vehicle routing problems, we aim to plan routes that are optimal in terms of total travel time, i.e., in the context of EVs, our objective is minimising both the travel time and charging time.



■ **Figure 3** Left: An example trip for the successor representation, Right: *Successor* array.

3.1 PDP Constraints

The essential step in the PDP is keeping track of the (non-negative) trip time t at non-charger nodes by

$$t[j] \geq t[i] + t_s[i] + \Delta t[i, j] \quad i \in N - \{\text{charger}\} \quad \text{and} \quad j = \text{succ}[i] \quad (2)$$

The constraint above ensures that the service time t_s of non-charger node i and also the time required to reach $\text{succ}[i]$ from node i is preserved in the trip time when the EV arrives at $\text{succ}[i]$. We assume the point-to-point travel time matrix Δt has already been pre-computed and is available as input to our model. In addition, we consider a constant service time for every node in our model (zero for depot).

In PDP with time windows, we also need to make sure that the arrival times are always within the time limit of the transport demands, i.e, for every *pickup* node we have

$$t_{\text{low}}[i] \leq t[i] \leq t_{\text{up}}[i] \quad i \in \text{pickup} \quad (3)$$

where t_{low} and t_{up} are the lower and upper time limits of pickup nodes respectively. In order to prevent long trips for every individual transport request, we limit the the travel time for transport requests by

$$0 \leq t[j] - t[i] \leq \lambda \times \Delta t[i, j] \quad (i, j) \in (\text{pickup}, \text{dropoff}) \quad (4)$$

where $\lambda > 1$ is a constant factor that scales the time required to traverse the direct route between pickup and drop-off nodes. For example, setting $\lambda = 2$ means that the total time each transport request spends in our EV is at most two times longer than its direct route. The constraint above also makes sure that drop-off occurs after pickup.

Respecting the vehicle capacity is another essential step in PDP system models. That is, given the vehicle capacity C , we need to make sure that the EV transfers at most C passengers in every point-to-point trip:

$$0 \leq u[i] \leq C \quad i \in N \quad (5)$$

where the variable $u[i]$ represents the vehicle utilisation (number of passengers) when the EV departs from node i . Meanwhile, in order to accurately track the EVs' loads, we have

$$u[j] = u[i] + \Delta u[j] \quad i \in N \quad \text{and} \quad j = \text{succ}[i] \quad (6)$$

where $\Delta u[j]$ indicates the utilisation change at the successor node j (the node that will be visited after node i in the trip). The value of Δu is positive at pickup nodes, negative at drop-offs and zero at other nodes. Note that since this parameter will be part of our energy calculations, we use equality ($=$) in the constraint above to always have the exact utilisation value (and more accurate energy estimates respectively) at departure. Similar to

the point-to-point travel time array Δt , the Δu array is pre-computed using the problem specification. We can also add a constraint for vehicle utilisation at charger points. If we do not want to have any passenger on board while charging, we simply set

$$u[i] = 0 \quad i \in \text{charger} \quad (7)$$

As an extra constraint, we use the global CP constraint `subcircuit` to create a set of circuits (trips) through our *Successor* array. Since visiting all charging points is not necessary, this constraint allows us to plan trips without charging visits. Figure 4 (Right) depicts sample solution tours using this constraint when EVs are allowed to return to any of the depots.

3.2 Energy Constraints

We now present energy constraints needed for appropriate energy tracking in the PDP for EVs. We again use the trip sequence available in the *Successor* array for our energy tracking approach. For every *demand* node, the available energy of the EV at its successor node is estimated using the following constraint.

$$e[j] = e[i] - \Delta e[i, j, k] \quad i \in \{\text{pickup}, \text{dropoff}\} \quad \text{and} \quad j = \text{succ}[i] \quad \text{and} \quad k = u[i] \quad (8)$$

Where $e[j]$ is the arrival energy level at successor node j and the 3-dimensional array Δe represents the energy requirement of traversing the path between a demand location i and its successor $\text{succ}[i]$, with $u[i]$ passengers in the EV. Similarly, for the *depot* nodes we track energy consumption by

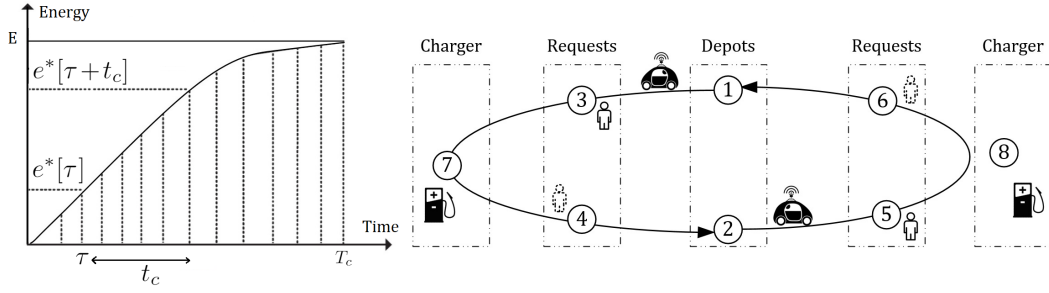
$$e[j] = e_{\text{init}}[i] - \Delta e[i, j, k] \quad i \in \text{depot} \quad \text{and} \quad j = \text{succ}[i] \quad \text{and} \quad k = u[i] \quad (9)$$

where $e_{\text{init}}[i]$ is the initial energy level of our EVs at their designated depot node i (zero for non-depot nodes). Furthermore, as charging time depends on the available energy at nodes, we use equality ($=$) in the constraints above to always have the exact energy values for our charging time calculations. In other words, by using equality, we do not allow the model to set arbitrarily low arrival energy values to benefit from higher charging rates (in lower energy levels) and consequently shorter charge times. It is worth mentioning that we define the range of our energy-based decision variables to be $[0, E]$ where E is the energy capacity of the EV. This means critical cases (running out of energy and overcharging) are already considered in our CP model, i.e., for every node i we have $0 \leq e[i] \leq E$.

We can see that adding mass (vehicle utilisation) to the PDP model increases the system complexity, since the number of passengers in the vehicle at any location is a decision variable, and therefore the energy requirement Δe may no longer be constant, depending on whether the energy model does take mass into account. Nonetheless, we can use any of the energy models presented in Section 2 to calculate the point-to-point energy requirements in Δe given the fact that the upper bound on extra mass is known (the number of passengers $u[i]$ is at most C). In the next section, we measure the impacts of each model on the planned routes.

We now explain our charging constraints. To model charging profiles without linear piece-wise approximation, we map each profile into an array indexed by time. Figure 4 (Left) depicts how we discretise a sample charging profile based on a time unit. In our model, $e^*[\tau]$ represents the amount of energy charged in a fully discharged battery if the EV has been in the charging station for τ units of time. Since EVs are not allowed to have negative energy levels (or we always have $0 \leq e^*[\tau]$), we use τ as an offset for charging time t_c . This means if the EV arrives at a charging point with energy $e_1^*[\tau_1]$ and departs with energy $e_2^*[\tau_2]$, we have $t_c = \tau_2 - \tau_1$ as the charging time. Therefore, for every visited charging node i we have

$$e^*[\tau[i]] \leq e[i] < e^*[\tau[i] + 1] \quad i \in \text{charger} \quad \text{and} \quad \tau[i] \in [0, T_c) \quad (10)$$



■ **Figure 4** Left: Charging profile, Right: Sample trips using the `subcircuit` constraint.

where $e[i]$ is the EV energy at arrival and $\tau[i]$ is the corresponding time offset for charger node i . T_c is also the maximum charging time to get charged for the full energy capacity E . Note that the constraint above looks up the closest (mapped) value to $e[i]$ such that the actual charging time always falls in the interval $[\tau, \tau + 1)$. This means the discretisation approach may lead to overestimating the charging time by at most one unit of time. Now, given $e^*[\tau + t_c]$ as the energy of the EV after visiting a charging node, we can set a constraint on the energy of the EV when it arrives at a successor node via a charging node.

$$e[j] = e^*[\tau[i] + t_c[i]] - \Delta e[i, j, k] \quad i \in \text{charger} \quad \text{and} \quad j = \text{succ}[i] \quad \text{and} \quad k = u[i] \quad (11)$$

The constraint above ensures that the energy consumption is appropriately tracked after every charging visit, but we need to make sure that the charging time is also incorporated in the total trip time. To this end, we first limit the charging time per charging node by

$$0 \leq t_c[i] \leq T_c - \tau[i] \quad i \in \text{charger} \quad (12)$$

where t_c is the charging time and T_c is the maximum charging time (time needed to get charged from 0 to 100% energy level). The constraint above also enforces that discharging in charging stations is not allowed as the charging time is always non-negative.

Now we finally define the lower arrival time to successor nodes via charging nodes to be

$$t[j] \geq t[i] + t_c[i] + t_s[i] + \Delta t[i, j] \quad i \in \text{charger} \quad \text{and} \quad j = \text{succ}[i] \quad (13)$$

where $t[i]$ is the time the EV arrives at the charging point and $t[j]$ is the time it is at the successor node. Furthermore, t_s is the service time spent at the charging node. Note that since different charging technologies have different charging profiles, we store our mapped energy values in a 2-dimensional array where the other dimension determines the charging type, i.e., we have $e^*[\tau[i], \epsilon[i]]$ where $\epsilon[i]$ would be the charging type of node i . Each charging node in our PDP model can handle one charging visit at a time. If some or all charger locations can handle more than one charging visit at a time, we need to create multiple charging nodes for these charging locations, each capable of handling one visit. In this case, the model may need additional constraints for charger scheduling.

Finally, the objective is to minimise the total driving time and charging time.

$$\text{Minimise} \quad \left(\sum_{i \in N} \Delta t[i, \text{succ}[i]] + \sum_{i \in \text{char}} t_c[i] \right) \quad (14)$$

4 Benchmark Setup

Transport models for EVs require accurate energy estimates on the underlying point-to-point paths. In order to examine our PDP model under realistic scenarios, a set of instances with all the energy measures (point-to-point energy requirements for different energy models) is required. To this end, following the strategy used in [4], we use the GPS traces from Uber Technologies Inc.³ to extract random realistic trips. The data file includes the GPS logs of more than 20,000 ride-sharing trips in San Francisco (CA, USA) over one week. Furthermore, choosing San Francisco as the test city allows us to better investigate the significance of gradient as one of the main parameters of vehicle dynamics.

Previous works on routing problems for EVs have used (partially) synthetic datasets and/or simplified energy models. For example, the well-known Solomon vehicle routing benchmark [17] is a commonly used synthetic dataset that has been adapted to EVs in [10, 16]. Since parameters in synthetic datasets are not fully realistic, they cannot perfectly reflect all the real challenges in EVs, especially energy-related parameters such as non-linear charging profiles and vehicle dynamics. Given the importance of energy parameters in EV routing problems, some recent studies have tried to establish more realistic datasets by respecting the relationships between distance, time and energy [9, 13, 4]. Nonetheless, these datasets are still not complete enough to be used in our complete energy-based PDP model.

We now explain our strategy to generate random test cases from the Uber ride-sharing dataset. Each line of the data file is in the following format:

<trip ID> <timestamp> <latitude> <longitude>

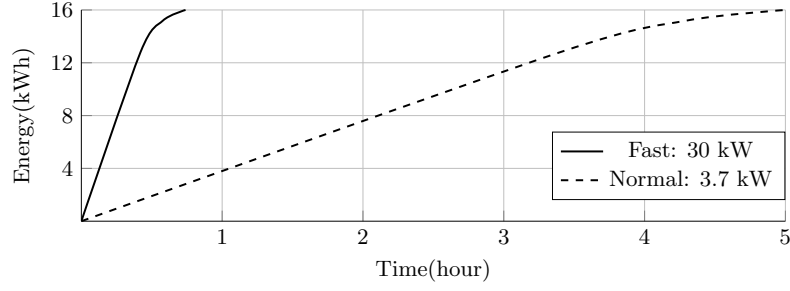
Every trip ID in the data file can be found in two lines, one for pickup and another for the corresponding drop-off. We rebuild every point-to-point transport demand using the pickup and drop-off locations (latitude, longitude). The travelled distance of trip IDs ranges from 100m to 15km, but to better analyse the energy matters in the PDP with EVs, we randomly pick trips of 8km and longer. The Uber GPS log file contains all the necessary data for traditional route planning models such as time and transport origin and destination, but part of the input data to our system model still needs to be determined with extra considerations.

Time windows. For each selected trip ID, we pick the timestamp of the pickup entry (the one with an earlier timestamp) as the desired pickup time. We then consider a 15-minute time window for every transport demand at pickup. As an example, if the actual pickup time of a transport request in the data file is t_{pu} , its corresponding time window is considered to be $[t_{pu} - 7.5min, t_{pu} + 7.5min]$. We set our time scaling factor λ to 1.5, meaning that the trip time of each individual transport request is at most 50% longer than its direct path. We also set 30 seconds as service time for every non-depot node.

Extra stop locations. We assume the EVs' depot to be a location in the city area of San Francisco. We also choose five charging locations from an online service⁴: three normal and two fast-charge points (one of the normal chargers at the depot). In addition, EVs are expected to return to the depot after serving transport demands. To prevent frequent charging detours, we limit EVs to at most one charging visit in each trip.

³ <https://github.com/dima42/uber-gps-analysis/tree/master/gpsdata>

⁴ <https://www.plugshare.com/>



■ **Figure 5** Normal and Fast charge profiles for *Peugeot iOn* with 16 kWh battery capacity.

Test EV. We choose our test EV to be the *Peugeot iOn*, one of the commonly used EVs in the literature with a vehicle capacity of four passengers and an energy capacity of 16 kWh [11]. We consider 75 kg for each passenger’s weight⁵ in our energy calculation. Figure 5 depicts the actual non-linear charging profiles of two charging technologies (normal and fast) available for the test vehicle of this study with the maximum charging time of five hours.

Energy coefficients. For the test EV of this study, we used an advanced (open source) EV simulator called ADVISOR (ADvance VehIcle SimulatOR)⁶ to learn our energy models and set coefficients α_i and β_i in Section 2. This software is developed on the engineering platform MATLAB and is enriched with complete powertrain models. The model of each component (such as the battery, electric machine, etc.) incorporates parameters such as all detailed equations of vehicle dynamics, temperature profiles, efficiency maps, auxiliary loads or even warm/cold start [1, 12]. Therefore, the selected simulator provides more accurate estimates on energy requirements of EVs under a variety of realistic scenarios, such as transporting different numbers of passengers on a road with a non-zero slope).

EV considerations. We assume that our EVs can use their full energy capacity, i.e., their energy level can be 0–100% and there is no limit for them at the end of the trips. Moreover, we assume that our EVs start their trips with 100% initial energy level.

Underlying graph. We extract the San Francisco road network from OpenStreetMap using the Python package OSMnx [3] and enrich the graph edges of the road network with elevation and speed data using the *Bing*, *Mapbox*, *Here* and *TomTom* APIs⁷.

Point-to-point paths. Given the road network of San Francisco as our underlying graph, we propose two types of paths for our experimental analysis:

1. *Shortest path:* for our base approach, we consider all point-to-point paths in our problem to be shortest path. For this purpose, we compute our first set of paths using Dijkstra’s algorithm [6] for any (origin,destination) pair in the instance.
2. *Energy-efficient path:* for our second approach, we optimise all point-to-point paths for their energy consumption, that is, instead of solving the underlying graph for *time*, we are interested in a path that offers the lowest energy consumption. As the energy requirement of links of the graph can be negative (in negative slopes), we use an adapted version of the Bellman-Ford algorithm [2, 7] to calculate point-to-point energy-optimal paths. Our adapted version uses the most accurate energy model presented in Section 2 (Eq. (1)) and takes the battery limits into account.

⁵ Based on European Directive 95/48/EC

⁶ <http://adv-vehicle-sim.sourceforge.net/>

⁷ www.bing.com; www.mapbox.com; developer.here.com; developer.tomtom.com

Time and energy arrays. The additional required information includes the time (Δt array) and energy (Δe array) requirements of all point-to-point paths. So far we have obtained two sets of paths for each problem instance. In the next step, we need to determine our time and energy arrays (resp. Δt and Δe) for each set. To this end, we take the resulting paths and compute their time simply via converting the *distance* attribute of the links of the paths to time using an average speed. For their energy requirement, we use all of our four energy models presented in Section 2 and generate four energy arrays for each set of paths. Note that we do not change the optimum paths in each set, but we recalculate the energy requirement of the paths each time with a different approach. Creating separate energy arrays allows us to better investigate the impacts of vehicle dynamics on our planned routes. We also set time and energy units to 10 Wh and 30 seconds respectively.

Instance format. Each instance is presented in the format of $\langle uv - t - n \rangle$ where v is the number of EVs, t is the number of transport requests and n is the instance variant. For example, the instance $\langle u1 - 10 - 2 \rangle$ is our second instance with 10 transport requests and one vehicle. We arbitrarily keep the number of the transport requests small. As finding the optimal solution to each instance is necessary for our energy analysis, we avoid generating very large and difficult instances. Each instance consists of the required input for our energy-based PDP model such as time windows, utilisation at each pickup, time and energy arrays, charger profiles and energy/time upper bounds. Our instances are publicly available ⁸.

5 Results and Discussion

We developed our model in MiniZinc and evaluated that using the CP solver Chuffed [5] as the back-end optimiser with the *free search* flag. We found MiniZinc MIP solvers slower for our CP model over the instances. For each scenario of point-to-point paths, we solved all of the instances to optimality using different energy models (resp. energy arrays). The proposed solver could solve our difficult instances with 20 demands, five chargers and two vehicles in a 90-minute timeout on a machine with an Intel Core i7-10850H running at 2.7 GHz and with 32 GB of RAM. Regarding the computational effort, we found handling charger nodes the main solving challenge as the charging time can be even larger than the trip time (five hours for normal chargers). As we will see later in this section, the solver rarely plans trips that include visits to normal chargers. As a further optimisation, we could reduce the runtime by introducing a subset of charger nodes in multiple runs of the model to improve the objective upper bound. For example, by removing the normal slow chargers from the list of stops we could solve the instances much faster (usually in less than five minutes). We also found the problem much more challenging for the solver with increased number of transport requests and vehicles. The standard solver was unable to optimally solve our larger instance with 30 demands and three vehicles in the time limit. It is worth highlighting that we did not notice a major difference in the runtime of the models with mass consideration (models without mass consideration were solved slightly faster).

Tables 3 and 4 present the results of solving the PDP for the designed instances of this study in two different scenarios. Table 3 shows the results for the first scenario with shortest point-to-point paths, and Table 4 presents the results for the second scenario with energy-optimum paths as an alternative. In the both tables, attribute E_b denotes that the transport model has been evaluated with the *basic* energy model for point-to-point energy calculations.

⁸ <https://bitbucket.org/s-ahmadi>

Similarly, E_g and E_{gv} indicate the use of additional parameters *gradient* and *speed* in the energy models. The results are shown without (left columns) and with (right columns) taking extra mass into account. Finally, the results for our full energy model are indicated with E_{gv}^m with all parameters considered. We again emphasise that the point-to-point paths in each scenario are the same for all of the models and only the energy calculation method is different. In the energy-efficient paths scenario, the point-to-point paths were pre-computed based on our full energy model E_{gv}^m . Our studied parameters consist of the value of the total driving time (t_{dr}), total charging time (t_{ch}), energy consumed while driving (e_{dr}), energy charged at charging nodes (e_{ch}), type of chargers used (ϵ), the ratio of the difference in routes ($\phi_r = \Delta R/R_b$) with the routes of E_b as the base, and the energy miscalculation ratio $\phi_e = \Delta E/E_{gv}^m$ with E_{gv}^m as the base (complete) energy model. We calculate the ratio $\Delta R/R_b$ by comparing the optimum *Successor* array against that of the base case obtained for E_b . In addition, we define the energy error ΔE to be the difference between energy values in E_{gv}^m and other models (E_b , E_g etc.).

We start our analysis with the results shown in Table 3. According to the route difference ratio, routes may vary up to 20% if we do not opt for the basic energy model in point-to-point energy calculations. Furthermore, we can see that in all cases, the energy consumption of tours increases when more accurate energy models are considered for our energy array. This means the basic energy model is the least accurate model, underestimating the trips' energy requirement in all of the instances. Nonetheless, the table shows that the mass-based energy model E_b^m is still not as accurate as the gradient-based models E_g or E_{gv} with both mass and speed added. Therefore, we can conclude that the gradient in E_g has more impact on energy consumption than mass in E_b^m .

The results in Table 3 show that taking driving patterns into consideration (models E_{gv} and E_{gv}^m) further increases the trips' energy requirement. The main reason is that there are generally more low-speed links on inner-city trips than other speed profiles, and low-speed links require more energy than medium or high-speed profiles (see energy coefficients in Table 1). This inefficiency is technically rooted in energy loss via frequent stop-go patterns in inner-city trips and little energy recuperation at low speeds.

In several cases, we see that serving transport requests with more accurate energy models is only feasible if a charging detour is planned. In other words, routes planned based on less accurate energy models can potentially be infeasible in reality as they might not consider any charging detour at critical energy levels. In the *u1-10-1* and *u2-20-3* instances, for example, we have a case where only our full energy model E_{gv}^m plans a route with a charging detour while other models assume the initial energy of the EVs is sufficient for the entire trip. These instances highlight another important observation. Although we already concluded that the gradient and driving speed have more impact on optimal routes than mass, as our full model E_{gv}^m considers all parameters including extra mass, we can see that neglecting mass may lead to planning infeasible routes even with relatively accurate energy estimates via E_{gv} .

Our next observation is that the objective value ($t_{dr} + t_{ch}$) changes when a different route is planned (non-zero ϕ_r) to meet the energy limits and also every time the models introduce a charging detour to the planned routes. For example, a 10.5-minute charging time is required to fast charge the EV for 5.19 kWh in instance *u1-10-5* using our full model E_{gv}^m . Note that the basic model E_b in this instance also plans a charging detour, but its estimated charging time is almost half (4.5-minute) of the charging time planned via E_{gv}^m . This is mainly because of around 3.1 kWh underestimation of energy in E_b . This significant difference in charging times means that having a charging visit in our plan does not necessarily guarantee the feasibility of trips. Therefore, having an accurate energy model is also vital for the correct calculation of charging times.

Table 4 shows the results when energy-optimum paths are used for the pre-computation of the time and energy arrays. A quick look over the values reminds us of the pattern we observed in Table 3 for time and energy values. We see larger time and energy values when more accurate models are employed in our PDP transport system, but there are some meaningful differences when we compare them with the results in Table 3. Firstly, energy efficient paths result in longer trips (larger objective values). This is mainly because of the fact that energy-efficient point-to-point paths are normally longer than shortest paths. Secondly, compared to the routes with shortest paths (Table 3), routes planned with energy-efficient paths using the E_b or E_b^m model consume more energy. The reason is, again, related to having longer paths and consequently larger energy values for point-to-point paths via simple distance based energy models. Nonetheless, if we compare the results of the tables for E_g and E_{gv} , the case is reversed and the impact of the gradient is revealed. Although we have not defined any explicit energy objective for our PDP, we can see that energy-optimum paths are actually contributing to lower energy consumption in all of our instances via our more accurate models E_g and E_{gv} , while making sure that all the transport requests have also been served. We can see the same pattern for the gradient and speed-based models with extra mass consideration, i.e., E_g^m and E_{gv}^m .

The results in Table 4 also show that energy-optimum paths via the E_g and E_{gv} models (and their mass-added variants) also contribute to less charging time in our instances. Interestingly, there are even several cases where no charging visit is required if we opt for energy optimum paths over traditional shortest paths. In instance *u2-20-5* using the accurate model E_{gv}^m , for example, we can avoid a charging detour for 2.91 kWh extra energy in six minutes. Comparing the time and energy values of the instance for E_{gv}^m from the tables, we notice that we can save 1.27 kWh by planing a route that is even 0.5 minutes faster than the traditional time-efficient route with a charging detour. It is worth mentioning that there is always a trade-off between total trip time and energy consumption. We can see that trips planned using the efficient paths can be more energy-efficient and at the same time slightly longer than trips obtained by shortest paths on average. Nevertheless, we can deduce that fewer/shorter charging visits are required via more accurate models if energy optimum paths are chosen and all transport request are satisfied.

The results in both tables show that the energy error ratio in an instance can be as big as 16.3% when using the basic energy model E_b . For our gradient and speed-based energy models E_g and E_{gv} the error ratios are at most 8.0% and 4.9% respectively over the instances. We also compared the average underestimation of energy in both tables for the instances with no change in the planned route (zero ϕ_r), e.g. instance *u1-10-6*. For this instance, we have a maximum error ratio of 11.0% in Table 3 but a smaller error in Table 4 for energy efficient paths (7.7% for model E_b). We see that the energy error ratio of using the shortest path is about 40% larger than that of using the energy-optimum path.

Tables 3, 4 also show the charger type used at any charging visit. According to the results, although we have more normal chargers than fast chargers in our instances, normal chargers are barely used in our PDP routes and fast chargers are preferred in almost all of the charging detours. In particular, only one of the charging visits in each scenario (Tables 3,4) uses the normal type. The likely reason for this choice is our objective to minimise the total travel time (including charging time) knowing that normal chargers are at least four times slower than fast chargers (see Figure 5). Therefore, slow chargers with low energy rates can potentially be removed from the charging nodes of transport models with EVs, reducing the routing complexity and runtime.

■ **Table 3** Experiment results using the shortest point-to-point paths, for every energy model (E.) with and without mass consideration in Δe . The results include driving time (t_{dr}) in minutes, charge time (t_{ch}) in minutes, energy consumed while driving (e_{dr}) in kWh, energy recharged at charger points (e_{ch}) in kWh, charger type used (ϵ) from {normal/fast}, route difference ratio (ϕ_r) and energy error (ϕ_e).

Instance	Without Mass Consideration								With Mass Consideration							
	E.	t_{dr} min	t_{ch} min	e_{dr} kWh	e_{ch} kWh	ϵ n,f	ϕ_r %	ϕ_e %	E.	t_{dr} min	t_{ch} min	e_{dr} kWh	e_{ch} kWh	ϵ n,f	ϕ_r %	ϕ_e %
u1-10-1	E_b	153.5	0.0	13.89	0.00	-	-	14.2	E_b^m	153.5	0.0	14.24	0.00	-	0.0	12.0
	E_g	153.5	0.0	15.11	0.00	-	0.0	6.7	E_g^m	153.5	0.0	15.67	0.00	-	0.0	3.2
	E_{gv}	153.5	0.0	15.60	0.00	-	0.0	3.6	E_{gv}^m	153.0	1.0	16.19	0.36	f	7.7	-
u1-10-2	E_b	160.5	0.0	14.59	0.00	-	-	13.9	E_b^m	160.5	0.0	14.88	0.00	-	7.7	12.2
	E_g	160.5	0.0	15.65	0.00	-	11.5	7.7	E_g^m	161.0	1.5	16.27	0.38	f	15.4	4.0
	E_{gv}	161.5	1.5	16.38	0.60	f	15.4	3.4	E_{gv}^m	161.5	2.5	16.95	1.06	f	15.4	-
u1-10-3	E_b	161.5	0.0	14.63	0.00	-	-	15.3	E_b^m	161.5	0.0	15.04	0.00	-	0.0	12.9
	E_g	161.5	0.0	15.93	0.00	-	0.0	7.8	E_g^m	163.5	2.0	16.78	0.83	f	7.7	2.8
	E_{gv}	163.5	1.5	16.59	0.76	f	7.7	3.9	E_{gv}^m	163.5	3.0	17.27	1.34	f	7.7	-
u1-10-4	E_b	173.0	0.0	15.61	0.00	-	-	14.2	E_b^m	173.0	0.0	15.98	0.00	-	0.0	12.2
	E_g	173.0	2.5	17.01	1.16	f	7.7	6.5	E_g^m	173.0	3.5	17.51	1.53	f	7.7	3.7
	E_{gv}	173.0	3.5	17.68	1.68	f	7.7	2.8	E_{gv}^m	173.0	5.0	18.19	2.31	f	7.7	-
u1-10-5	E_b	198.5	4.5	18.02	2.27	f	-	14.7	E_b^m	198.5	5.5	18.42	2.66	f	0.0	12.8
	E_g	198.5	8.5	19.94	4.03	f	0.0	5.6	E_g^m	198.5	10.0	20.42	4.42	f	0.0	3.3
	E_{gv}	198.5	10.0	20.53	4.55	f	0.0	2.8	E_{gv}^m	199.5	10.5	21.12	5.19	f	11.5	-
u1-10-6	E_b	205.0	5.5	18.48	2.67	f	-	11.0	E_b^m	205.0	6.5	18.90	3.08	f	0.0	9.0
	E_g	205.0	7.0	19.39	3.43	f	0.0	6.7	E_g^m	205.0	8.0	19.88	3.88	f	0.0	4.3
	E_{gv}	205.0	8.5	20.23	4.28	f	0.0	2.6	E_{gv}^m	205.0	10.0	20.77	5.00	f	0.0	-
u1-14-1	E_b	173.0	0.0	15.74	0.00	-	-	15.9	E_b^m	175.5	1.5	16.54	0.63	f	20.6	11.6
	E_g	175.5	3.0	17.23	1.48	f	14.7	7.9	E_g^m	175.5	5.0	18.11	2.35	f	14.7	3.2
	E_{gv}	175.5	4.0	17.81	1.90	f	14.7	4.8	E_{gv}^m	175.5	5.5	18.71	2.78	f	5.8	-
u2-20-1	E_b	283.5	0.0	25.82	0.00	-	-	16.3	E_b^m	283.5	0.0	26.58	0.00	-	0.0	13.7
	E_g	283.5	0.0	28.78	0.00	-	0.0	6.6	E_g^m	284.0	0.0	30.10	0.00	-	14.9	2.3
	E_{gv}	283.5	0.0	29.54	0.00	-	0.0	4.1	E_{gv}^m	284.5	0.0	30.81	0.00	-	6.4	-
u2-20-2	E_b	290.5	0.0	26.16	0.00	-	-	14.5	E_b^m	290.5	0.0	26.93	0.00	-	4.3	12.0
	E_g	290.5	0.0	28.68	0.00	-	0.0	6.3	E_g^m	290.5	0.0	29.67	0.00	-	4.3	3.1
	E_{gv}	290.5	0.0	29.61	0.00	-	4.3	3.3	E_{gv}^m	290.5	0.0	30.61	0.00	-	0.0	-
u2-20-3	E_b	301.5	0.0	27.50	0.00	-	-	14.5	E_b^m	301.5	0.0	28.22	0.00	-	0.0	12.3
	E_g	301.5	0.0	30.39	0.00	-	0.0	5.6	E_g^m	301.5	0.0	31.58	0.00	-	0.0	1.8
	E_{gv}	301.5	0.0	31.21	0.00	-	0.0	3.0	E_{gv}^m	302.0	3.0	32.18	0.68	f	12.8	-
u2-20-4	E_b	303.5	0.0	27.54	0.00	-	-	14.5	E_b^m	303.5	0.0	28.36	0.00	-	0.0	13.5
	E_g	304.5	0.0	30.70	0.00	-	14.9	6.3	E_g^m	305.0	0.0	31.91	0.00	-	4.3	2.6
	E_{gv}	305.0	0.0	31.59	0.00	-	8.5	3.6	E_{gv}^m	303.5	3.5	32.78	1.68	f	14.9	-
u2-20-5	E_b	308.5	0.0	27.98	0.00	-	-	15.7	E_b^m	308.5	0.0	28.71	0.00	-	0.0	12.5
	E_g	308.5	0.0	30.74	0.00	-	0.0	6.3	E_g^m	309.0	1.0	31.95	0.07	n	4.3	2.7
	E_{gv}	308.5	0.0	31.47	0.00	-	0.0	4.1	E_{gv}^m	310.5	6.0	32.82	2.91	f	17.0	-
u2-20-6	E_b	322.5	0.0	29.06	0.00	-	-	15.6	E_b^m	322.5	0.0	28.36	0.00	-	0.0	13.5
	E_g	323.5	0.0	31.73	0.00	-	12.8	7.8	E_g^m	324.0	2.5	33.02	1.23	f	19.1	4.1
	E_{gv}	324.0	2.5	32.84	1.18	f	19.1	4.6	E_{gv}^m	324.5	5.5	34.43	2.47	f	14.9	-

■ **Table 4** Experiment results using the lowest energy point-to-point paths, for every energy model (E.) with and without mass consideration in Δe . The results include driving time (t_{dr}) in minutes, charge time (t_{ch}) in minutes, energy consumed while driving (e_{dr}) in kWh, energy recharged at charger points (e_{ch}) in kWh, charger type used (ϵ) from {normal/fast}, route difference ratio (ϕ_r) and energy error (ϕ_e).

	Without Mass Consideration								With Mass Consideration							
Instance	E.	t_{dr} min	t_{ch} min	e_{dr} kWh	e_{ch} kWh	ϵ n,f	ϕ_r %	ϕ_e %	E.	t_{dr} min	t_{ch} min	e_{dr} kWh	e_{ch} kWh	ϵ n,f	ϕ_r %	ϕ_e %
u1-10-1	E_b	155.5	0.0	14.02	0.00	-	-	11.7	E_b^m	155.5	0.0	14.40	0.00	-	0.0	9.3
	E_g	155.5	0.0	14.74	0.00	-	0.0	7.1	E_g^m	155.5	0.0	15.27	0.00	-	0.0	3.8
	E_{gv}	155.5	0.0	15.31	0.00	-	0.0	3.5	E_{gv}^m	155.5	0.0	15.87	0.00	-	0.0	-
u1-10-2	E_b	162.5	0.0	14.74	0.00	-	-	12.0	E_b^m	162.5	0.0	15.07	0.00	-	0.0	10.0
	E_g	162.5	0.0	15.41	0.00	-	0.0	8.0	E_g^m	162.5	0.0	15.94	0.00	-	0.0	4.8
	E_{gv}	162.5	0.5	16.02	0.03	n	7.7	4.3	E_{gv}^m	163.5	2.0	16.74	0.86	f	7.7	-
u1-10-3	E_b	162.0	0.0	14.68	0.00	-	-	14.45	E_b^m	162.0	0.0	15.11	0.00	-	0.0	11.9
	E_g	162.0	0.0	15.79	0.00	-	0.0	8.0	E_g^m	164.0	1.5	16.61	0.73	f	7.7	3.2
	E_{gv}	164.0	1.5	16.56	0.75	f	19.2	3.5	E_{gv}^m	164.0	2.5	17.16	1.27	f	7.7	-
u1-10-4	E_b	176.0	0.0	15.86	0.00	-	-	10.2	E_b^m	176.5	1.0	16.29	0.37	-	7.7	7.8
	E_g	176.5	1.5	16.55	0.73	f	7.7	6.3	E_g^m	176.5	2.5	16.99	1.07	f	7.7	3.8
	E_{gv}	176.5	3.0	17.22	1.46	f	7.7	2.5	E_{gv}^m	176.5	3.5	17.67	1.76	f	7.7	-
u1-10-5	E_b	201.5	4.5	18.11	2.20	f	-	11.3	E_b^m	201.5	5.0	18.51	2.54	f	0.0	9.3
	E_g	201.5	7.0	19.26	3.40	f	11.5	5.6	E_g^m	201.5	8.0	19.75	3.84	f	11.5	3.2
	E_{gv}	201.5	8.0	19.89	4.05	f	11.5	2.5	E_{gv}^m	201.5	9.0	20.41	4.51	f	11.5	-
u1-10-6	E_b	208.0	6.0	18.83	2.84	f	-	7.7	E_b^m	208.0	6.5	19.25	3.25	f	0.0	5.7
	E_g	208.0	7.0	19.31	3.37	f	0.0	5.4	E_g^m	208.0	7.5	19.78	3.81	f	0.0	3.1
	E_{gv}	208.0	8.0	19.90	4.05	f	0.0	2.5	E_{gv}^m	208.0	9.0	20.41	4.51	f	0.0	-
u1-14-1	E_b	177.5	1.0	16.04	0.05	-	-	12.4	E_b^m	179.5	2.0	16.80	0.97	f	17.6	8.2
	E_g	179.5	2.5	16.96	1.05	f	8.8	7.4	E_g^m	179.5	4.0	17.77	1.90	f	17.6	2.9
	E_{gv}	179.5	3.5	17.41	1.60	f	17.6	4.9	E_{gv}^m	179.5	5.0	18.31	2.50	f	8.8	-
u2-20-1	E_b	286.5	0.0	25.99	0.00	-	-	13.7	E_b^m	286.5	0.00	26.77	0.00	-	0.0	11.1
	E_g	286.5	0.0	28.02	0.00	-	0.0	7.0	E_g^m	286.5	0.00	29.06	0.00	-	0.0	3.5
	E_{gv}	286.5	0.0	28.94	0.00	-	0.0	3.9	E_{gv}^m	287.5	0.00	30.12	0.00	-	6.4	-
u2-20-2	E_b	291.5	0.0	26.33	0.00	-	-	12.4	E_b^m	292.5	0.0	27.20	0.0	-	4.3	9.5
	E_g	293.5	0.0	28.00	0.00	-	8.5	6.8	E_g^m	293.5	0.0	28.92	0.0	-	8.5	3.7
	E_{gv}	293.5	0.0	29.04	0.00	-	8.5	3.3	E_{gv}^m	293.5	0.0	30.04	0.0	-	8.5	-
u2-20-3	E_b	305.5	0.0	27.86	0.00	-	-	11.6	E_b^m	305.5	0.0	28.58	0.00	-	0.0	9.4
	E_g	305.5	0.0	29.37	0.00	-	0.0	6.9	E_g^m	305.5	0.0	30.45	0.00	-	0.0	3.4
	E_{gv}	305.5	0.0	30.37	0.00	-	0.0	3.7	E_{gv}^m	306.0	0.0	31.53	0.00	-	8.5	-
u2-20-4	E_b	304.5	0.0	27.59	0.00	-	-	13.9	E_b^m	304.5	0.0	28.44	0.00	-	0.0	11.2
	E_g	304.5	0.0	29.72	0.00	-	0.0	7.2	E_g^m	304.5	1.5	30.94	0.61	f	8.5	3.4
	E_{gv}	304.5	1.0	30.75	0.29	f	8.5	4.0	E_{gv}^m	304.5	2.5	32.03	1.11	f	8.5	-
u2-20-5	E_b	307.0	0.0	27.81	0.00	-	-	11.9	E_b^m	307.0	0.0	28.60	0.00	-	0.0	9.4
	E_g	307.0	0.0	29.25	0.00	-	0.0	7.3	E_g^m	309.5	0.0	30.48	0.00	-	8.5	3.4
	E_{gv}	309.5	0.0	30.44	0.00	-	8.5	3.5	E_{gv}^m	310.0	0.0	31.55	0.00	-	10.6	-
u2-20-6	E_b	325.5	0.0	29.42	0.00	-	-	12.9	E_b^m	325.5	0.0	30.26	0.00	-	0.0	10.4
	E_g	326.5	0.0	31.32	0.00	-	8.5	7.3	E_g^m	326.0	2.0	32.59	0.80	f	14.9	3.5
	E_{gv}	326.5	1.5	32.42	0.61	f	14.9	4.0	E_{gv}^m	327.5	5.0	33.78	2.45	f	21.3	-

6 Conclusion

This paper investigates the impacts of vehicle dynamics on routing problems with Electric Vehicles (EVs), particularly in the Pickup-and-Delivery Problem (PDP). We developed a model for the PDP based on six different energy calculation methods that considers parameters of vehicle dynamics to varying degrees. The selected parameters are road gradient, extra mass and driving patterns (speed profiles). We also developed a complete PDP system model with charging and energy constraints and evaluated our underlying energy models under a set of adapted realistic instances. The results indicate that the energy requirement of routes planned for EVs can be underestimated by up to 16.3% in our instances if the fundamental parameters of gradient, speed and mass are ignored in the energy calculations. Although adding the mass metric into the energy calculation increases the model complexity, the results of this study show that ignoring mass from the energy calculation of EVs can potentially lead to infeasible trips with insufficient energy to complete the planned trip. Comparing the impacts of parameters on routes' energy attributes, we can see that gradient and speed make a greater contribution to the actual energy requirement of routes than mass in our experiment. We also investigate an alternative scenario for point-to-point paths in the PDP by optimising underlying paths for their energy consumption. The results of experiments on our instances show that choosing energy-optimum paths instead of traditional shortest paths can make the planned trips more efficient in terms of energy, but slightly longer in term of time. Nonetheless, trips planned with our alternative scenario showed better energy efficiency and require less/shorter charging visits.

The use of a high-level, constraint-based modelling system like MiniZinc allowed us to experiment with these different energy models without creating new dedicated solving approaches. The results from our experiments show that a significant investment in such new algorithms may not only be useful, but crucial in order to ensure valid and efficient vehicle routing with EVs.

Future work could look at integrating the energy models of this study with other real-world routing problems for EVs, for example trip planning period. For PDP, an interesting direction is developing/adapting appropriate heuristics for our complete energy-based model, as adding mass to the calculation increases the complexity and runtime.

References

- 1 Saman Ahmadi, SMT Bathaee, and Amir H Hosseinpour. Improving fuel economy and performance of a fuel-cell hybrid electric vehicle (fuel-cell, battery, and ultra-capacitor) using optimized energy management strategy. *Energy Conversion and Management*, 160:74–84, 2018.
- 2 Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.
- 3 Geoff Boeing. Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Comput. Environ. Urban Syst.*, 65:126–139, 2017. doi:10.1016/j.compenvurbsys.2017.05.004.
- 4 Claudia Bongiovanni, Mor Kaspi, and Nikolas Geroliminis. The electric autonomous dial-a-ride problem. *Transportation Research Part B: Methodological*, 122:436–456, 2019.
- 5 Geoffrey Chu. *Improving Combinatorial Optimization*. PhD thesis, Department of Computing and Information Systems, University of Melbourne, 2011.
- 6 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. doi:10.1007/BF01386390.
- 7 Lester R Ford Jr. Network flow theory. Technical report, Rand Corp Santa Monica Ca, 1956.

- 8 Dominik Goeke. *Emerging Trends in Logistics: New Models and Algorithms for Vehicle Routing*. PhD thesis, Department of Business and Economics at the University of Kaiserslautern, 2018.
- 9 Dominik Goeke and Michael Schneider. Routing a mixed fleet of electric and conventional vehicles. *Eur. J. Oper. Res.*, 245(1):81–99, 2015. doi:10.1016/j.ejor.2015.01.049.
- 10 Merve Keskin and Bülent Çatay. A matheuristic method for the electric vehicle routing problem with time windows and fast chargers. *Comput. Oper. Res.*, 100:172–188, 2018. doi:10.1016/j.cor.2018.06.019.
- 11 Çagri Koç, Ola Jabali, Jorge E. Mendoza, and Gilbert Laporte. The electric vehicle routing problem with shared charging stations. *Int. Trans. Oper. Res.*, 26(4):1211–1243, 2019. doi:10.1111/itor.12620.
- 12 Tony Markel, Aaron Brooker, T Hendricks, V Johnson, Kenneth Kelly, Bill Kramer, Michael O’Keefe, Sam Sprik, and Keith Wipke. Advisor: a systems analysis tool for advanced vehicle modeling. *Journal of power sources*, 110(2):255–266, 2002.
- 13 Alejandro Montoya, Christelle Guéret, Jorge E. Mendoza, and Juan G. Villegas. The electric vehicle routing problem with nonlinear charging function. *Transportation Research Part B: Methodological*, 103:87–110, 2017. Green Urban Transportation. doi:10.1016/j.trb.2017.02.004.
- 14 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.
- 15 Samuel Pelletier, Ola Jabali, and Gilbert Laporte. Charge scheduling for electric freight vehicles. *Transportation Research Part B: Methodological*, 115:246–269, 2018. doi:10.1016/j.trb.2018.07.010.
- 16 Michael Schneider, Andreas Stenger, and Dominik Goeke. The electric vehicle-routing problem with time windows and recharging stations. *Transp. Sci.*, 48(4):500–520, 2014. doi:10.1287/trsc.2013.0490.
- 17 Marius M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Oper. Res.*, 35(2):254–265, 1987. doi:10.1287/opre.35.2.254.
- 18 Paolo Toth and Daniele Vigo. *Vehicle Routing*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2014. doi:10.1137/1.9781611973594.
- 19 Hao Wang and Ruey Long Cheu. Operations of a taxi fleet for advance reservations using electric vehicles and charging stations. *Transportation Research Record*, 2352(1):1–10, 2013. doi:10.3141/2352-01.

Building High Strength Mixed Covering Arrays with Constraints

Carlos Ansótegui ✉ 🏠 

Logic & Optimization Group (LOG), University of Lleida, Spain

Jesús Ojeda ✉ 🏠 

Logic & Optimization Group (LOG), University of Lleida, Spain

Eduard Torres ✉ 🏠 

Logic & Optimization Group (LOG), University of Lleida, Spain

Abstract

Covering arrays have become a key piece in Combinatorial Testing. In particular, we focus on the efficient construction of Covering Arrays with Constraints of high strength. SAT solving technology has been proven to be well suited when solving Covering Arrays with Constraints. However, the size of the SAT reformulations rapidly grows up with higher strengths. To this end, we present a new incomplete algorithm that mitigates substantially memory blow-ups. The experimental results confirm the goodness of the approach, opening avenues for new practical applications.

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

Keywords and phrases Combinatorial Testing, Covering Arrays, Maximum Satisfiability

Digital Object Identifier 10.4230/LIPIcs.CP.2021.12

Supplementary Material The tools we implemented are available in <http://hardlog.udl.cat/static/doc/prbot-its/html/index.html> as well as detailed installation and execution instructions.

Funding Supported by MICINNs PROOFS (PID2019-109137GB-C21) and FPU fellowship (FPU18/02929).

1 Introduction

Imagine that we want to test a system (a circuit, a program, a cloud application, an industrial engine, a GUI, etc.) to detect errors, bugs, or faults. The System Under Test (SUT) is in essence a black box with a set of input parameters P which take values into a finite domain. These input parameters are assigned to a particular value and then the SUT is run or executed. We assume the only observable output is whether the system crashed or not.

To validate the SUT is working properly, we can simply iteratively conduct a set of tests (assignments of values to the input parameters) and check whether the SUT is working as expected or not. In practice, when the SUT is run, even if we do not explicitly assign a value to a given input parameter it will take its value by default or it will be automatically assigned following some criterion.

Notice that the number of settings (possible tests) to the input parameters (the parameter space) is $\prod_{p \in P} g_p \in \mathcal{O}(g^{|P|})$ (where g_p is the cardinality of the domain of parameter p and g is the cardinality of the greatest domain) what yields a *combinatorial* explosion and makes unrealistic to run the SUT under all the possible tests.

Combinatorial Testing (CT) [26] techniques aim to build test suites of a reasonable size but yet powerful enough to *cover* most of the errors, bugs, or faults reported to frequently arise. The point is that, in general, the errors are caused by the interaction of a *relatively small* set of the parameters [22]. Notice that a single test covers $\binom{|P|}{t}$ interactions, where t



© Carlos Ansótegui, Jesús Ojeda, and Eduard Torres;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 12; pp. 12:1–12:17



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

(referred to as the *strength*) is the number of parameters involved in the interaction. Therefore, every time we evaluate the SUT under a given test we implicitly check or validate $\binom{|P|}{t}$ interactions of t parameters (referred to as t -tuples).

A test suite of size N for a SUT of P parameters that covers *all* the t -tuples is also known as a Covering Array $CA(N; t, P)$ of strength t . The minimum N for which there exists a $CA(N; t, P)$ is referred to as the Covering Array Number $CAN(t, P)$. Additionally, notice that any test suite of size $< CAN(t, P)$ will not cover all t -tuples, but we may be still interested in covering the maximum number of t -tuples with the number of tests our budget can afford.

In this paper, we show how to build *Mixed* Covering Arrays with *Constraints* (MCACs) of high strength. The term *Mixed* refers to the possibility of having parameter domains of different sizes. The term *Constraints* refers to the existence of some parameter interactions that are not allowed in the system. These forbidden interactions are usually implicitly described by a set of SUT constraints. Therefore, the tests in our test suite must *satisfy* the SUT constraints. In particular, the problem of computing an MCAC of minimum length is NP-hard [24].

There exist several greedy approaches for building MCACs, such as PICT [13] (from Microsoft), based on the OTAT framework [11], and ACTS [10] (used by more than 4000 corporate users and universities), based on the IPOG algorithm [14]. However, they are not well suited in terms of handling SUT constraints and will scale poorly as the complexity or hardness of the SUT constraints grows. This is why, here, we focus on constraint programming based approaches; particularly, we work with Satisfiability (SAT) based approaches [9].

SAT technology provides a highly competitive generic problem approach for solving decision and optimization problems. In particular, the decision problem to be solved is translated to the SAT problem which determines whether there is an assignment to the Boolean variables in a propositional formula in Conjunctive Normal Form (CNF) (set of clauses) that *satisfies* the formula. Additionally, optimization problems can be translated into the Maximum Satisfiability (MaxSAT) problem which is the optimization version of the SAT problem.

The CALOT [30] tool for building MCACs is based on an incremental SAT solving approach which iteratively decreases the upper bound on the size of the test suite, formulating at every iteration as a SAT problem whether there exists an MCAC of size N , till $CAN(t, P)$ is reached. The CALOT approach is extended by recent work in [2] where a MaxSAT formulation based on [4] is proposed allowing the application of the new generation of complete and incomplete MaxSAT solvers [5]. The initial upper bound for these approaches is computed through the application of the ACTS tool.

While these approaches may be efficient enough for testing some SUTs, the size of the SAT or MaxSAT formulas required for building MCACs rapidly grows with the number of tests and size of SUT constraints but mostly with the strength t taken into consideration.

Regarding the number of tests and size of the SUT constraints, the SAT and MaxSAT formulations of the mentioned approaches need to incorporate at least N copies of the SUT constraints where N is the size of the test suite we try to build. In this sense, if the ACTS tool is not able to provide a good enough upper bound then other strategies need to be taken into account since the trivial upper bound, as discussed, can be unaffordable in terms of size. There are approaches like [29] (based on SAT and the domain-dependent PICT heuristic) and [2] (based on MaxSAT) that mitigate this problem by iteratively constructing the test suite, i.e. adding just one single test at a time that aims to maximize the number of interactions covered so far ¹. The addition of one single test guarantees we only deal with one copy of the SUT constraints.

¹ [2] can add more than one test at each iteration.

Regarding the strength t , the size of the SAT/MaxSAT formulas into existing approaches is proportional to the potential number of allowed interactions, i.e. $\mathcal{O}\left(\binom{|P|}{t} \cdot g^t\right)$ where g is the cardinality of the greatest domain. Typical applications use values of $t = 2$ and barely $t = 3$. However, the more complex the SUT is, the higher the probability that faulty or buggy interactions be caused by a larger number of parameters. Therefore, we need to consider higher values like $t = 4$ and $t = 5$, what clearly is a bottleneck for the mentioned SAT or MaxSAT approaches.

Finally, there are other recent Constraint Programming approaches but they focus on $t = 2$ ([17, 18]) or they do not allow SUT constraints ([21]).

In this paper, we show how we can build practical higher strength MCACs through SAT technology without incurring in memory blow-ups. In particular, we first present a new incomplete algorithm named (Refined Build One Test – Incremental Test Suite) RBOT-its, inspired on Algorithm 5 in [29]. RBOT-its builds the MCAC test by test and optimizes (refines) subsets of the incremental test suite built so far by applying a MaxSAT based approach. Then, we present another incomplete algorithm named PRBOT-its (Pool-based Refined Build One Test – Incremental Test Suite) that iteratively builds the MCAC while simultaneously keeping in a memory pool just a fraction of all the possible t -tuples of the SUT fulfilling the memory size requirements.

The paper is structured as follows. In Section 2 we introduce some definitions on Covering Arrays, SAT and MaxSAT. Section 3 shows how the $CAN(t, P)$ problem can be encoded to MaxSAT. Section 4 presents the BOT-its algorithm (Build One Test – Iterative Test Suite), an algorithm that incrementally builds MCACs test by test. Section 5 presents the RBOT-its algorithm that uses a MaxSAT approach to improve the BOT-its algorithm. Section 6 describes the PRBOT-its algorithm that shows how to adapt RBOT-its to operate on low memory requirements. In Section 7 we study how these approaches compare to the ACTS tool. Finally, in Section 8 we conclude and mention some future work.

2 Preliminaries

We introduce some definitions related to Covering Arrays and SAT technology.

► **Definition 1.** A *System Under Test (SUT) model* is a tuple $\langle P, \varphi \rangle$, where P is a finite set of variables p of finite domain, called *SUT parameters*, and φ is a set of constraints on P , called *SUT constraints*, that implicitly represents the parameterizations that the system accepts. We denote by $d(p)$ and g_p , respectively, the domain and the cardinality domain of p . For the sake of clarity, we will assume that the system accepts at least one parameterization.

In the following, we assume $S = \langle P, \varphi \rangle$ to be a SUT model. We will refer to P as S_P , and to φ as S_φ .

► **Definition 2.** An *assignment* is a set of pairs (p, v) where p is a variable and v is a value of the domain of p . A *test case* for S is a full assignment A to the variables in S_P such that A entails S_φ (i.e. $A \models S_\varphi$). A *parameter tuple* of S is a subset $\pi \subseteq S_P$. A *value tuple* of S is a partial assignment to S_P ; in particular, we refer to a value tuple of length t as a t -tuple.

► **Definition 3.** A t -tuple τ is *forbidden* if τ does not entail S_φ (i.e. $\tau \not\models S_\varphi$). Otherwise, it is *allowed*. We refer to the set of allowed t -tuples as $\mathcal{T}_a = \{\tau \mid \tau \models S_\varphi\}$.

► **Definition 4.** A test case v *covers* a value tuple τ if both assign the same domain value to the variables in the value tuple, i.e., $v \models \tau$. A test suite Υ *covers* a value tuple τ (i.e., $\tau \subseteq \Upsilon$) if there exist a test case $v \in \Upsilon$ s.t. $v \models \tau$. We refer to $v \not\models \tau$ ($\tau \not\subseteq \Upsilon$) when a test case (test suite) does not cover τ .

► **Definition 5.** A Mixed Covering Array with Constraints (MCAC), denoted by $CA(N; t, S)$, is a set of N test cases for a SUT model S such that all t -tuples are at least covered by one test case. The term Mixed reflects that the domains of the parameters in S_P are allowed to have different cardinalities. The term Constraints reflects that S_φ is not empty ².

► **Definition 6.** The MCAC problem is to find an MCAC of size N .

► **Definition 7.** The Covering Array Number, $CAN(t, S)$, is the minimum N for which there exists an MCAC $CA(N; t, S)$. The Covering Array Number problem is to find an MCAC of size $CAN(t, S)$.

► **Definition 8.** A literal is a propositional variable x or a negated propositional variable $\neg x$. A clause is a disjunction of literals. A Conjunctive Normal Form (CNF) is a conjunction of clauses.

► **Definition 9.** A weighted clause is a pair (c, w) , where c is a clause and w , its weight, is a natural number or infinity. A clause is hard if its weight is infinity (or no weight is given); otherwise, it is soft. A Weighted Partial MaxSAT instance is a multiset of weighted clauses.

► **Definition 10.** A truth assignment for an instance ϕ is a mapping that assigns to each propositional variable in ϕ either 0 (False) or 1 (True). A truth assignment is partial if the mapping is not defined for all the propositional variables in ϕ .

► **Definition 11.** A truth assignment I satisfies a literal x ($\neg x$) if I maps x to 1 (0); otherwise, it is falsified. A truth assignment I satisfies a clause if I satisfies at least one of its literals; otherwise, it is violated or falsified. The cost of a clause (c, w) under I is 0 if I satisfies the clause; otherwise, it is w . Given a partial truth assignment I , a literal or a clause is undefined if it is neither satisfied nor falsified. A clause c is a unit clause under I if c is not satisfied by I and contains exactly one undefined literal.

► **Definition 12.** The cost of a formula ϕ under a truth assignment I , denoted by $\text{cost}(I, \phi)$, is the aggregated cost of all its clauses under I .

► **Definition 13.** The Weighted Partial MaxSAT problem for an instance ϕ is to find an assignment in which the sum of weights of the falsified soft clauses is minimal, denoted by $\text{cost}(\phi)$, and all the hard clauses are satisfied. The Partial MaxSAT problem is the Weighted Partial MaxSAT problem where all weights of soft clauses are equal. The SAT problem is the Partial MaxSAT problem when there are no soft clauses. An instance of Weighted Partial MaxSAT, or any of its variants, is unsatisfiable if its optimal cost is ∞ . A SAT instance ϕ is satisfiable if there is a truth assignment I , called model, such that $\text{cost}(I, \phi) = 0$.

► **Definition 14.** An Exactly-One (EO) constraint is a cardinality constraint of the form $\sum_{i=1}^n l_i = 1$ where l_i are propositional literals.

3 The $CAN(t, S)$ problem as MaxSAT

In this section, we first show a SAT encoding for the MCAC problem inspired on previous approaches [19, 20, 6, 25, 4, 30, 2]. Then, we present the MaxSAT encoding for the $CAN(t, S)$ problem presented in [4, 2]. Exactly One cardinality constraints are translated into CNF through the regular encoding [1, 16].

² Notice that the CSPLib 045 problem definition of Covering Arrays [28] does not consider SUT Constraints.

First, we encode through variables $x_{i,p,v}$ that a test case i assigns value v to parameter p . We restrict each parameter to take one value per test case as follows (where $[N] = \{1, \dots, N\}$):

$$\bigwedge_{i \in [N]} \bigwedge_{p \in S_P} \sum_{v \in d(p)} x_{i,p,v} = 1 \quad (X)$$

In order to enforce the SUT constraints, we convert φ to SAT³ by substituting each (p, v) in φ by the corresponding literal on the propositional variable $x_{i,p,v}$ for each test case i .

$$\bigwedge_{i \in [N]} CNF \left(S_\varphi \left\{ \frac{\neg x_{i,p,v}}{p \neq v}, \frac{x_{i,p,v}}{p = v} \right\} \right) \quad (SUTX)$$

Variables c_τ^i represent that t -tuple τ is covered by test case i or by any lower test case j , where $1 \leq j \leq i$ (equation $CCX(a)$). To ensure that τ will be covered by some test, we set c_τ^N to be True and c_τ^0 to be False (equations $CCX(b)$ and $CCX(c)$). Notice that only t -tuples that can be covered by a test case are encoded, i.e., $\tau \in \mathcal{T}_a$.

$$\bigwedge_{i \in [N]} \bigwedge_{\tau \in \mathcal{T}_a} \bigwedge_{(p,v) \in \tau} (c_\tau^i \rightarrow c_\tau^{i-1} \vee x_{i,p,v}) \quad (a) \ (CCX)$$

$$\bigwedge_{\tau \in \mathcal{T}_a} c_\tau^N \quad (b)$$

$$\bigwedge_{\tau \in \mathcal{T}_a} (c_\tau^N \rightarrow \neg c_\tau^0) \quad (c)$$

► **Proposition 15.** *Let $Sat_{CCX}^{N,t,S}$ be $X \wedge SUTX \wedge SCCX$. $Sat_{CCX}^{N,t,S}$ is satisfiable iff a $CA(N; t, S)$ exists.*

As we can see, sets $SUTX$ and CCX will be responsible for memory blow-ups when dealing with a large number of tests or allowed t -tuples.

The presented $Sat_{CCX}^{N,t,S}$ encoding requires an upper bound on N and a way to avoid encoding the forbidden t -tuples. These can be extracted from any suboptimal MCAC solution. We can take as upper bound N the number of tests of the solution and discard all the missing t -tuples (as these will be forbidden). After that, row symmetry breaking techniques can be applied. We can compute which is the parameter tuple of length t with the maximum number r of t -tuples, and then fix these r t -tuples in the first r test cases. Notice that these t -tuples are mutually exclusive and must be covered into different test cases. We will refer to the lower bound as $lb = r - 1$ (i.e. it is not possible to find an MCAC with $r - 1$ tests).

This $Sat_{CCX}^{N,t,S}$ encoding can be extended to a MaxSAT encoding for the $CAN(t, S)$ problem, as described in [2, 4]. We will use an indicator variable u_i that is True iff test case i is part of the MCAC. The objective function of the optimization problem, which aims to minimize the number of variables u_i set to True, is encoded into Partial MaxSAT by adding the following set of soft clauses:

$$\bigwedge_{i \in [lb+2 \dots N]} (\neg u_i, 1) \quad (SoftU)$$

³ We consider that φ is already in CNF format.

12:6 Building High Strength Mixed Covering Arrays with Constraints

Notice that we only need to use $N - (lb + 1)$ indicator variables since we know that the covering array will have at least $lb + 1$ tests. To avoid symmetries, it is also enforced that if test case $i + 1$ belongs to the MCAC, so does the previous test case i :

$$\bigwedge_{i \in [lb+2 \dots N-1]} (u_{i+1} \rightarrow u_i) \quad (BSU)$$

Finally, we just need to state how variables u_i are related to variables c_τ^i . This constraint reflects that if u_i is False (i.e., tests $\geq i$ are not in the solution), then the tuple τ has to be covered at some test below i :

$$\bigwedge_{i \in [lb+2 \dots N]} \bigwedge_{\tau \in \mathcal{T}_a} (\neg u_i \rightarrow c_\tau^{i-1}) \quad (CCU)$$

► **Proposition 16.** *Let $PMSat_{CCX}^{N,t,S,lb}$ be $SoftU \wedge BSU \wedge CCU \wedge Sat_{CCX}^{N,t,S}$. If $N \geq CAN(t, S)$, the optimal cost of the Partial MaxSAT instance $PMSat_{CCX}^{N,t,S,lb}$ is $CAN(t, S) - (lb + 1)$, otherwise it is ∞ .*

The main problem with these SAT and MaxSAT encodings is that their size dramatically grows with the number of tests and t -tuples to cover. This makes the SAT-based solving approach unpractical in real scenarios. In the next sections, we show how to avoid memory blow-ups by describing new incomplete approaches.

4 Incremental Test Construction

To reduce the number of tests that we need to encode, the idea is to incrementally build the test suite, test by test. Therefore, at any iteration, we just encode the SUT constraints once.

Algorithm BOT-its (Build One Test - Iterative Test Suite), which is inspired on Algorithm 5 in [29], builds an MCAC by iteratively calling the BuildOneTest (BOT) algorithm (an algorithm that greedily builds a new test, see details below). BOT-its keeps a pool p of the t -tuples yet to cover. Then, it incrementally extends the working test suite Υ by appending the new test v computed by the BOT algorithm. The pool p is simplified by erasing those t -tuples covered by v . Finally, the algorithm returns when the pool becomes empty.

■ **Algorithm BOT-its** Build One Test – Incremental Test Suite algorithm.

Input : SUT model S , strength t , consistency check conflict budget cb
Output : Test suite Υ

```

1  $\Upsilon \leftarrow \emptyset$  # Working test suite
2  $p \leftarrow$  pool with all  $t$ -tuples of  $S$ 
3  $sat \leftarrow$  incremental SAT solver initialized with  $X$  and  $SUTX$  constraints
4 while  $p \neq \emptyset$  do
5    $v, p \leftarrow BOT(S, p, sat, cb)$ 
6    $\Upsilon \leftarrow \Upsilon \cup \{v\}$ 
7    $p_v \leftarrow \{\tau \mid \tau \in p \wedge v \models \tau\}$  # Tuples in  $p$  covered by  $v$ 
8    $p \leftarrow p \setminus p_v$ 
9 return  $\Upsilon$ 

```

Next, we show the pseudocode for the BuildOneTest (BOT) algorithm, also inspired on Algorithm 5 in [29]. The BOT algorithm receives the pool p with the t -tuples yet to cover. In order to build the current test, BOT uses the PICT heuristic [13] to identify the parameter tuple (to which we refer as the PICT t -tuple) with most t -tuples in the pool. Then, it selects one to initialize the test under construction (line 1).

■ **Algorithm BuildOneTest (BOT)** Inspired on Algorithm 5 in [29].

Input : SUT model S , Tuples pool p , SAT solver sat , consistency check conflict budget cb

Output : A new test case v

All functions can access S , p and sat

```

1  $v \leftarrow$  choose  $\tau \in p$  as in PICT s.t.  $consistent(\tau, \infty)$  #  $v$  covers at least  $\tau$ 
2 while there exist  $(p, v)$  s.t.  $v \cup \{(p, v)\}$  covers a tuple in  $p$  and
    $consistent(v \cup \{(p, v)\}, cb)$  do
3   | Choose such best  $(p, v)$  #  $v \cup \{(p, v)\}$  covers more tuples in  $p$ 
4   |  $v \leftarrow v \cup \{(p, v)\}$ 
5 if exists  $\tau \in p$  s.t.  $\tau$  can be covered in  $v$  and  $consistent(\tau, cb)$  then
6   | choose  $\tau \in p$  as in PICT
7   |  $v \leftarrow v \cup \tau$ 
8   | go to line 2
9  $v \leftarrow amend(v)$ 
10 return  $v, p$ 

```

To make sure the PICT selection is consistent with the SUT constraints, BOT runs a consistency check (of unlimited cb conflicts). In particular, in function *consistent* in BOT auxiliary functions, a SAT solver is used to check the validity of the parameters assigned so far with respect to the SUT constraints. The SAT instance represents the SUT constraints and the SAT solver is executed using as assumptions the partial assignment of all the fixed parameters in the current test. If the check fails, an unsatisfiable core is retrieved⁴, i.e., a subset of the formula that is already unsatisfiable. In particular, the core contains the set of assumptions responsible for the unsat answer. Moreover, the t -tuples in the pool subsumed by the core are removed since these are forbidden tuples (line 4 in function *consistent*). Notice that this way a lazy removal of forbidden tuples is implemented.

After the PICT selection, it iteratively selects from the set of unassigned parameters, the pair parameter-value (p, v) that, in combination with the parameters fixed so far, covers at least one t -tuple in the pool, preferring the one that covers most (lines 2 - 4). To preemptively detect if the selected parameter plus the previous partial assignment is inconsistent with the SUT constraints it calls function *consistent* but with a limited number of conflicts cb , since the check can be expensive and we can not afford a full check at this point.

Whenever the above process saturates, i.e. reaches a fixpoint, and there are yet unassigned parameters, a new t -tuple is selected as in PICT and assigned to the test. Then, the process starts again (line 8). In this case, we also guarantee the selected tuple is consistent with the SUT constraints running *consistent* function with limited conflicts budget cb .

At this point, we have heuristically built a partial test that aims to cover most of the t -tuples in the pool, but we may not be able to extend it to a full test consistent with the SUT constraints. Therefore, the partial test may have to be amended (line 9).

⁴ When $cb \neq \infty$ the result of the check might be unknown.

■ **Algorithm BOT auxiliary functions** Auxiliary functions for algorithm BOT.

```

# All functions can access  $S$ ,  $p$  and  $sat$ 
1 function consistent( $\tau, cb$ )
2   if  $sat.solve(\tau, cb) = True$  then return True
3   else
4      $p \leftarrow p \setminus \{\tau \mid sat.core() \subseteq \tau \wedge \tau \in p\}$            #  $p$  updated in place
5     return False
6 function amend( $v$ )
7   while not consistent( $v, \infty$ ) do
8      $(p, v) \leftarrow$  most recently fixed  $(p, v)$  in  $v$  s.t.  $(p, v) \in sat.core()$ 
9      $v \leftarrow v \setminus \{(p, v)\}$ 
10  Fix unfixed parameters in  $v$  according to  $sat.model()$ 
11  return  $v$ 

```

This *amend* process (see BOT auxiliary functions) tries to preserve the greatest slice of the partial test that can be extended to a full test consistent with the SUT constraints through the call to function *consistent* with an unlimited budget. In case the partial test is inconsistent, to amend it, the assumptions in the core are removed in reverse chronological order (lines 7 - 9 in function *amend*) till the SAT solver is able to complete the test satisfying the SUT constraints (line 10).

When the BOT algorithm ends, it returns the new test just built v and the input pool p without those forbidden t -tuples that were detected (line 4 in function *consistent*).

The implementation of Algorithm 5 in [29], on which BOT-its and BOT algorithms are inspired, is not available after request to the authors for reproducibility purposes. Our BOT algorithm, apart from implementation details, differs fundamentally on function *consistent*. In particular, on how we specifically conduct a consistency check with a limited number of conflicts.

5 Refining Test Suites

In Section 4 we showed how algorithm BOT-its builds incrementally an MCAC. Notice that the MCAC might not be optimal (i.e. it may exist a smaller MCAC) since BOT-its is a *greedy* algorithm.

Taking as upper bound the size of the suboptimal MCAC provided by the BOT-its algorithm (see Section 4) we can always try to find an smaller MCAC as described in Section 3. Notice that depending on the number of parameters, the strength t and the number of tests, the Partial MaxSAT encoding might be unreasonably large.

To circumvent this issue we essentially compute whether a portion of the MCAC under construction can be *refined* to use fewer tests but cover the same t -tuples in the pool p . We refer to this portion (test suite) as the *window* to be *refined*.

In this section we present algorithm RBOT-its, which is an improvement over BOT-its. Red lines show the extensions.

In particular, we keep an *sliding window* of tests that starts at $w.i$ and ends in the last test of Υ . This window also keeps track of the t -tuples $(w.p)$ of the pool p covered by the window (line 11).

■ **Algorithm RBOT-its** Refined BOT-its algorithm. Differences with BOT-its in red.

```

Input  : SUT model  $S$ , strength  $t$ , consistency check conflict budget  $cb$ 
Output: Test suite  $\Upsilon$ 
1  $\Upsilon \leftarrow \emptyset$                                 # Working test suite
2  $p \leftarrow$  pool with all  $t$ -tuples of  $S$ 
3  $sat \leftarrow$  incremental SAT solver initialized with  $X$  and  $SUTX$  constraints
4  $w.p \leftarrow \emptyset$                           # Window of covered tuples
5  $w.i \leftarrow 0$                                 # Window starting test index
6 while  $p \neq \emptyset$  do
7    $v, p \leftarrow BOT(S, p, sat, cb)$ 
8    $\Upsilon \leftarrow \Upsilon \cup \{v\}$ 
9    $p_v \leftarrow \{\tau \mid \tau \in p \wedge v \models \tau\}$       # Tuples in  $p$  covered by  $v$ 
10   $p \leftarrow p \setminus p_v$ 
11   $w.p \leftarrow w.p \cup p_v$ 
12  while  $window\_is\_full(\Upsilon, w)$  do
13     $\Upsilon, p, w \leftarrow refine(\Upsilon, p, w)$ 
14   $\Upsilon, p, w \leftarrow refine(\Upsilon, p, w)$ 
15 return  $\Upsilon$ 

```

We keep track of the potential memory size of the Partial MaxSAT required to refine the window. While we hit the maximum allowed size by our system (i.e. function *window_is_full* in line 12 returns true) we execute the refining process (line 13). As we will see below, the refine process, even reducing the number of tests, it may cause to cover additional t -tuples that were not previously in the window. The side effect is that the window may remain full in terms of memory requirements.

Once the algorithm has covered all t -tuples in p , we apply a last refinement to the last window to ensure that it is refined even if the window is not full (line 14).

Function *refine* in Refine tries to cover the same tuples covered in the window $w.p$ but using less tests. First, it encodes as Partial MaxSAT the problem of building a test suite with the minimum number of tests that covers the t -tuples in the window. This can be achieved by making use of the Partial MaxSAT encoding for the $CAN(t, S)$ problem described in Section 3, but taking as \mathcal{T}_a the set of t -tuples into the window and as upper bound ub the window size.

Then, we run a MaxSAT solver and extract the test suite induced by the solution it reports. If the size of this test suite is smaller than the window size, we use it to replace the window in Υ (line 5). We also update the t -tuples covered by the window, since we may cover extra tuples p_x with the new tests (lines 6 - 8). Otherwise, we reduce the size of the window by excluding the test $w.i$ and update properly the window (lines 10 - 13).

6 Incremental Pool of t -tuples

There is yet a main practical problem with the BOT-its algorithm which is the high memory consumption by the pool of t -tuples to be covered. In particular, when t or the number of parameters is high enough.

In this section we present algorithm PRBOT-its, an extension of RBOT-its (see Section 5) to avoid memory blow-ups by limiting the number of t -tuples to be considered when building a test. Red lines show the differences respect to algorithm RBOT-its.

■ **Algorithm Refine** Test suites refinement function.

```

# refine function can access  $S$ ,  $t$  and  $b$ 
1 function refine( $\Upsilon$ ,  $p$ ,  $w$ )
2    $\varphi \leftarrow \text{encode}(S, \Upsilon_{\geq w.i}, w.p)$ 
3    $\Upsilon_r \leftarrow \text{solve}(\varphi)$ 
4   if  $\Upsilon_r \neq \emptyset$  and  $|\Upsilon_r| < |\Upsilon_{\geq w.i}|$  then
5     Replace  $\Upsilon_{\geq w.i}$  by  $\Upsilon_r$  in  $\Upsilon$ 
6      $p_x \leftarrow \{\tau \mid \tau \in p \wedge \Upsilon_r \models \tau\}$ 
7      $p \leftarrow p \setminus p_x$ 
8      $w.p \leftarrow w.p \cup p_x$ 
9   else
10     $v_r \leftarrow \text{test case with index } w.i \text{ in } \Upsilon$ 
11     $\Upsilon \leftarrow \Upsilon \setminus \{v_r\}$ 
12     $w.p \leftarrow w.p \setminus \{\tau \mid \tau \in w.p \wedge v_r \models \tau\}$ 
13     $w.i \leftarrow w.i + 1$ 
14  return  $\Upsilon$ ,  $p$ ,  $w$ 

```

This algorithm works on a partial pool p of size at most b . The pool is incrementally filled with new pending t -tuples, to finally traverse all the t -tuples (line 8). Once the pool p is full, the BOT algorithm is called to build a test that tries to cover as much t -tuples as possible in p (line 9, see Section 4). Then, the algorithm proceeds as algorithm RBOT-its (lines 10 – 16). The main loop ends when the pool is empty and there are not pending tuples (unseen tuples) to add to the pool (function *unseen_tuples?*). Finally, as in algorithm RBOT-its we perform a last refinement.

BOT algorithm has been also modified in the following way. In particular, within function *consistent* (called by BOT algorithm) whenever we discard forbidden tuples, we additionally call function *fill_pool* after line 4 in BOT auxiliary functions, as follows:

$$\Upsilon, p, w, \tau \leftarrow \text{fill_pool}(\Upsilon, p, w, \tau)$$

The goal is to take advantage of the available extra space in the pool thanks to the lazy detection and removal of forbidden tuples. Consequently, the call to function BOT in algorithm PRBOT-its (line 9) is extended with the additional entry parameters Υ, w and output parameters w, τ .

To fill the pool of t -tuples we call function *fill_pool* in Fill pool. This function iteratively adds new t -tuples to the pool that are neither in Υ nor in the pool, till p is full or all t -tuples have been processed (seen) (lines 2 – 4).

New t -tuples are selected taking into account the latest tuple seen τ by calling function *next_tuple* (a total order is implicitly assumed, line 3). Notice that whether τ is a forbidden tuple (not consistent with the SUT constraints) it is handled by the BOT algorithm into the *consistent* function as previously described.

If τ was not already covered in Υ it is added to the pool p . Otherwise, if the new tuple is in particular covered by the current window it is consequently added to the window pool (line 6). Since the window may get full, as in previous algorithms we refine the window pool till it is not full anymore (lines 7 – 8).

■ **Algorithm PRBOT-its** Pool-based RBOT-its algorithm. Differences with RBOT-its in red.

```

Input  : SUT model  $S$ , strength  $t$ , consistency check conflict budget  $cb$ , pool
          budget  $b$ 
Output: Test suite  $\Upsilon$ 
# All functions can access  $S$ ,  $t$  and  $b$ 
1  $\Upsilon \leftarrow \emptyset$                                      # Working test suite
2  $sat \leftarrow$  incremental SAT solver initialized with  $X$  and  $SUTX$  constraints
3  $w.p \leftarrow \emptyset$                                # Window of covered tuples
4  $w.i \leftarrow 0$                                      # Window starting test index
5  $p \leftarrow \emptyset$                                # Working pool of tuples to cover
6  $\tau \leftarrow \emptyset$ 
7 while  $p \neq \emptyset$  or unseen_tuples?( $S, t, \tau$ ) do
8    $\Upsilon, p, w, \tau \leftarrow$  fill_pool( $\Upsilon, p, w, \tau$ )
9    $v, p, w, \tau \leftarrow$  BOT( $S, p, sat, cb, \Upsilon, w$ )
10   $\Upsilon \leftarrow \Upsilon \cup \{v\}$ 
11   $p_v \leftarrow \{\tau \mid \tau \in p \wedge v \models \tau\}$       # Tuples in  $p$  covered by  $v$ 
12   $p \leftarrow p \setminus p_v$ 
13   $w.p \leftarrow w.p \cup p_v$ 
14  while window_is_full( $\Upsilon, w$ ) do
15     $\Upsilon, p, w \leftarrow$  refine( $\Upsilon, p, w$ )
16  $\Upsilon, p, w \leftarrow$  refine( $\Upsilon, p, w$ )
17 return  $\Upsilon$ 

```

7 Experimental Results

In this section, we report the experimental investigation we conducted to assess the performance of the approaches proposed in the preceding sections. We use a total of 58 SUT instances, which are extracted from [12], with 5 real-world and 30 artificially generated covering array problems, [27] with 20 real-world instances, [31] with two industrial instances and, [29] with another industrial instance.

In Table 1 we show the information about each SUT instance. S_P provides the number of parameters and their domain (e.g. in instance *Banking1*, $3^4 4^1$ means 4 parameters of domain 3 and 1 of domain 4) and, S_φ the number of SUT constraints and their sizes (e.g. instance *Banking1* has 112 constraints that involve 5 parameters, 5^{112} in the table).

The environment of execution consists of a computer cluster with machines equipped with two Intel Xeon Silver 4110 (octa-core processors at 2.1GHz, 11MB cache memory) and 96GB DDR4 main memory. All the experiments were executed with a timeout of 12h and a limit of 12GB of RAM. We executed all the algorithms with 10 different seeds, except for the ACTS tool (as it does not expose the *seed* parameter).

We use Python as a programming language and the Python framework OptiLog [3] that provides bindings to state-of-the-art SAT solvers. For our experimentation, we use Glucose 4.1.

■ **Algorithm Fill pool** Fill pool function.

```

# fill_pool function can access S, t and b
1 function fill_pool( $\Upsilon$ , p, w,  $\tau$ )
2   while  $|p| < b$  and unseen_tuples?( $S, t, \tau$ ) do
3      $\tau \leftarrow \text{next\_tuple}(S, t, \tau)$ 
4     if  $\tau \not\subseteq \Upsilon$  then  $p \leftarrow p \cup \{\tau\}$ 
5     elif  $\tau \subseteq \Upsilon_{\geq w.i}$  then
6        $w.p \leftarrow w.p \cup \{\tau\}$ 
7       while window_is_full( $\Upsilon, w$ ) do
8          $\Upsilon, p, w \leftarrow \text{refine}(\Upsilon, p, w)$ 
9   return  $\Upsilon, p, w, \tau$ 

```

We implemented our own version of BOT-its, as the implementation of Algorithm 5 described in [29] was not available from authors for reproducibility purposes⁵. We also found that our implementation is not able to reproduce exactly the results reported in the original work. In particular, we notice that in our case the sizes of the reported MCACs are just slightly higher. Moreover, our implementation also seems to be significantly slower⁶. Notice the authors used as underlying SAT solver lingeling [7] and we use Glucose 4.1, and this may explain part of the divergence. However, this also means that if the implementation of Algorithm 5 from [29] was available we could probably even get better results with our algorithms RBOT-its and PRBOT-its which extend BOT-its. We set the consistency check conflict budget *cb* parameter for all the BOT-its algorithms to 1 (see Section 4).

For the Refine function in algorithms RBOT-its and PRBOT-its we consider the encoding $PM\text{Sat}_{CCX}^{N,t,S,lb}$ described in Section 3. We use a custom implementation of the *linear* [15, 23] MaxSAT algorithm that is able to report suboptimal solutions⁷, using CaDiCaL as the underlying SAT solver [8]. We set a window size of approximately 500MB, a total time limit for the MaxSAT solver of 180s, and a timeout of 30s between solutions (see Section 5). Notice that this setting could be fine-tuned although we did not carry out this analysis. In previous approaches results are provided up to $t = 3$, here we carry out our experiments for $t = 3$, $t = 4$, and $t = 5$ which, as mentioned previously, are also of interest to many applications.

The first question we address is the impact of RBOT-its, the refined version of BOT-its, in terms of size of the reported test suite and run time for $t = 3$. Moreover, we compare with ACTS. We describe the results in Table 1 under columns *tests* and *time*, respectively. Since all approaches are incremental construction methods, we report (under columns “%”) a lower bound on the percentage of allowed t -tuples covered by the retrieved test suite. When the percentage is 100 it means it was possible to build an MCAC. On the other hand, instances that have a “-” in all columns were not able to report any test suite. As we can see, RBOT-its is able to report better MCAC sizes than ACTS and BOT-its on 42 of the 58 instances. This confirms the goodness of the refined approach.

The second question we address is about how much memory is consumed by the BOT-its algorithm. In particular, we estimate the required memory to keep all the t -tuples in memory at the same time. We consider integers of 32 bits and we exclude the memory resources

⁵ The tools we implemented are available in <http://hardlog.udl.cat/static/doc/prbot-its/html/index.html> as well as detailed installation and execution instructions.

⁶ In [29] their algorithms are implemented in C programming language

⁷ Since RBOT-its is incomplete by nature, there is actually no need to use a complete MaxSAT solver.

■ **Table 1** SUT parameters domains and constraints for each instance (columns S_P and S_φ) and memory consumption for $t = 3$ (*mem*). Test suite size, percentage of tuple coverage and time for $t = 3$. In bold the method with better results with the lexicographic criteria (coverage percentage, number of tests, exhausted time). For the coverage percentage enough precision was taken into account. Resources: 12GB memory and 12h timeout.

inst	S_P	S_φ	mem	$t = 3$								
				ACTS			BOT-its			RBOT-its		
				tests	%	time	tests	%	time	tests	%	time
Cohen et al. [12]												
1	$2^{86}3^34^15^56^2$	$2^{20}3^34^1$	20.1MB	293	100%	4s	294.20	100%	12m	294.20	100%	1.3h
2	$2^{86}3^34^35^16^1$	$2^{19}3^3$	15.6MB	174	100%	3s	176.50	100%	6m	149.10	100%	39m
3	$2^{27}4^2$	$2^{20}3^1$	416.0kB	71	100%	1s	72.90	100%	4s	50.50	100%	5m
4	$2^{51}3^34^25^1$	$2^{15}3^2$	3.7MB	102	100%	2s	108.10	100%	48s	81.10	100%	7m
5	$2^{155}3^74^35^56^4$	$2^{32}3^64^1$	112.7MB	386	100%	14s	384	100%	1.6h	384	100%	3.3h
6	$2^{73}4^36^1$	$2^{26}3^4$	8.1MB	119	100%	2s	133.20	100%	2m	98.60	100%	14m
7	$2^{20}3^1$	$2^{13}3^2$	399.7kB	35	100%	1s	39	100%	3s	28.40	100%	3m
8	$2^{109}3^24^25^36^3$	$2^{32}3^44^1$	34.5MB	326	100%	5s	306.60	100%	23m	306.20	100%	1.1h
9	$2^{57}3^14^15^16^1$	$2^{30}3^7$	4.2MB	84	100%	2s	94.30	100%	44s	60	100%	4m
10	$2^{130}3^64^55^26^4$	$2^{40}3^7$	68.2MB	329	100%	9s	342.60	100%	51m	341.30	100%	2.4h
11	$2^{84}3^44^25^26^4$	$2^{28}3^4$	20.1MB	318	100%	4s	328.70	100%	13m	328.60	100%	1.4h
12	$2^{136}3^44^35^16^3$	$2^{23}3^4$	60.5MB	263	100%	7s	269.80	100%	36m	250	100%	1.6h
13	$2^{124}3^44^15^26^2$	$2^{22}3^4$	43.5MB	200	100%	7s	214.40	100%	19m	183.70	100%	1.0h
14	$2^{81}3^54^36^3$	$2^{13}3^2$	16.3MB	244	100%	3s	244.30	100%	7m	216.30	100%	20m
15	$2^{50}3^44^15^26^1$	$2^{20}3^2$	4.1MB	173	100%	2s	180.10	100%	1m	150.90	100%	5m
16	$2^{81}3^34^26^1$	$2^{30}3^4$	11.6MB	117	100%	3s	138.50	100%	3m	96.40	100%	9m
17	$2^{128}3^34^25^16^3$	$2^{25}3^4$	48.3MB	265	100%	6s	263.50	100%	30m	239.40	100%	1.3h
18	$2^{127}3^24^45^56^2$	$2^{23}3^44^1$	59.9MB	344	100%	8s	327.20	100%	41m	327.20	100%	2.1h
19	$2^{172}3^94^95^36^4$	$2^{38}3^5$	166.3MB	373	100%	21s	385	100%	2.6h	365.50	100%	6.7h
20	$2^{138}3^34^55^26^7$	$2^{42}3^6$	94.5MB	463	100%	12s	465.60	100%	1.5h	465.60	100%	4.3h
21	$2^{76}3^34^25^56^3$	$2^{40}3^6$	13MB	235	100%	3s	235.40	100%	5m	216.50	100%	17m
22	$2^{72}3^44^16^2$	$2^{20}3^2$	9.3MB	164	100%	2s	164.70	100%	3m	144	100%	8m
23	$2^{25}3^16^1$	$2^{13}3^2$	352.7kB	48	100%	1s	55.40	100%	3s	37.30	100%	3m
24	$2^{110}3^25^36^4$	$2^{25}3^4$	34.5MB	341	100%	5s	337.70	100%	25m	337.70	100%	1.6h
25	$2^{118}3^64^25^26^6$	$2^{23}3^34^1$	54.3MB	404	100%	7s	407.70	100%	47m	407.70	100%	2.6h
26	$2^{87}3^14^35^4$	$2^{28}3^4$	16.8MB	207	100%	3s	205.10	100%	7m	195.30	100%	47m
27	$2^{55}3^24^15^16^2$	$2^{17}3^3$	5.1MB	204	100%	2s	210.90	100%	2m	180.50	100%	10m
28	$2^{167}3^{16}4^25^36^6$	$2^{31}3^6$	160.7MB	420	100%	21s	421.80	100%	2.6h	421.80	100%	4.6h
29	$2^{134}3^75^3$	$2^{19}3^3$	52.4MB	154	100%	5s	156.10	100%	20m	125.70	100%	43m
30	$2^{73}3^34^3$	$2^{31}3^4$	8.5MB	100	100%	2s	93.70	100%	2m	73.80	100%	14m
apache	$2^{158}3^84^55^16^1$	$2^33^14^25^1$	92.5MB	173	100%	9s	191.60	100%	36m	168.20	100%	1.7h
bugzilla	$2^{49}3^14^2$	2^43^1	2.3MB	68	100%	1s	72.20	100%	22s	49.50	100%	9m
gcc	$2^{189}3^{10}$	$2^{37}3^3$	127.6MB	108	100%	10s	121	100%	43m	81.80	100%	1.4h
spins	$2^{13}4^5$	2^{13}	156.2kB	98	100%	1s	112.80	100%	2s	105.60	100%	3m
spinv	$2^{42}3^24^{11}$	$2^{47}3^2$	4.3MB	286	100%	2s	251.70	100%	2m	238.90	100%	1.2h
Segall et al. [27]												
Banking1	3^44^1	5^{112}	3.8kB	58	100%	2s	55.10	100%	0s	45	100%	30s
Banking2	$2^{14}4^1$	2^3	51.2kB	39	100%	1s	44.70	100%	0s	30	100%	3m
CommProtocol	$2^{10}7^1$	$2^{10}3^{10}4^{12}5^{24}$ $6^{30}7^{30}8^{12}$	26.0kB	49	100%	3s	50.30	100%	0s	41	100%	3m
Concurrency	2^5	$2^43^15^2$	0.9kB	8	100%	1s	8	100%	0s	8	100%	0s
Healthcare1	$2^63^25^16^1$	2^33^{18}	31.9kB	105	100%	1s	107.50	100%	0s	96	100%	9s
Healthcare2	$2^53^64^1$	$2^13^65^{18}$	48.2kB	67	100%	1s	68.40	100%	0s	54.80	100%	3m
Healthcare3	$2^{16}3^64^55^16^1$	2^{31}	918.8kB	209	100%	1s	205.70	100%	15s	177.10	100%	41m
Healthcare4	$2^{13}3^{12}4^65^26^17^1$	2^{22}	2.2MB	294	100%	1s	309	100%	39s	274.90	100%	53m
Insurance	$2^63^15^26^211^113^117^131^1$	-	1.3MB	6866	100%	1s	6861.10	100%	3m	6858.40	100%	15m
NetworkMgmt	$2^24^15^310^211^1$	2^{20}	189.4kB	1125	100%	1s	1107.70	100%	4s	1100.40	100%	2m
ProcessorComm1	$2^33^64^9$	2^{13}	172.7kB	163	100%	1s	144.10	100%	2s	131.60	100%	3m
ProcessorComm2	$2^33^{12}4^85^2$	1^42^{121}	1015.3kB	161	100%	2s	169.30	100%	11s	145.50	100%	31m
Services	$2^33^45^28^210^2$	$3^{386}4^2$	365.6kB	963	100%	6s	926.80	100%	13s	926.80	100%	5.7h
Storage1	$2^13^14^15^1$	4^{95}	1.8kB	25	100%	2s	25	100%	0s	25	100%	0s
Storage2	3^46^1	-	5.1kB	74	100%	0s	71.50	100%	0s	54	100%	1s
Storage3	$2^93^15^36^18^1$	$2^{38}3^{10}$	184.4kB	239	100%	1s	239.20	100%	3s	222	100%	9m
Storage4	$2^53^74^15^26^27^110^113^1$	2^{24}	1.0MB	990	100%	1s	970.40	100%	28s	916.40	100%	15m
Storage5	$2^53^55^628^19^110^211^1$	2^{151}	2.1MB	1879	100%	4s	1936.10	100%	3m	1000.50	96%	12h
SystemMgmt	$2^53^45^1$	$2^{13}3^4$	26.7kB	60	100%	1s	58.10	100%	0s	45	100%	2s
Telecom	$2^53^14^25^16^1$	$2^{11}3^14^9$	43.2kB	126	100%	1s	125.20	100%	0s	120	100%	5s
Yu et al. [31]												
RL-A-mod	$2^53^44^75^46^57^48^112^3$	$1^{12}2^{491}3^{345}$	8.6MB	1132	100%	16s	1079.40	100%	4m	1069.20	100%	7.8h
RL-B-mod	$2^83^24^35^36^19^1$ $10^112^{14}20^124^137^1$	$1^82^{1127}3^{277}$ $4^{1755}5^{1064}6^{2048}$	16.4MB	14977	100%	4m	13319.40	100%	3.1h	4954	92%	12h
Yamada et al. [29]												
Company2	$2^63^48^4$	$1^22^{35}3^{89}4^{54}5^{34}$ $6^{20}7^{34}8^{16}9^4$	247.9kB	424	100%	15s	432.50	100%	7s	427.20	100%	54m

Table 2 Test suite size, Percentage of tuple coverage and Time for $t = 4$ and $t = 5$. In bold the method with better results with the lexicographic criteria (coverage percentage, number of tests, exhausted time). For the coverage percentage enough precision was taken into account. Resources: 12GB memory and 12h timeout.

family		inst	mem	t = 4						t = 5														
				ACTS		BOT-its		PBOT-its		mem	ACTS		BOT-its		PBOT-its									
				tests	%	time	tests	%	time		tests	%	time	tests	%	time	tests	%	time					
Cohen et al.	1	1.4GB	1080	100%	9m	-	-	1721.30	100%	2.6h	1111.70	47%	12h	74.2GB	-	-	-	1603.10	15%	12h	938	3%	12h	
	2	1.0GB	873	100%	4m	-	-	846.60	100%	1.5h	736.70	100%	6.9h	48.8GB	-	-	-	1144.90	35%	12h	964.50	11%	12h	
	3	7.5MB	212	100%	2s	253.90	100%	2m	253.90	100%	2m	176.80	100%	1.1h	99.3MB	593	100%	7s	747.60	100%	46m	507.20	100%	6.2h
	4	147.8MB	374	100%	10s	433.90	100%	1.1h	430.40	100%	55m	339	100%	3.8h	4.2GB	-	-	-	1010.70	100%	3.9h	1425.30	49%	12h
	5	14.1GB	2442	100%	2.5h	-	-	2514.70	100%	11.3h	1111.10	41%	12h	1.3TB	-	-	-	1603.90	1%	12h	1546.70	1%	12h	
	6	422.4MB	491	100%	47s	-	-	557	100%	43m	426.70	100%	1.7h	16.0GB	-	-	-	833.50	60%	12h	811.60	38%	12h	
	7	7.1MB	93	100%	2s	112.50	100%	54s	112.50	100%	55s	88.90	100%	27m	94.2MB	244	100%	9s	295.80	100%	17m	235.20	100%	2.0h
	8	2.9GB	1988	100%	25m	-	-	1826	100%	5.3h	1089.80	47%	12h	184.2GB	-	-	-	1993.80	5%	12h	1448.90	1%	12h	
	9	172.9MB	268	100%	15s	347.10	100%	54m	345.80	100%	50m	228.60	100%	2.9h	5.2GB	-	-	-	1140.70	100%	5.8h	760.90	67%	12h
	10	7.2GB	2063	100%	1.2h	-	-	2128.50	99%	12h	1285.30	46%	12h	579.4GB	-	-	-	1589.90	2%	12h	1231.30	1%	12h	
	11	1.4GB	1885	100%	10m	-	-	1984.10	100%	2.4h	1776.60	47%	12h	74.0GB	-	-	-	3752.40	9%	12h	3409.40	3%	12h	
	12	6.1GB	1465	100%	47m	-	-	1491.80	100%	6.1h	1393.70	81%	12h	475.5GB	-	-	-	1270.80	4%	12h	922.40	1%	12h	
	13	4.0GB	1040	100%	22m	-	-	1087.90	100%	4.6h	958.40	100%	8.2h	273.4GB	-	-	-	1508.90	5%	12h	1482.90	2%	12h	
	14	1.1GB	1163	100%	5m	-	-	1250.20	100%	1.3h	1145.10	100%	10.4h	52.1GB	-	-	-	2127.90	27%	12h	1998.10	8%	12h	
	15	171.5MB	770	100%	20s	819.90	100%	2.5h	815.40	100%	2.2h	710.10	100%	10.1h	5.1GB	-	-	-	3335.10	100%	6.3h	1363	45%	12h
	16	691.4MB	453	100%	2m	-	-	539.50	100%	1.1h	411.10	100%	3.5h	29.7GB	-	-	-	655.50	35%	12h	624.10	1%	12h	
	17	4.5GB	1514	100%	32m	-	-	1446.70	100%	3.9h	1331.80	100%	8.6h	325.2GB	-	-	-	1861	4%	12h	1643.70	2%	12h	
	18	6.0GB	2145	100%	52m	-	-	2020.90	100%	4.8h	1052	42%	12h	465.5GB	-	-	-	1391.70	3%	12h	1346.80	1%	12h	
	19	28.7GB	2535	100%	5.1h	-	-	689.10	99%	12h	665.70	38%	12h	2.5TB	-	-	-	1415.90	1%	12h	847.40	1%	12h	
	20	11.1GB	3278	100%	2.3h	-	-	1424.20	99%	12h	101.4	28%	12h	998.5GB	-	-	-	2088.50	1%	12h	1002	1%	12h	
	21	796.2MB	1070	100%	3m	-	-	1149.90	100%	1.6h	1039.20	100%	9.2h	35.3GB	-	-	-	3012.90	17%	12h	1079.70	7%	12h	
	22	511.3MB	664	100%	1m	-	-	762.90	100%	59m	611.30	100%	5.5h	20.3GB	-	-	-	939.30	75%	12h	766.30	26%	12h	
	23	5.9MB	140	100%	2s	162.50	100%	55s	162.50	100%	57s	122.50	100%	36m	73.5MB	375	100%	6s	455.10	100%	17m	347.60	100%	3.1h
	24	2.9GB	2105	100%	25m	-	-	2052.30	100%	4.2h	1007.20	45%	12h	184.0GB	-	-	-	2236.20	6%	12h	1851.40	2%	12h	
	25	5.3GB	2673	100%	55m	-	-	2681.90	100%	6.3h	1214.60	42%	12h	394.1GB	-	-	-	1613	3%	12h	1474.70	1%	12h	
	26	1.1GB	1111	100%	3m	-	-	1054.50	100%	1.9h	1004.50	100%	7.9h	55.1GB	-	-	-	1669.50	20%	12h	1167.80	3%	12h	
	27	224.5MB	1004	100%	35s	-	-	1034.20	100%	3.9h	931.80	60%	12h	7.2GB	-	-	-	4486.80	99%	12h	1343.90	16%	12h	
	28	22.7GB	2888	100%	5.1h	-	-	887.30	69%	12h	866.20	26%	12h	2.4TB	-	-	-	1842.20	1%	12h	1392.50	1%	12h	
	29	5.1GB	681	100%	22m	-	-	758.70	100%	3.0h	604.40	100%	7.7h	374.4GB	-	-	-	746.70	6%	12h	712.90	2%	12h	
	30	456.4MB	386	100%	56s	-	-	392.20	100%	4.0h	313.90	100%	2.4h	17.6GB	-	-	-	517.40	64%	12h	407	37%	12h	
Segall et al.	apache	10.9GB	838	100%	1.1h	-	-	944.30	100%	3.4h	814.50	100%	9.3h	971.3GB	-	-	-	1027.80	3%	12h	982.20	1%	12h	
	bugzilla	79.4MB	242	100%	3s	275.70	100%	23m	277.30	100%	21m	192	100%	1.7h	1.9GB	752	100%	4m	932.60	100%	1.3h	643.10	100%	6.8h
	gcc	16.7GB	444	100%	1.2h	-	-	535.90	100%	5.6h	364.80	100%	8.0h	1.6TB	-	-	-	436.40	3%	12h	353.80	1%	12h	
	spins	1.9MB	393	100%	1s	431.20	100%	36s	431.20	100%	37s	396.10	100%	2.3h	16.6MB	1449	100%	2s	1448.80	100%	14m	1390.30	100%	6.2h
	spinw	181.2MB	1631	100%	38s	1377.60	100%	5.9h	1380.80	100%	4.6h	800.40	65%	12h	5.5GB	8202	100%	1.1h	1962.70	89%	12h	763	7%	12h
	Banking1	8.0kB	139	100%	2s	150.20	100%	0s	150.20	100%	0s	139	100%	3m	6.3kB	212	100%	2s	212	100%	0s	212	100%	0s
	Banking2	432.2kB	96	100%	1s	109.20	100%	3s	109.20	100%	3s	87.30	100%	10m	2.4MB	232	100%	1s	262.90	100%	30s	225.30	100%	1.5h
	CommProtocol	157.5kB	97	100%	3s	100.10	100%	1s	100.10	100%	1s	86	100%	3m	616.9kB	167	100%	4s	167	100%	5s	102.50	100%	19m
	Concurrency	1.2kB	8	100%	1s	8	100%	1s	8	100%	1s	8	100%	1s	0.6kB	8	100%	1s	8	100%	1s	8	100%	1s
	Healthcare1	201.2kB	341	100%	1s	331.60	100%	3s	331.60	100%	3s	300	100%	21m	801.8kB	814	100%	1s	829.40	100%	21s	829.40	100%	22s
	Healthcare2	379.6kB	220	100%	1s	233.50	100%	4s	233.50	100%	4s	216.90	100%	52m	1.9MB	708	100%	1s	716.80	100%	45s	716.80	100%	46s
	Healthcare3	21.4MB	1004	100%	4s	997.80	100%	15m	997.80	100%	15m	904.70	100%	9.3h	366.0MB	4239	100%	78s	4206.30	99%	12h	1846.20	57%	12h
	Healthcare4	73.2MB	1644	100%	8s	1792.50	100%	1.2h	1786.60	100%	1.0h	1035.20	77%	12h	1.7GB	8204	100%	7m	2843.30	23%	12h	2367	11%	12h
	Insurance	27.3MB	75764	100%	4s	69969.70	99%	12.0h	69377.70	99%	12.0h	67183.30	94%	12h	363.7MB	4947.48	100%	53s	27183.50	11%	12h	26578.20	6%	12h
	NetworkMgmt	2.0MB	6267	100%	2s	6143.10	100%	3m	6143.10	100%	3m	5276.20	99%	12h	12.6MB	29272	100%	6s	29764.60	100%	1.7h	17096.20	93%	12h
	ProcessorComm1	2.1MB	670	100%	1s	638.30	100%	41s	638.30	100%	41s	586.60	100%	5.1h	18.5MB	2588	100%	3s	2589.20	100%	19m	1088.10	95%	12h
	ProcessorComm2	24.3MB	744	100%	6s	789.20	100%	7m	744	100%	7m	688.30	100%	7.1h	424.0MB	3094	100%	40s	3333	100%	4.2h	1132	84%	12h
	Services	5.4MB	6855	100%	9s	6606.60	100%	16m	6606.60	100%	17m	6539.20	100%	4.3h	52.5MB	37393	100%	22s	17868.60	93%	12h	5805.60	65%	12h
Storage1	1.9kB	25	100%	2s	25	100%	0s	25	100%	0s	25	100%	0s	0.1kB	-	-	-	-	-	-	-	-	-	0s
Storage2	11.4kB	195	100%	0s	105.50	100%	0s	105.50	100%	0s	162	100%	4s	9.5kB	486	100%	0s	486	100%	0s	486	100%	0s	
Storage3	2.2MB	752	100%	2s	755.30	100%	54s	755.30	100%	55s	686.90	100%	5.5h	18.8MB	2106	100%	4s	2265.10	100%	14m	938	97%	12h	
Storage4	23.2MB	6636	100%	5s	6729	100%	49m	6735.60	100%	50m	2007.60	93%	12h	372.2MB	39490	100%	37s	-	-	-	14598.40	97%	12h	
Storage5	61.7MB	13292	100%	20s	13183.10	100%	7.9h	13183.10	100%	7.9h	1608	73%	12h	1.3GB	78464	100%	9m	-	-	-	5395.30	7%	12h	
SystemMgmt	162.5kB	152	100%	1s	151.20	100%	1s	151.20	100%	1s	135	100%	18s	629.8kB	317	100%	1s	333.60	100%	6s	333.60	100%	6s	
Telecom	302.9kB	392	100%	1s	404	100%	3s	404	100%	3s	361.40	100%	3m	1.3MB	1110	100%								

required by other auxiliary data structures or by the SAT solver called within BOT-its. Tables 1 and 2 show the result of our analysis under column *mem*. For $t = 4$ there are 20 out of the 58 instances that would consume more than 1GB. For $t = 5$ the memory consumption is greatly increased, as 23 of the 58 instances would consume more than 32GB (some of these instances would need more than 1TB). Therefore, it is obvious we can not aim to run any approach that explicitly considers all allowed t -tuples or tests at once under low memory requirements.

The third question we address is whether the Pool-based versions of BOT-its and RBOT-its are efficient compared to ACTS for $t = 4$ and $t = 5$. For both PRBOT-its and PBOT-its (as PRBOT-its but *refine* is deactivated) we consider a pool budget of 1GB (1278264 tuples for $t = 4$ and 721600 for $t = 5$). For $t = 4$ the combination of PBOT-its and PRBOT-its report better sizes than ACTS and BOT-its in 35 of the 58 instances. Finally, for $t = 5$ we found that ACTS and BOT-its can only report test suites for 39 and 18 instances respectively, while PBOT-its and PRBOT-its can report test suites for all the 57 instances⁸.

Overall, we found that ACTS reports MCACs in 49 more instances than RBOT-its and PRBOT-its. However, we may be observing an horizon effect, as RBOT-its and PRBOT-its with the given resources are able to improve the results of ACTS in 89 out of 107 instances where both these algorithms and ACTS reach 100% of coverage, where ACTS only obtains better results in 8 (the remaining 10 are ties).

Regarding run times, ACTS is significantly faster than BOT-its, RBOT-its, PBOT-its and PRBOT-its. However, ACTS will report the same suboptimal solution with more available run time. In contrast, RBOT-its, and PRBOT-its can get better solutions if we increase the timeout for the MaxSAT call related to the refining process.

A more fine grained analysis on the new methods reveals the following insights.

We observe PBOT-its subsumes BOT-its, as it can obtain an MCAC on the same instances as BOT-its plus 23 and 7 more for $t = 4$ and $t = 5$ respectively. Regarding MCAC sizes we observe similarities with the results reported by BOT-its. Regarding run times we found that PBOT-its can obtain MCACs slightly faster than BOT-its.

Finally, we also note that with enough run time, RBOT-its and PRBOT-its algorithms would subsume BOT-its and PBOT-its respectively. In particular, results show that the *refine* approach can reduce the sizes on 92 out of the 106 instances where all these algorithms are able to obtain an MCAC, while for the remaining 14 instances they report the same sizes. In these particular cases, we observe that *refine* has not been able to improve the size of the window within the given time constraints, so these results could be improved by tuning the time limits, the MaxSAT solver's parameters or even using a different MaxSAT solver.

To conclude this section, it seems we can confirm the goodness of the PRBOT-its algorithm. We have shown how the *refine* method can be used to improve the sizes of the reported suboptimal MCACs. Additionally, we extended the practical usage of algorithm BOT-its to strengths higher than $t = 3$.

8 Conclusions and Future Work

Bugs or failures involving 4 or 5 parameters (even more) do exist and are likely to arise in complex systems. We have provided an effective approach to compute MCACs of such strength with low memory requirements. This low memory consumption plus the partitioning nature of the Pool based approach opens the avenue for more practical parallelized approaches.

⁸ For instance *Storage1* it is not possible to report an MCAC for $t = 5$ as it only has 4 parameters.

References

- 1 Carlos Ansótegui and Felip Manyà. Mapping problems with finite-domain variables into problems with boolean variables. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, pages 1–15, 2004.
- 2 Carlos Ansótegui, Felip Manyà, Jesus Ojeda, Josep M. Salvia, and Eduard Torres. Incomplete maxsat approaches for combinatorial testing. *arXiv*, abs/2105.12552, 2021. [arXiv:2105.12552](#).
- 3 Carlos Ansótegui, Jesus Ojeda, António Pacheco, Josep Pon, Josep M. Salvia, and Eduard Torres. Optilog: A framework for sat-based systems. 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 1–10. Springer, 2021. doi:10.1007/978-3-030-80223-3_1.
- 4 Carlos Ansótegui, Idelfonso Izquierdo, Felip Manyà, and José Torres Jiménez. A max-sat-based approach to constructing optimal covering arrays. *Frontiers in Artificial Intelligence and Applications*, 256:51–59, 2013.
- 5 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. MaxSAT Evaluation 2019 : Solver and Benchmark Descriptions. Technical Report Department of Computer Science Report Series B-2019-2, University of Helsinki, 2019.
- 6 Mutsunori Banbara, Haruki Matsunaka, Naoyuki Tamura, and Katsumi Inoue. Generating combinatorial test cases by efficient sat encodings suitable for cdcl sat solvers. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 112–126, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- 7 Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. In *SAT Competition 2013*, 2013.
- 8 Armin Biere. CaDiCaL at the SAT Race 2019. In Marijn Heule, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications B*, pages 8–9. University of Helsinki, 2019.
- 9 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
- 10 Mehra N. Borazjany, Linbin Yu, Yu Lei, Raghu Kacker, and Rick Kuhn. Combinatorial testing of ACTS: A case study. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 591–600, 2012.
- 11 Renée C. Bryce, Charles J. Colbourn, and Myra B. Cohen. A framework of greedy methods for constructing interaction test suites. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 146–155, 2005.
- 12 Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.
- 13 Jacek Czerwónka. Pairwise testing in real world. In *Proc. of the Twenty-fourth Annual Pacific Northwest Software Quality Conference, 10-11 October 2006, Portland, Oregon*, pages 419–430, 2006.
- 14 Feng Duan, Yu Lei, Linbin Yu, Raghu N. Kacker, and D. Richard Kuhn. Optimizing ipog’s vertical growth with constraints based on hypergraph coloring. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*, pages 181–188, 2017.
- 15 Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, January 2006. Publisher: IOS Press.

- 16 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.
- 17 Arnaud Gotlieb, Aymeric Hervieu, and Benoit Baudry. Minimum pairwise coverage using constraint programming techniques. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 773–774, 2012. doi:10.1109/ICST.2012.174.
- 18 Aymeric Hervieu, Dusica Marijan, Arnaud Gotlieb, and Benoit Baudry. Practical minimization of pairwise-covering test configurations using constraint programming. *Information and Software Technology*, 71:129–146, 2016. doi:10.1016/j.infsof.2015.11.007.
- 19 Brahim Hnich, Steven Prestwich, and Evgeny Selensky. Constraint-based approaches to the covering test problem. In Boi V. Faltings, Adrian Petcu, François Fages, and Francesca Rossi, editors, *Recent Advances in Constraints*, pages 172–186, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- 20 Brahim Hnich, Steven D. Prestwich, Evgeny Selensky, and Barbara M. Smith. Constraint Models for the Covering Test Problem. *Constraints*, 11(2):199–219, July 2006.
- 21 Serdar Kadioglu. Column generation for interaction coverage in combinatorial software testing, 2017. arXiv:1712.07081.
- 22 D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng.*, 30(6):418–421, 2004.
- 23 Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–64, 2010. Publisher: IOS Press.
- 24 Elizabeth Maltais and Lucia Moura. Finding the best CAFE is np-hard. In *LATIN 2010: Theoretical Informatics, 9th Latin American Symposium, Oaxaca, Mexico, April 19-23, 2010. Proceedings*, pages 356–371, 2010.
- 25 Toru Nanba, Tatsuhiro Tsuchiya, and Tohru Kikuno. Using satisfiability solving for pairwise testing in the presence of constraints. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E95.A(9):1501–1505, 2012.
- 26 Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):1–29, 2011.
- 27 Itai Segall, Rachel Tzoref-Brill, and Eitan Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, page 254–264, New York, NY, USA, 2011. Association for Computing Machinery.
- 28 Evgeny Selensky. CSPLib problem 045: The covering array problem. <http://www.csplib.org/Problems/prob045>.
- 29 Akihisa Yamada, Armin Biere, Cyrille Artho, Takashi Kitamura, and Eun-Hye Choi. Greedy combinatorial test case generation using unsatisfiable cores. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, page 614–624, New York, NY, USA, 2016. Association for Computing Machinery.
- 30 Akihisa Yamada, Takashi Kitamura, Cyrille Artho, Eun-Hye Choi, Yutaka Oiwa, and Armin Biere. Optimization of combinatorial testing by incremental SAT solving. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–10, 2015.
- 31 L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn. Constraint handling in combinatorial test generation using forbidden tuples. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–9, 2015.

On How Turing and Singleton Arc Consistency Broke the Enigma Code

Valentin Antuori ✉

Renault, LAAS-CNRS, Université de Toulouse, CNRS, France

Tom Portoleau ✉

LAAS-CNRS, Université de Toulouse, CNRS, France

Louis Rivière ✉

LAAS-CNRS, Université de Toulouse, CNRS, ANITI, France

Emmanuel Hebrard ✉ 

LAAS-CNRS, Université de Toulouse, CNRS, ANITI, France

Abstract

In this paper, we highlight an intriguing connection between the cryptographic attacks on Enigma's code and local consistency reasoning in constraint programming.

The coding challenge proposed to the students during the 2020 ACP summer school, to be solved by constraint programming, was to decipher a message encoded using the well known Enigma machine, with as only clue a tiny portion of the original message. A number of students quickly crafted a model, thus nicely showcasing CP technology – as well as their own brightness. The detail that is slightly less favorable to CP technology is that solving this model on modern hardware is challenging, whereas the “Bombe”, an antique computing device, could solve it eighty years ago.

We argue that from a constraint programming point of view, the key aspects of the techniques designed by Polish and British cryptanalysts can be seen as, respectively, path consistency and singleton arc consistency on some constraint satisfaction problems.

2012 ACM Subject Classification Mathematics of computing → Combinatoric problems; Mathematics of computing → Combinatorial optimization

Keywords and phrases Constraint Programming, Cryptography

Digital Object Identifier 10.4230/LIPIcs.CP.2021.13

Supplementary Material *Software (Source Code)*: <https://gitlab.laas.fr/vantuori/sacnigma>
archived at `swb:1:dir:98f5264c0b6821dbf5caf769a2ebf70e4cda5b92`

1 Introduction

Enigma was a cipher machine that had been commercialised since 1923. Breaking its code was a decisive breakthrough with a significant impact on the outcome of World War II.

The machine resembles a portable typewriter. Once configured in a particular setting agreed upon by the sender and the receiver, it can be used to encrypt a message. When typing with the machine, each letter is ciphered to a seemingly random letter indicated by a light bulb. The encrypted message, or *ciphertext*, can be deciphered by the receiver using his own Enigma machine. The code is indeed symmetric and typing the ciphertext with the same machine setting yields the original message.

The Enigma code was first broken by the mathematicians Marian Rejewski, Jerzy Różycki and Henryk Zygalski for the Polish Cipher Bureau before the war, although this method relied on a weakness due to an operating practice that was abandoned during the war. This knowledge on the machine, however, was shared with the allies and helped British cryptanalysts to break the code. To this end, Alan Turing and Gordon Welchman designed “The Bombe”, an electro-mechanical device that made it possible to decipher Enigma's encrypted messages until the end of the war.



© Valentin Antuori, Tom Portoleau, Louis Rivière, and Emmanuel Hebrard;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 13; pp. 13:1–13:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper, we look back to this story from the viewpoint of constraint programming. We first describe a constraint model to break Enigma's code in Section 4. Given a portion of the original message (the *crib*), a solution of this model represents the internal settings of Enigma (the *cryptographic key*) that produced, and can decrypt, the intercepted communication. Modelling this constraint satisfaction problem was the topic of a “hackathon” held during the ACP summer school in 2020, and several participants managed to break Enigma's code during this event.¹ This problem would be deceptively tricky to tackle without a rich modelling framework, and it nicely illustrates the effectiveness of constraint programming in that respect. However, solving this model with state-of-the-art solvers appears to be challenging, which highlights the prowess that was solving this problem eighty years ago. Interestingly, it appears that both methods used by the Polish Cipher Bureau or at Bletchley Park can be equated to known concepts of consistency: *Singleton Arc Consistency* on the constraint model introduced in this paper for the latter (see Section 5), and *Path Consistency* on some precomputed constraints for the former (see Section 6).

2 The Constraint Satisfaction Problem and Consistency

A Constraint Satisfaction Problem (CSP) is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where $\mathcal{X} = \{x_1, \dots, x_n\}$, is a set of *variables*; $\mathcal{D} = \{D(x_1), \dots, D(x_n)\}$ a set of *domains*; and $\mathcal{C} = \{c_1, \dots, c_t\}$ a set of *constraints*. An *assignment* maps² values to variables, we write $x \leftarrow v$ for the assignment of value v to variable x . A constraint c_j is given by a pair $(S(c_j), R(c_j))$ where $S(c_j)$ is a subset of \mathcal{X} , and $R(c_j)$ is $|S(c_j)|$ -ary relation, that is, a set of satisfying assignments of $S(c_j)$.

The assignment $A = \{x_1 \leftarrow v_1, \dots, x_k \leftarrow v_k\}$ of the set of variables $\{x_1, \dots, x_k\}$ is *consistent* for a constraint c if and only if its projection $\{x_i \leftarrow v_i \mid 1 \leq i \leq k \ \& \ x_i \in S(c)\}$ to $S(c)$ can be extended to an assignment in $R(c_j)$; Assignment A is *globally consistent* if and only if it is consistent for every constraint in \mathcal{C} ; it is *valid* if and only if, for every variable x_i , we have $v_i \in D(x_i)$. A *solution* of a CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is a valid, globally consistent assignment of \mathcal{X} . We write $\mathcal{D}|_{x \leftarrow v}$ for the set of domains where x is mapped to $\{v\}$ and equal to \mathcal{D} on all other variables, and $\mathcal{D}' \subseteq \mathcal{D}$ when $\forall x \in \mathcal{X}, D'(x) \subseteq D(x)$.

The notion of consistency is key to solving constraint satisfaction problems. We define here three consistencies that we shall relate to historical methods for attacking Enigma's code. These definitions are standard generalisations to non-binary constraints. In particular, the definition of consistency of an assignment for a constraint as its *projection* to the constraint's scope being *extendable* to the constraint's relation is useful to generalize path consistency.

► **Definition 1.** A support for a constraint c is a valid and consistent assignment of $S(c)$.

► **Definition 2** (Arc Consistency (AC) [6]). A variable x is arc consistent (AC) with respect to a constraint c if and only if, for each $v \in D(x)$, there exists a support for c that contains $x \leftarrow v$. A constraint c is AC if and only if every variable $x \in S(c)$ is AC with respect to c . A CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is AC if and only if every constraint $c \in \mathcal{C}$ is AC.

► **Definition 3** (Singleton Arc Consistency (SAC) [3]). An instantiation $x \leftarrow v$ is singleton arc consistent (SAC) if and only if there exists $\mathcal{D}' \subseteq \mathcal{D}|_{x \leftarrow v}$ such that the CSP $(\mathcal{X}, \mathcal{D}', \mathcal{C})$ is AC. A variable x is SAC if and only if, and for each $v \in D(x)$, $x \leftarrow v$ is SAC. A CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is SAC if and only if every variable $x \in \mathcal{X}$ is SAC.

¹ <https://acp-iaro-school.sciencesconf.org/>

² We use functions instead of tuples to make variable ordering irrelevant.

► **Definition 4** (Path Support). *Given a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ and three distinct variables x_1, x_2 and x_3 , the assignment $\{x_3 \leftarrow v_3\}$ is a path support of assignment $\{x_1 \leftarrow v_1, x_2 \leftarrow v_2\}$ for variable x_3 if and only if the assignment $\{x_1 \leftarrow v_1, x_2 \leftarrow v_2, x_3 \leftarrow v_3\}$ is valid and globally consistent.*

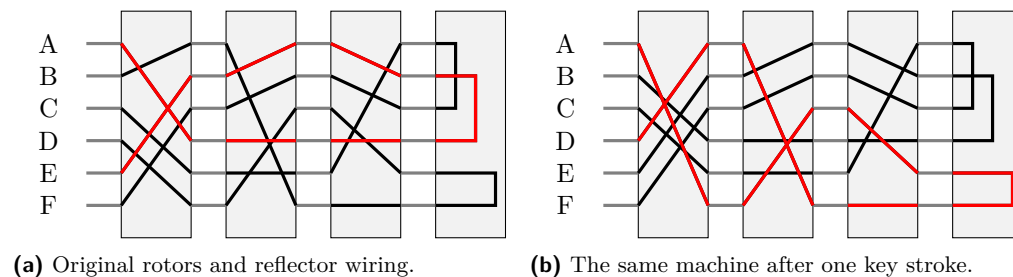
► **Definition 5** (Path Consistency (PC) [7]). *An assignment $\{x_1 \leftarrow v_1, x_2 \leftarrow v_2\}$ of two variables is path consistent (PC) if and only if it has a path support for every variable.*

A CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is PC if and only if every valid and globally consistent assignment of every pair of variables is path consistent.

Enforcing (singleton) *AC* on a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ means finding the largest domain \mathcal{D}' , in the sense of inclusion as defined above, such that $\mathcal{D}' \subseteq \mathcal{D}$ and $(\mathcal{X}, \mathcal{D}', \mathcal{C})$ is (singleton) *AC*. Notice that there is a single such fix point, since the set of possible domains forms a lattice where infimum and supremum are obtained respectively by the intersection and the union.

3 The Enigma Code

In its simplest form, Enigma's encryption system is composed of three *rotor* wheels and a *reflector* wheel. A rotor wheel can be seen as a *simple substitution cipher* whereby every letter is mapped to a given letter, forming a permutation of the alphabet. The signal then goes through the two other wheels, then through the reflector and finally through the three rotors but backwards. Figure 1a illustrates, on a reduced alphabet, the rotor wheels (first three boxes) and reflectors (last box) wiring the input keyboard to an output system composed of lightbulbs indicating the substituted letter. In this case, pressing the key A eventually lights the bulb E. In other words, this mechanism is a simple substitution cipher whereby the alphabet is applied a permutation, e.g., (AE)(BD)(CF) in Figure 1a. Notice that the reflector wheel is symmetric and antireflexive. As a result, the overall permutation is symmetric and the same machine can therefore be used to decipher: pressing the key E lights bulb A. Moreover, it is antireflexive: no letter is mapped to itself, which proved to be a weakness.



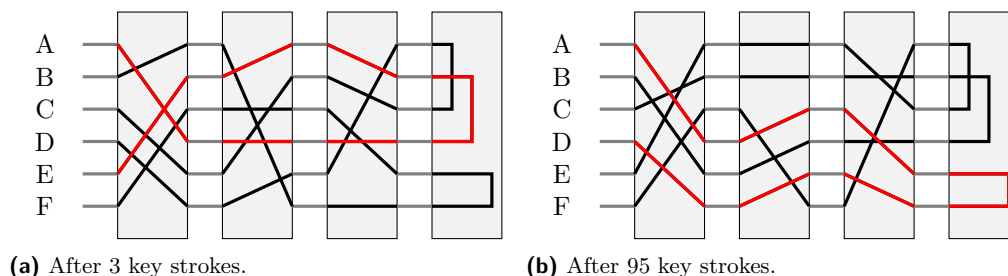
■ **Figure 1** Illustration of Enigma's rotors and reflector.

However, the feature that made Enigma such a strong cryptographic system is that rotors are not static: they advance at every key stroke. More precisely, at every key stroke the “fast” rotor (leftmost) advances one step. Figure 1b shows the same machine after one key stroke. The fast rotor is shifted down by one position: the wire that was connecting B to A now connects A to F, and so forth. As a result, pressing the same letter A now lights bulb D.

Moreover, when the fast rotor has completed a turn and is back to its reference position, a turnover notch is activated, and the middle rotor advances one step. Similarly, when the middle rotor has completed a turn, the “slow” (rightmost) rotor advances one step. Figure 2 shows the same machine after 6 key strokes (the fast rotor has made a full turn and is back

13:4 On How Turing and Singleton Arc Consistency Broke the Enigma Code

to its original position and the middle rotor has advanced 1 step), or after 95 key strokes (the fast rotor has advanced 5 steps from its reference position, the middle rotor 3 steps and the slow rotor 2 steps). The simple substitution cipher in the latter position is (AD)(BC)(EF).



■ **Figure 2** Illustration of the same machine as in Figure 1a.

All Enigma machines shared the same set of three rotor wheels and the same reflector wheel, however, rotors could be rearranged in any order and initialised in any position. For instance, the daily settings could be given as “312, CSP”. In that case, on that day, the 3rd rotor wheel would be placed on the left, the 1st wheel on the middle and the 2nd wheel on the right. Moreover, before (de)ciphering any message, the left, middle and right rotor wheels would be positioned respectively so that the letters C, S and P showed up in some windows designed for this purpose. Additionally, the positions of the turnover notches of the left and middle rotor wheels, indicating the reference position at which the next rotor to the right would advance, could also be changed. The cryptographic key shared by the sender and receiver of the message was therefore the $3!$ rotor orders and the 26^3 rotor positions, as well as the 26^2 reference positions.³ Therefore, there are $3! \times 26^3 \times 26^2 = 72188256$ possible keys.⁴

The first versions of the machine, commercialised in the 1920's operated on those principles, and hence had relatively few possible cryptographic keys. The version used by the German military added one sophistication: a *plugboard* (or *steckerbrett*) inserted between the input keyboard and the rotor scrambler, and also between the scrambler and the output light bulbs. The plugboard is also a simple substitution cipher, mapping L letters to another – different – letter, and the remaining $26 - 2L$ to themselves. However, this further cipher could easily be changed manually and was therefore part of the cryptographic key. For $L = 10$ plugs, the number of possibilities is 150,738,274,937,250, and the total number of possible settings is thus 158,962,555,217,826,360,000 and even 26^2 times more counting the reference positions.

Breaking the code entails finding, for a given ciphertext, the *rotor settings* (the choice of rotors, their order, the reference position of the two leftmost rotors, and their initial position) and the *plugboard configuration*. Those settings can be seen as the cryptographic key that one needs to recover in order to decipher the messages. However, if every component contributes to the combinatorial number of keys, the rotor position is the most important. Indeed, the reference positions do not affect the first letters of the message. As long as the middle rotor does not advance, the text is unchanged. Moreover, the reference position of the first rotor can be deduced from when the text starts being gibberish. The plugboard configuration only affects part of the letters, for instance with six plugs, fourteen letters are not changed. Therefore given a correct guess on the order and position for the rotors, setting up the reference position arbitrary (say to AA) and the plugboard to the identity, would be sufficient

³ Only the left and middle rotors have a turnover notch.

⁴ Later versions introduced two more rotors to choose from, raising this number to ${}^3P_5 \times 26^5 = 721882560$.

to decrypt a small portion of the message. Therefore, verifying that this guess is correct is relatively easy, as well as deducing the reference position and the plugboard configuration. Finally, the rotor ordering (and choice thereof) was often guessed by other means than computation, or the attack was repeated for every possible order until it succeeded.

4 A Constraint Programming Model

In this section we introduce a constraint model that emulates the Enigma machine, and can be used to break its code. This model assumes that the rotors and their order are known. Since it is actually unknown, the problem might have to be solved $3! = 6$ (resp. ${}^3P_5 = 60$) times for three rotor wheels (resp. three out of five).

Let **ciphertext** be a string of K letters; **rotor** be the rotor wiring, whereby **rotor** $[j][x]$ is the output letter, on input x , of rotor j in its reference position; and **reflector** be the reflector wiring, whereby **reflector** $[x]$ is the output of the reflector on input x . In the following, we use N for the size of the alphabet and M for the number of rotor wheels (respectively 26 and 3 for Enigma). Moreover, we write $[a, b]$ for the discrete interval $\{a, a + 1, \dots, b\}$ and $[b]$ as a shortcut for $[0, b - 1]$. The model uses four sets of variables:

$$\begin{aligned} \text{plaintext} : \text{plaintext}_i &\in [N] & \forall i &\in [K] \\ \text{key} : \text{key}_j &\in [N] & \forall j &\in [M] \\ \text{ref} : \text{ref}_j &\in [N] & \forall j &\in [M - 1] \\ \text{plugboard} : \text{plug}[x] &\in [N] & \forall x &\in [N] \end{aligned}$$

The variables **plaintext** stand for the original message. The other variables stand for the settings of the machine used to cipher it: the variables **key** stand for the rotor position (rotor wheel j was advanced by key_j steps) the variables **ref** stand for the reference position of the two leftmost rotor wheels (rotor $j + 1$ advances one step when rotor j returns to position ref_j), and the variables **plugboard** stand for the plugboard connection ($\text{plug}[x]$ is the letter “steckered” to x by the plugboard). In a nutshell it ensures that an Enigma machine with rotors **rotor** that have been setup with reference position **ref**, in initial position **key** and with plugboard configuration **plugboard** ciphers **plaintext** to **ciphertext**. We use the auxiliary variables **x** with $x_{i,j} \in [N] \forall i \in [K], \forall j \in [2M + 2]$, with $x_{i,j}$ standing for the output of the scrambling device j for the letter at position i in the plaintext. The scrambling devices are ordered as explained in Section 3: plugboard first, then the three rotors, followed by the reflector, the three rotors backwards, and finally the plugboard again. We also use the auxiliary variables **p** with $p_{i,j} \in [K + K/N^j], \forall i \in [K], \forall j \in [M]$, to represent the total number of steps that the $j + 1$ -th rotor wheel has advanced when reading the $i + 1$ -th letter. Finally, the auxiliary variables **offset** with, for $j \in [2]$, offset_j represent the reduction in number of steps to advance for rotor j to reach its reference position ref_j for the first time.⁵

In the reminder of the section, we will extensively use the *Element* constraint [4]:

► **Definition 6 (Element).** Let $\mathbf{x} = \{x_1, \dots, x_K\}$ be a set of variables, and k, y be two variables. The constraint *Element* (\mathbf{x}, k, y) is the pair (S, R) where the relation R contains all assignments of $S = \mathbf{x} \cup \{k, y\}$ satisfying the predicate $x_k = y$.

When we write the expression “ x_k ”, with k a variable and $\mathbf{x} = x_1, \dots, x_n$ an array of variables, this should be read as an extra variable y constrained with *Element* (\mathbf{x}, k, y) . Similarly, for any arithmetic operator $\oplus \in \{+, -, *, /, \text{mod}\}$, the expression “ $x_1 \oplus x_2$ ” should be read as an extra variable y with the constraint defined by the predicate $x_1 \oplus x_2 = y$.

⁵ We include these variables for completeness, although they are often both set to the constant 0 (A).

4.1 Forward Rotor Model

If $\mathbf{rotor}[j][x]$ is the letter read after going through rotor j from input x , then after advancing k turns, the rotor produces the letter $(\mathbf{rotor}[j][(x + k) \bmod N] - k) \bmod N$ on the same input. The relation between the input letter $x_{i,j}$ and the output letter $x_{i,j+1}$ of the forward traversal of rotor j can therefore be encoded as follows:

$$p_{i,0} = \mathbf{key}_0 + i \quad \forall i \in [K] \quad (1)$$

$$\mathbf{offset}_j = \begin{cases} -\mathbf{ref}_j & \text{if } \mathbf{ref}_j \leq \mathbf{key}_j \\ N - \mathbf{ref}_j & \text{otherwise} \end{cases} \quad \forall j \in [M-1] \quad (2)$$

$$p_{i,j} = \mathbf{key}_j + \frac{p_{i,j-1} + \mathbf{offset}_{j-1}}{N} \quad \forall i \in [K], \forall j \in [1, M] \quad (3)$$

$$x_{i,j+1} = (\mathbf{rotor}[j][(x_{i,j} + p_{i,j}) \bmod N] - p_{i,j}) \bmod N \quad \forall i \in [K], \forall j \in [M] \quad (4)$$

Constraints (1, 2 and 3) channel the auxiliary variables \mathbf{p} , where $p_{i,j}$ stands for the number of times the rotor j has turned starting for position 0 when reading the i -th letter, to the initial positions \mathbf{key} of the rotors and to their reference position \mathbf{ref} via the variables \mathbf{offset} . Constraints (4) represent the substitution cipher used with input letter $x_{i,j}$ and output letter $x_{i,j+1}$: if rotor j advanced p steps, the wire that initially connected the letter α to the letter β now connects the letter $\alpha - p$ to the letter $\beta - p$ (modulo $N = 26$).

4.2 Reflector Model & Backward Rotor Model

The reflector is also a substitution cipher, but static (it does not change from a letter to the next), symmetric ($\forall x, \forall y \mathbf{reflector}[x] = y \iff \mathbf{reflector}[y] = x$) and antireflexive ($\forall x \mathbf{reflector}[x] \neq x$). Constraints 5 model the relation between the input $x_{i,M+1}$ of the reflector (the signal corresponding to the i -th letter after going through the all rotors forward) and its output $x_{i,M+2}$ with another *Element* constraint. Then, the signal travels through the rotors, but backward. Constraints 6 are similar as for the forward pass.

$$x_{i,M+1} = \mathbf{reflector}[x_{i,M}] \quad \forall i \in [K] \quad (5)$$

$$x_{i,M+j+1} = (\mathbf{rotor}[j][(x_{i,M+j+2} + p_{i,M-j}) \bmod N] - p_{i,M-j}) \bmod N \quad \forall i \in [K], \forall j \in [M] \quad (6)$$

4.3 Plugboard Model

Finally, we need to model the plugboard of the military version. Since it is composed of L plugs, each one connecting two letters, it is a symmetric permutation with $N - 2L$ *fixed points*, i.e., it leaves $N - 2L$ letters unchanged. Unlike the reflector (which is fully known) or the rotors (for which the only unknowns are their positions), the plugboard is not known for the attacker: its configuration is part of the cryptographic key to be computed during the attack. It is encoded as a vector $\mathbf{plugboard} = \langle \mathbf{plug}[1], \dots, \mathbf{plug}[N] \rangle$ of variables with domain $[N]$ where $\mathbf{plug}[x]$ stands for the letter mapped to letter x , and the following constraints:

$$\text{ALLDIFFERENT}(\mathbf{plugboard}) \quad (7)$$

$$(\text{plug}[x] = y) \Leftrightarrow (\text{plug}[y] = x) \quad \forall x, y \in [N] \quad (8)$$

$$\sum_{x=1}^N (\text{plug}[x] = x) = N - 2L \quad (9)$$

$$\text{plug}[\text{plaintext}_i] = x_{i,0} \quad \forall i \in [K] \quad (10)$$

$$\text{plug}[\text{ciphertext}[i]] = x_{i,2M+1} \quad \forall i \in [K] \quad (11)$$

Constraints (8), (9) and (9) ensure that the plugboard is in a legal configuration by stating that **plugboard** is a permutation (7); with $N - 2L$ identities (9)⁶; and is symmetric (8). Constraints (10) represent the transformation of the input by the plugboard, and Constraints (11) the final transformation, also by the plugboard, yielding the ciphertext as output.

4.4 Breaking the Code

So far, the model introduced in this section emulates the Enigma machine: a solution stands for a plaintext \mathbf{x} whose cipher with message key **key** and plugboard **plugboard** is the given ciphertext. However, there are too many consistent assignments of \mathbf{x} , **key** and **plugboard** to consider enumerating the solutions in order to find the original text.

In order to actually break the code, we use the same technique used by cryptanalyst at Bletchley Park in the 1940s. We suppose that we are somehow given a “crib”, that is, a portion of deciphered message.⁷ For instance, suppose that we know that the plaintext corresponding to the portion of ciphertext **IRSJYTCORS** starting at position s is in fact **CONSTRAINT**. Plaintext and ciphertext can therefore be aligned as shown in Table 1.

■ **Table 1** A crib: an alignment of plaintext and ciphertext.

s	$s + 1$	$s + 2$	$s + 3$	$s + 4$	$s + 5$	$s + 6$	$s + 7$	$s + 8$	$s + 9$
C	O	N	S	T	R	A	I	N	T
I	R	S	J	Y	T	C	O	R	S

Given a string of plaintext **T** and a string of ciphertext **C** such that $|\mathbf{T}| = |\mathbf{C}|$ starting at position s corresponding to a crib, we can find compatible initial rotor positions **key**, and plugboard configurations **plugboard**, by solving the model introduced in this section⁸ with $K = |\mathbf{C}|$, Constraints (1–11), as well as the equalities:

$$\text{plaintext}_i = \mathbf{T}[i] \quad \forall i \in [s, s + K] \quad (12)$$

This model may have several solutions, and only one corresponds to the actual cryptographic key. In practice, however, even small cribs can have a reasonable number of solutions, and verifying them manually can be done by deciphering the rest of the ciphertext with the same key. Experimental evaluations show that solving this model using standard CSP solvers is

⁶ The number of connections L went from 6 to 10 depending on Enigma’s versions. The equality “ $\text{plug}[x] = x$ ” is read as its natural conversion from Boolean to $\{0, 1\}$.

⁷ Cribs were obtained by guessing that a word or a sentence was likely to be in the message, and using the fact that a letter is never ciphered to itself to align that portion of plaintext with the ciphertext.

⁸ Notice that the definition of several constraints must take into account the positional offset s .

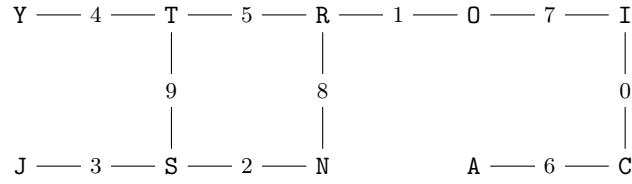
not trivial (see Section 7). In the two following sections, we review how the code was broken in practice and show how those ideas can be mapped to the notion of consistency. These observations directly lead to an efficient CP based method to break Enigma's code.

5 Breaking Enigma's Code: The British Method

The method developed by Alan Turing (and improved by Gordon Welchman) computes the rotor position and the plugboard configuration used to cipher the message. It ignores the reference positions, however, and actually fails if there was a turnover (i.e., if the middle rotor advanced when ciphering the crib). A very advanced machine for its time, called “Bombe” was built for that purpose at Bletchley Park. The Bombe was not only capable of quickly simulating several rotor positions in parallel, but it could also either rule out a message key, or (partially) compute a plugboard configuration consistent with that message key.

The method used by British cryptanalysts also relied on acquiring a crib, that is, a string \mathbf{T} of original text aligned to a string \mathbf{C} of same length in the ciphertext, as shown in Table 1. A *menu* for the Bombe was then extracted from the crib:

► **Definition 7 (Menu).** *Given a string of plaintext \mathbf{T} and a string of ciphertext \mathbf{C} such that $|\mathbf{T}| = |\mathbf{C}|$, the menu obtained from matching them is a graph with one vertex per letter in $\mathbf{T} \cup \mathbf{C}$, and an edge for each pair of matched letters $\mathbf{T}[i], \mathbf{C}[i]$ labelled by their position, i.e., $G = (\mathbf{T} \cup \mathbf{C}, \{(\{\mathbf{T}[i], \mathbf{C}[i]\}, i) \mid i \in [|\mathbf{T}|]\})$. We write $N_G(x) = \{(y, i) \mid (\{x, y\}, i) \in G\}$ for the neighborhood of letter x in the menu. Notice that it carries the edge labels.*

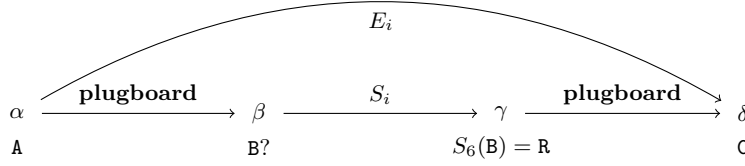


■ **Figure 3** Illustration of the Bombe's menu from the crib in Table 1.

Figure 3 illustrates the menu extracted from the crib of Example 1.⁹ An edge, say $(\{A, C\}, 6)$, of the menu indicates the letter A (resp. C) at position 6 is ciphered to C (resp. A). Now, observe that given some positions for the rotors, the Bombe can compute the scramble function (i.e., permutation of the alphabet) $S_i : [N] \mapsto [N]$ corresponding to the Enigma machine, ignoring the plugboard. It is then possible to make some inference on the configuration of the plugboard, as sketched in Figure 4. Indeed, suppose that the plugboard associates A to B. We can compute $S_i(B)$, and in particular $S_6(B)$, let it be R. Now thanks to the crib, we know that the cipher for A at position 6 is C, therefore the plugboard must associate R to C. This process can be repeated starting from any edge of the menu ending in R or C, yielding more deductions. Eventually, a contradiction might be found, or a fixed point might be reached. This is Turing and Welchman's inference rule, which can be formalised as follows:

► **Definition 8 (Plugboard configuration).** *Let \mathcal{P} be a collection of subsets of $[N]$ such that $|p| \leq 2 \forall p \in \mathcal{P}$. We say that \mathcal{P} is valid if and only if elements of \mathcal{P} are pairwise disjoint and contains no more than L pairs and no more than $N - 2L$ singletons. We say that \mathcal{P} is complete if and only if $\bigcup_{p \in \mathcal{P}} p = [N]$. The collection \mathcal{P} is valid and complete if and only if it corresponds to a legal plugboard configuration where, for every $\{\alpha, \beta\} \in \mathcal{P}$ the letters α and β are connected, and for every $\{\alpha\} \in \mathcal{P}$, the letter α is not changed by the plugboard.*

⁹ For the sake of the example, we let the first index s be 0



■ **Figure 4** Illustration of Enigma's encryption scheme: the input character (here α) is mapped to β by the plugboard; then scrambled to γ by going through the rotors forward, the reflector and the rotors backward; and finally mapped to δ by the plugboard.

► **Proposition 9** (Inference rule). *Let S be a scrambler (Enigma without the plugboard) and G a menu. If \mathcal{P} is the valid and complete plugboard configuration used during encryption of G , then, for every $\{\alpha, \beta\} \in \mathcal{P}$:*

- *for every $(\delta, i) \in N_G(\alpha)$, $\{\delta, S_i(\beta)\} \in \mathcal{P}$, and*
- *for every $(\delta, i) \in N_G(\beta)$, $\{\delta, S_i(\alpha)\} \in \mathcal{P}$.*

Proof. Suppose that the first rule does not hold, i.e., $\{\alpha, \beta\} \in \mathcal{P}$, $\exists(\delta, i) \in N_G(\alpha)$ such that $\{\delta, S_i(\beta)\} \notin \mathcal{P}$. The letter α is mapped to β by the plugboard and is scrambled to $S_i(\beta)$. However, since there is an edge $(\{\alpha, \delta\}, i)$ in the menu, we know that the encryption setup was such that the input letter α at position i yields letter δ . Therefore, the plugboard must match the letters $S_i(\beta)$ and δ . If there is no $p \in \mathcal{P}$ such that $\beta \in p$, then \mathcal{P} is not complete, and otherwise, it is not consistent, hence a contradiction.

The second rule follows from the same reason but starting from letter β . ◀

This inference rule can be used to discard a guessed plugboard connection. The Bombe is an electro-mechanical device that automatises this process. It is composed of several clones of the Enigma machine, each capable of emulating the encryption of all 26 letters in parallel for a given rotor setting, and wires for every possible plugboard connection. Given a message key **key**, and a menu, the Bombe was initialised by setting up one of the clones to emulate the scrambler S_i for every edge label i in the menu. Then a connection $\{\alpha, \beta\}$ could be “guessed” by sending a current flow through the corresponding wire into the Bombe. The inference rule described in Proposition 9 would then be applied as the current flowed through the input β of the clone S_i , for each $(\delta, i) \in N_G(\alpha)$. The current would in turn flow to the connection between $S_i(\beta)$ and δ and again to some clones of Enigma as indicated in the menu. This can either result in a fixed point where \mathcal{P} is closed under that rule, or yields an invalid plugboard configuration. In the latter case, the connection $\{\alpha, \beta\}$ can be ruled out, and another guess can be made. If there exists a letter for which no matching is possible, the message **key** is ruled out and the same process is repeated for another message key. Otherwise, **key** is a candidate key and a (partial) plugboard configuration is given by the connection in which the current flows.

► **Definition 10** (The Bombe method). *The Bombe made it possible, starting from a tentative connection $\{\alpha, \beta\}$, to enforce the inference rule described in Proposition 9 until either it fails (\mathcal{P} is no longer valid), or succeeds (it does not fail and reaches a fixed point). In the former case, the connection $\{\alpha, \beta\}$ can be ruled out.*

The Bombe method denotes the process where, for each rotor position, every possible plugboard connection involving a letter in the crib is ruled out if the process above fails. The Bombe method causes a stop if and only if it reaches a fixed point where every letter can appear in at least one connection. In that case, the current rotor position might be the correct one (the message key) and it is checked by other means. Otherwise, this key is ruled out and the process resume with another of the 26^3 positions.

► **Lemma 11.** *Let $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ be the CSP of Section 4. If the rule of Proposition 9 deduces the connection $\{\delta, \gamma\}$ from menu G and $\mathcal{P} = \{\{\alpha, \beta\}\}$ then enforcing AC on $(\mathcal{X}, \mathcal{D}|_{\text{plug}[\alpha] \leftarrow \beta}, \mathcal{C})$, reduces the domain of $\text{plug}[\delta]$ to $\{\gamma\}$ and of $\text{plug}[\gamma]$ to $\{\delta\}$.*

Proof. Suppose that the rule of Proposition 9 deduces the connection $\{\delta, \gamma\}$. We assume first that the first rule triggered and hence $(\delta, i) \in N_G(\alpha)$ and $S_i(\beta) = \gamma$.

Constraints (1 – 6) and (10 – 12) are all functional: when all but one of their variables are fixed, the last variable has a single consistent value.

Since $(\delta, i) \in N_G(\alpha)$, the letter α in the plaintext of the crib is matched to δ in the ciphertext at position i . By enforcing AC on Constraint 12, we have $D(\text{plaintext}_i) = \{\alpha\}$, and since $D(\text{plug}[\alpha]) = \{\beta\}$, by enforcing AC on Constraint (10) we have $D(x_{i,0}) = \{\beta\}$.

Moreover, since **key** is constant, so is **p**, by enforcing AC on Constraint (1) since only one non-constant variable remains. As a result, enforcing AC on Constraints (4) reduces the domains of variables $x_{i,1}, x_{i,2}$ and $x_{i,3}$ to single values. The same is true for Constraint (5) and Constraints (6). The variable $x_{i,2M+1}$ is therefore assigned, and it must be to value $S_i(\beta) = \gamma$. Enforcing AC on Constraint (11) sets the domain of variable $\text{plug}[\delta]$ to $\{\gamma\}$. Finally, Constraint (8) set the domain of $\text{plug}[\gamma]$ to $\{\delta\}$.

If it was the second rule that triggered, the same demonstration applies, although the chain of propagations to consider goes in the reverse direction: from $x_{j,2M+1}$ to $x_{j,0}$. ◀

The implication in Lemma 11 is not an equivalence simply because in some cases, Constraints 7, 8 and 9 might trigger and more connections could then be deduced via propagation. For instance if $L = 1$ and $\alpha \neq \beta$, then all variables in **plugboard** would be assigned by propagation of Constraint 9. The following theorem is also an implication for the same reasons, and in this case these constraints are even more likely to be relevant since the domains are reduced while enforcing SAC .

► **Theorem 12.** *If the Bombe method does not produce a stop, then enforcing singleton arc consistency on the model described in Section 4, with the variables **key** assigned, also fails.*

Proof. Let $\mathcal{CN} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be the CSP described in Section 4. In particular, **plugboard** $\subseteq \mathcal{X}$, however, **plaintext** and **key** are constant under the theorem's hypothesis.

Suppose that the Bombe method rules out a connection $\{\alpha, \beta\}$. It means that the rule of Proposition 9 produces an invalid plugboard configuration \mathcal{P} . However, from Lemma 11 we know that for every $\{\alpha, \beta\} \in \mathcal{P}$, after enforcing AC on the CSP $(\mathcal{X}, \mathcal{D}|_{\text{plug}[\alpha] \leftarrow \beta}, \mathcal{C})$, all but the value β (resp. α) are removed from $D(\text{plug}[\alpha])$ (resp. $D(\text{plug}[\beta])$). It entails that there exist two elements of \mathcal{P} that are not disjoint, e.g., $\{\alpha, \beta\} \in \mathcal{P}$ and $\{\alpha, \gamma\} \in \mathcal{P}$. In this case, the corresponding domain $D(\text{plug}[\alpha])$ is wiped out.

Therefore, every connection that is ruled out by the Bombe method is also ruled out by enforcing SAC . The Bombe method fails if all the connections involving a particular letter have been ruled out. If this happens for letter α , then the domain of the variable $\text{plug}[\alpha]$ is emptied, and therefore SAC fails. ◀

The Bombe can therefore be seen as a CSP solver that branches on the variables **key**, and enforces singleton arc consistency on leaves of this search tree. When SAC fails, the rotor position corresponding to that branch is ruled out, otherwise the machine *stops*. Then, the (partial) plugboard configuration is verified manually. If it cannot be extended to a configuration that correctly deciphers the message, then this rotor position is ruled out as well, and the Bombe resumes its search with a new assignment of **key**. It was therefore important to reduce the likelihood of such *false stops*.

6 Breaking Enigma's Code: The Polish Method

The Enigma code was actually first broken, before the war, by Polish mathematicians Marian Rejewski, Jerzy Różycki and Henryck Zygalski [5], who also successfully reconstructed, from partial and indirect intelligence, the Enigma machine and built mechanical devices to emulate it. Their insights later proved essential to the British effort. Turing and Welchman's method relied on a crib and would guess the rotor position and the plugboard configuration used to encrypt the message regardless of how it was chosen (in particular we shall see that this rotor position is different to the one given in the daily settings). The method of the Polish Cipher Bureau, on the other hand, relied on intercepting several messages sent with the same daily settings, and would compute not directly the message key, but the daily settings.

Using repeatedly the same encryption key would be a serious security flaw. Operators were thus instructed to randomly choose a 3-letter *message key*. This key (e.g., SAT) would be encrypted *twice*¹⁰ using the rotor position indicated in the daily settings (e.g., the text SATSAT would be ciphered using the key CSP to some 6-letter ciphertext). Then, the machine would be reset to position SAT and the message would be ciphered using that message key, and both ciphers (6-letter prefix and text) sent in the same message. The receiver would then set its own Enigma machine to position CSP, decipher the prefix, reset its machine to the obtained position SAT, and finally decrypt the message. This method still had the weakness of using the same key (here, CSP) to encrypt all messages keys for a given day. Rejewski devised a first method, involving a dedicated machine, the *cyclometer*, that partially automated the design of a lookup table (the *cards catalog*) from which the daily rotor positions could easily be found, provided several prefixes encrypted with the same daily settings [1].

The procedure was therefore upgraded: the sender chose a *plaintext key* besides the message key. Then, the rotors were positioned according to the daily rotor settings *shifted by the plaintext key*. In other words, the actual encryption key was equal to *rotor settings + plaintext key*, where “+” stands for the modular, component-wise addition. This key was used to encrypt the message key and it was sent in plaintext with the encrypted message. For instance, if the rotor setting is CSP, the chosen plaintext key MIP, then the rotors would be put in position OAE = CSP + MIP to cipher the chosen message key (say SAT). The text SATSAT would yield, for instance, DGFAGX. Then the rotors were reset to position SAT and the actual message was ciphered to “**ciphertext**”. Finally, the message sent would be:

MIP DGFAGX ciphertext

The receiver would add the plaintext key to the daily setting to recover the actual rotor position, then, as previously, decipher the message key, reset the machine to the corresponding rotor position and decipher the message. The difference with the previous practice, however, was that the rotor position used to cipher the 6-letter prefix was never twice the same. The fact that the message key was repeated twice, however, proved nonetheless to be a fatal flaw.

When rotors and plugboard are in a given configuration, Enigma corresponds to a symmetric permutation, although the permutation changes with every letter since the rotors advance. Let E_i^{XYZ} stand for the permutation applied by Enigma to the i -th letter with initial rotor position XYZ.¹¹ Consider the prefix MIP DGFAGX and let $\mathbf{key} = \{key_1, key_2, key_3\}$ denote the unknown rotor setting common to all messages of a given day. We know that a

¹⁰ This practice was abandoned May 1st 1940, hence making the Polish attack irrelevant.

¹¹ The plugboard is ignored here.

letter (say x) is mapped to the letter D in $E_1^{\text{key}+\text{MIP}}$ and to A in $E_4^{\text{key}+\text{MIP}}$. Moreover, since $E_1^{\text{key}+\text{MIP}}$ is symmetric, it maps A to x . Therefore, the composition $E_1^{\text{key}+\text{MIP}} E_4^{\text{key}+\text{MIP}}$ maps D to A and vice versa. Because the plugboard is unknown we do not learn anything from that.

However, observe that $E_2^{\text{key}+\text{MIP}} E_5^{\text{key}+\text{MIP}}$ transforms the letter G to itself. Mathematician Henryk Zygalski noticed that this was relevant because the plugboard changes the letter but conserve the fixed points of permutation $E_2^{\text{key}+\text{MIP}} E_5^{\text{key}+\text{MIP}}$ (an element unchanged by the permutation), and because not all rotor settings have such fixed points [5].

A device, “the Bomba”,¹² was designed and built by the Polish Cipher Bureau to go over all of the 26^3 rotor settings, and record which settings could allow such a fixed point in one of the three composed permutations. Only 40% of the rotor positions had a composition fixed point. A set of 26 perforated sheets (known as the *Zygalski sheets*) were to be produced for each of the 6 possible rotor orders.¹³ Each sheet corresponds to a letter, standing for the first letter of the unknown rotor position, and contains a 26×26 Boolean matrix standing for whether the second and third letters could extend the first letter to a position allowing a fixed point. We denote $Z[\alpha, \beta, \gamma]$ the fact that there is a hole at positions β, γ in the Zygalski sheet for letter α , which is true if and only if the rotor position $\alpha\beta\gamma$ has a composition fixed point (and hence may encode the two occurrences of a letter in the prefix to the same code).

Consider now a prefix MIP VNEVSX. There is a fixed point at position (1, 4), hence the Zygalski sheets allowed some inference: one would first guess the order of the rotors and a first letter key_1 indicating the position of the first rotor. Then, let $\alpha = \text{key}_1 + \text{M}$. The α -th sheet would be taken from the box standing for the chosen order. The matrix on that sheet (shifted by I and P in the respective dimensions¹⁴) thus stands for the possible values of key_2 and key_3 . This narrows down the number of possibilities by 60%. Now, suppose that a message with prefix: SMT DGFAGX is intercepted the same day. This prefix also has a fixed point, but at position (2, 5). Therefore, the same reasoning applies, using sheet $\text{key}_1 + \text{S} + 1$ in the same box, but shifted by M and T. The “+1” models that the fixed point is on the 2nd and 5th letters, which is equivalent to observing it on the 1st and 4th letters, however with the position of the left rotor advanced one step. The subset of holes that allow both fixed points is obtained by shining light through the two sheets, properly shifted and aligned. Usually, a dozen messages including a fixed point (and as many sheets) were necessary to narrow down the number of possibilities to either 0, in which case the guess was proven wrong; or 1, in which case the rotor position **key** and the rotor order could be easily retrieved.

This method does not take the plugboard into consideration. However, since 14 letters were not affected by the plugboard¹⁵, it is possible to reconstruct the message manually. It also fails in case a turnover of the middle rotor happens before the end of the 6-letter prefix.

Aligning several Zygalski sheets corresponds to solving the following CSP:

► **Definition 13** (Zygalski’s CSP). *We call Zygalski’s CSP the constraint satisfaction problem with set of variables $\text{key} = \{\text{key}_1, \text{key}_2, \text{key}_3\}$, and for each message with plaintext key $\alpha\beta\gamma$ containing a fixed point at positions $(1 + i, 4 + i)$, a constraint given by the predicate $Z[\text{key}_1 + \alpha + i, \text{key}_2 + \beta, \text{key}_3 + \gamma]$.*

► **Theorem 14.** *The letter α for the position of the first rotor is refuted by the Zygalski sheets method if and only if no pair of instantiations $\{\text{key}_2 \leftarrow \beta, \text{key}_3 \leftarrow \gamma\}$ has $\text{key}_1 \leftarrow \alpha$ as path support in Zygalski’s CSP.*

¹² An aggregate of six Enigma machines, one for each permutations of the prefix.

¹³ The process was long and difficult, even with the Bomba: only 2 sets had been completed when the invasion of Poland began, but the sheets were eventually finalized at Blechley Park.

¹⁴ The sheets had 25 repeated rows and columns to allow for shifting modulo 26.

¹⁵ Only 6 plugs were used when the Polish began to attack Enigma.

Proof. By definition, a hole with coordinates β, γ will let the light shine through all sheets, if and only if $\{key_1 \leftarrow \alpha, key_2 \leftarrow \beta, key_3 \leftarrow \gamma\}$ is consistent with every constraint of Zygal's CSP, that is, if and only if $key_1 \leftarrow \alpha$ is a path support of $\{key_2 \leftarrow \beta, key_3 \leftarrow \gamma\}$. ◀

The process of superimposing several Zygal's sheets corresponding to a guess of the letter α for the position of the first rotor is an efficient method to compute the 2-tuples which have $key_1 \leftarrow \alpha$ as path consistent support. Moreover, since there are only three variables in the network, if $key_1 \leftarrow \alpha$ is a path support of $\{key_2 \leftarrow \beta, key_3 \leftarrow \gamma\}$ then $key_2 \leftarrow \beta$ is a path support of $\{key_1 \leftarrow \alpha, key_3 \leftarrow \gamma\}$ and $key_3 \leftarrow \gamma$ is a path support of $\{key_1 \leftarrow \alpha, key_2 \leftarrow \beta\}$. Therefore path consistency can be achieved by only checking the values of a single variable in this way, there is a solution if and only there is a tuple with a path support.

Path consistency is usually only applied to binary constraints. However, the definition we use naturally extends this consistency to non-binary constraints and in that context, the analogy stands. For instance, consider two Zygal's sheets: one allowing the keys ABC, ADE, BBE and BDC and one allowing the keys ABE, ADC, BBC and BDE. There is no solution, and indeed the Zygal's CSP is not *PC* (e.g., the consistent and valid assignment $\{key_1 \leftarrow A, key_2 \leftarrow B\}$ cannot be extended to the third variable), yet the CSP is *AC* and *SAC*.

7 Experimental Evaluation

We ran experiments to verify the impact of *SAC* on this problem. The constraint model was implemented using the toolkit Choco [8].¹⁶ To emulate the Bombe, we force the heuristic to select the variables standing for the positions of the rotors (**key**) before other variables. In the version denoted **Choco+SAC**, we run *SAC* only once these variables are all assigned. When *SAC* does not fail, which corresponds to a stop of the Bombe, or in the default version denoted **Choco**, we let the constraint solver either find a solution for the variables **plugboard**, or prove that no solution exists for the current rotor position. For both methods we treated every variable in **ref** as the constant 0 as was done in the methods we discussed previously. Not doing so would increase the search space, and the number of solutions, by a factor 26^2 .

In order to generate benchmark instances, we used many cribs of length 12 from wikipedia texts that we ciphered using a random position of the rotors I, II and III of the first Enigma machine that was introduced in 1930 [9], and a random reflector.

Fast:	E K M F L G D Q V Z N T O W Y H X U S P A I B R C J
Middle:	A J D K S I R U X B L H W T M C Q G Z N P Y F V O E
Slow:	B D F H J L C P R T X V Z N Y E I W G A K M U S Q O
Reflector:	P R Y Z L X O S Q K J E N M G A I B H W V U T F C D

We selected 260 cribs, so that we uniformly cover a range of values for the following parameters: *size of the menu* (size), i.e., the number of vertices in the graph of the extracted menu and *number of cycles* (#cycle). The idea is that sparse or acyclic menus produce more stops. In particular Turing made an analysis of how many stops would occur according to the values of these parameters [2]. Therefore, higher number of cycles and lower menu size should make the problem easier. Moreover, we ignored menus with four or more connected components. We average the results on instances with same size and number of cycles.

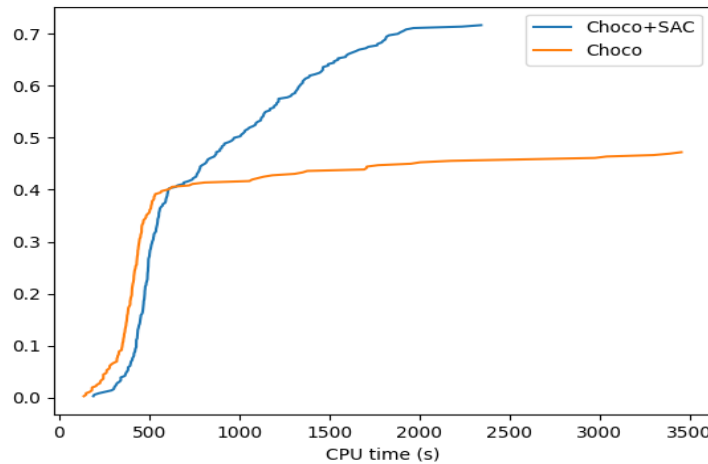
¹⁶The source code is available here: <https://gitlab.laas.fr/vantuori/sacnigma.git>.

■ **Table 2** Results of **Choco** and **Choco+SAC** on a range of cribs.

Crib					Choco			Choco+SAC		
#inst.	#cycle	size	#stops	#sol.	#solved	CPU	#fail	#solved	CPU	#fail
10	0	12	128.4	17912	3	378	388790.7	10	1197	17448.6
10	0	13	242.8	12214	7	901	1036519.1	10	922	17334.3
10	0	14	5.6	593	7	989	968841.6	10	866	17570.4
10	0	15	2.0	26	8	438	414718.2	10	607	17574.0
10	0	16	1.0	14	8	1208	1251729.9	10	813	17575.0
10	1	11	68.6	124111	5	475	542790.6	10	908	17507.5
10	1	12	156.2	33912	6	958	1435607.2	10	751	17421.3
10	1	13	1.8	1182	5	604	693048.2	10	1061	17574.2
10	1	14	1.9	166	8	606	580758.1	10	748	17574.1
10	1	15	1.0	77	8	369	350508.9	10	560	17575.0
10	1	16	1.0	40	8	1057	1239245.6	10	832	17575.0
10	2	10	18.2	165837	4	604	585009.0	9	1048	17557.8
10	2	11	63.0	45071	7	637	958325.9	10	796	17515.3
10	2	12	1.2	2141	7	476	544253.6	10	734	17574.8
10	2	13	2.0	4002	4	391	378688.0	10	793	17574.0
10	2	14	1.0	85	9	391	376638.7	10	578	17575.0
10	2	15	1.0	39	7	715	798622.9	10	755	17575.0
10	2	16	1.0	32	9	352	349036.6	10	618	17575.0
10	3	9	7.6	240639	6	437	338330.8	9	907	17573.2
10	3	10	27.7	192315	2	451	659440.0	10	1267	17551.4
10	3	11	1.1	10001	5	719	987604.6	10	959	17574.9
10	3	12	2.1	19165	4	560	679398.8	10	916	17573.9
10	3	13	1.0	341	7	341	331790.3	10	696	17575.0
10	3	14	1.0	127	9	390	359362.0	10	534	17575.0
10	3	15	1.0	486	8	358	342439.2	10	528	17575.0
10	3	16	1.0	31	9	524	513744.6	10	612	17575.0

All experiments were run on 4 cluster nodes, with Intel Xeon CPU E5-2695 v4 2.10GHz cores running Linux Ubuntu 16.04.4. We ran both models for one hour or until completion on every instance. We report in Table 2 the characteristics of the cribs, the average number of solutions ($\#sol.$), and the average number of solutions with distinct rotor positions, that is, the number of “stops” ($\#stops$). Then, for both methods, we report the number of instances solved ($\#solved$), that is, where all solutions have been enumerated within the one hour cutoff, the average CPU time (in seconds) over solved instances (CPU) and the average number of fails during search ($\#fail$). We can observe that using *SAC* significantly improves the model: **Choco** solves about 69% of the instances in less than one hour, whereas **Choco+SAC** solves 99% of the instances in about 15 minutes in average. The number of fails of **Choco+SAC** shows clearly that, as expected, constraint propagation does not actually cut the search tree for the rotor positions. All of the 17576 possible keys are explored. However, it also shows that in the few cases where *SAC* does not fail (the stops), virtually every singleton arc consistent permutation of the plugboard is consistent with the crib. Indeed the solver does not fail, and the number of solutions may become relatively large in that case. On the other hand, the number of fails of **Choco** shows a more conventional picture where plugboard configurations are ruled out by a blend of propagation and search.

The number of stops is lower than we would expect of the Bombe by about one or two orders of magnitude for low number of cycles. This is because we count only rotor positions for which a consistent plugboard configuration exists, whereas the Bombe stops as soon as a



■ **Figure 5** Cumulative probability to break the code in less than X seconds.

(slightly weaker form of) *SAC* can be enforced. Moreover, enforcing *SAC* on the constraint model takes advantage of propagation of the constraints modeling the plugboard (e.g., the *ALLDIFFERENT* constraint). Finally, the Bombe only takes into account the letters of the menu, i.e., it does not check that these extra letters too must have a legal plug connection.

Interestingly, we observe that the number of solutions tends to be larger for denser and more cyclic menus, even though the number of consistent rotor positions decreases, as we would expect from the Bombe. It shows that there are many more consistent plugboard configurations in this case. Overall, those parameters have a lower impact on the constraint model as they seem to have had on the Bombe. Overall, the method **Choco+SAC** is not extremely fast, but it is very robust. The graph in Figure 5 shows that in the most favorable cases, not enforcing singleton arc consistency can be the most efficient approach.

8 Conclusion

We have shown that the method designed by Alan Turing and Gordon Welchman at Bletchley Park to break the Enigma code has uncanny similarities with applying Singleton Arc Consistency on a constraint satisfaction problem modeling the machine. Experiments show that indeed, Singleton Arc Consistency significantly reduces the computation time required to decipher a message. Moreover, the method designed by Marian Rejewski et al. before that can also be related to achieving Path Consistency on another constraint satisfaction problem.

References

- 1 Chris Christensen. Polish Mathematicians Finding Patterns in Enigma Messages. *Mathematics Magazine*, 80(4):247–273, 2007. URL: <http://www.jstor.org/stable/27643040>.
- 2 Cipher A. Deavours and Louis Kruh. The Turing Bombe: Was it Enough? *Cryptologia*, 14(4):331–349, 1990.
- 3 Romuald Debruyne and Christian Bessiere. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI)*, 1997.

- 4 Pascal Van Hentenryck and Jean-Philippe Carillon. Generality versus Specificity: An Experience with AI and OR Techniques. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI)*, 1988.
- 5 Wladyslaw Kozaczuk. *Enigma. How the German Machine Cipher Was Broken, and how It Was Read by the Allies in World War II*. University Publications of America, 1984.
- 6 Alan K. Mackworth. Consistency in Networks of Relations. In Bonnie Lynn Webber and Nils J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 69–78. Morgan Kaufmann, 1981. doi:10.1016/B978-0-934613-03-3.50009-X.
- 7 Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974. doi:10.1016/0020-0255(74)90008-5.
- 8 Charles Prud’homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Solver Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. URL: <http://www.choco-solver.org>.
- 9 Wikipedia contributors. Enigma rotor details, 2021. URL: https://en.wikipedia.org/wiki/Enigma_rotor_details.

Combining Monte Carlo Tree Search and Depth First Search Methods for a Car Manufacturing Workshop Scheduling Problem

Valentin Antuori ✉

Renault, Plessis-Robinson, France

LAAS-CNRS, Université de Toulouse, CNRS, France

Emmanuel Hebrard ✉ 

LAAS-CNRS, Université de Toulouse, CNRS, ANITI, France

Marie-José Huguet ✉

LAAS-CNRS, Université de Toulouse, CNRS, INSA, France

Siham Essodaigui ✉

Renault, Plessis-Robinson, France

Alain Nguyen ✉

Renault, Plessis-Robinson, France

Abstract

Many state-of-the-art methods for combinatorial games rely on Monte Carlo Tree Search (MCTS) method, coupled with machine learning techniques, and these techniques have also recently been applied to combinatorial optimization. In this paper, we propose an efficient approach to a Travelling Salesman Problem with time windows and capacity constraints from the automotive industry. This approach combines the principles of MCTS to balance exploration and exploitation of the search space and a backtracking method to explore promising branches, and to collect relevant information on visited subtrees. This is done simply by replacing the Monte-Carlo rollouts by budget-limited runs of a DFS method. Moreover, the evaluation of the promise of a node in the Monte-Carlo search tree is key, and is a major difference with the case of games. For that purpose, we propose to evaluate a node using the marginal increase of a lower bound of the objective function, weighted with an exponential decay on the depth, in previous simulations. Finally, since the number of Monte-Carlo rollouts and hence the confidence on the evaluation is higher towards the root of the search tree, we propose to adjust the balance exploration/exploitation to the length of the branch. Our experiments show that this method clearly outperforms the best known approaches for this problem.

2012 ACM Subject Classification Mathematics of computing → Combinatoric problems; Mathematics of computing → Combinatorial optimization; Computing methodologies → Planning and scheduling; Computing methodologies → Discrete space search

Keywords and phrases Monte-Carlo Tree Search, Travelling Salesman Problem, Scheduling

Digital Object Identifier 10.4230/LIPIcs.CP.2021.14

Supplementary Material *Software (Source Code)*: <https://gitlab.laas.fr/vantuori/mcts-cp>

1 Introduction

The assembly floor of our car manufacturer partner contains several machines, each producing a certain type of components and as many machines consuming those components. The process of moving components across the workshop, from the point where they are produced to the point where they are consumed is a major bottleneck for the production rate of the plant. The resulting transportation problem can be seen as a *repetitive single vehicle pickup and delivery problem with time windows and capacity constraint*. The repetitive aspect comes from the fact that over a weekly schedule, the pickups and deliveries between the same



© Valentin Antuori, Emmanuel Hebrard, Marie-José Huguet, Siham Essodaigui, and Alain Nguyen; licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 14; pp. 14:1–14:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

pairs of machines is repeated at a given frequency, and for the same reason, both tasks are constrained in time. Finally, the capacity comes from the specific trolleys used by operators, which can be stacked in trains of a bounded length.

The method used in the industrial context is a large scale scheduling model solved using local search solver. A range of approaches relying on reinforcement learning (RL) were recently proposed in [2]. A simple stochastic branching policy (a linear model over some problem-specific parameters) is learned via RL, and used either to guide a constraint programming approach with rapid restarts, a constraint approach with limited discrepancy search, or a multistart local search method. All three methods vastly outperform the industrial method both on real and synthetic data sets.

In this paper, we introduce a new approach, combining Monte-Carlo Tree Search (MCTS) with budget-limited Depth First Search (DFS). MCTS was initially designed for solving AI games [6], and, over the last few years, MCTS combined with reinforcement learning and deep learning has enabled a breakthrough in the resolution of many combinatorial games (such as Go, with AlphaGo and AlphaGo Zero[22, 23]). Monte-Carlo Tree Search [6] offers a good generic strategy to tackle combinatorial problems. The expected outcome of a subtree is evaluated via Monte-Carlo simulation: starting from an open node of the search tree, a complete solution is built using a randomized heuristic policy. The outcome of the rollout is back-propagated to that node and all its ancestors down to the root by computing an average. Then, the next node to expand is selected by traversing the search tree from the root using multi-armed bandits algorithms (e.g., Upper Confidence bounds applied to Trees, UCT [11]) until reaching a node that has not been expanded yet. Without requiring built-in domain knowledge, Monte-Carlo rollouts provide good guidance, and the expansion phase gives guarantees on the compromise between exploration and exploitation. We show that in our problem, replacing the Monte-Carlo rollouts by randomized, limited-budget DFS is effective.

Several hybridizations with combinatorial optimization frameworks have been proposed. MCTS has been combined with constraint programming (CP) in [13], where the simulation phase stops at first fail, and the authors do not allow backtracking. Moreover, in order to allow restarts, instead of keeping an evaluation of every open node, this is done on pairs variable/value, in a way inspired by the RAVE (Rapid Action Value Estimation) heuristic used in Go [8]. Finally, took advantage of the fact that Gecode [7] uses copying instead of trailing, to open every search node visited during a rollout. In the field of Boolean satisfiability (SAT), MCTS has been combined with SAT solver [17], however, in this case without including the defining characteristics (clause learning, VSIDS, ect.) of modern SAT solvers. In [9], the authors propose to hybridize MCTS with local search to solve the MAX-SAT problem. They use a fixed limited-budget stochastic local search in place of the rollouts. Finally, in [21], the UCT algorithm has been used in mixed integer linear programming (MILP), although replacing Monte-Carlo rollouts by a lower bound obtains with the Linear Programming (LP) relaxation.

We are not aware of MCTS approaches using DFS rollouts. However, this is closely related to the Hybrid Best First Search (HBFS) algorithm introduced in [1] where limited DFS is interleaved with BFS, although the choice of leaf to expand is not based on the same principles. Besides using DFS, we propose two adaptations of MCTS method designed to be effective on our problem, but directly applicable in any combinatorial problem.

First, since the goal of a Monte-Carlo rollout is to evaluate a single decision, and since each subsequent heuristic decision reduces the relative impact of that first decision, we argue that the definition of the overall reward should reflect this form of “diminishing returns”.

Therefore, we propose to define the outcome of a rollout as the sum of the marginal increments of the lower bound at each step, weighted by a coefficient in $]0, 1[$ that exponentially decreases with the depth. When the coefficient tends towards 1, the outcome tends towards the overall objective value of the solution, and when it tends towards 0, the short term growth of the lower bound weigh more and more. Observe that this scheme is generic, it only requires a lower bound of the objective function which is monotonically non decreasing at each decision.

Second, in the multi-armed bandit algorithm, the tradeoff between exploration and exploitation is controled by a constant factor c for the exploration term. As the tree becomes deeper, the number of iterations of the multi-armed bandit along a branch grows. Therefore, the probability that it will deviate from the best branch so far grows exponentially with the depth of the branch. To offset this, we apply an exponential decay to the parameter c towards the root, so that the likelihood of deviating at the root decreases rapidly when the depth of the tree grows.

The paper is organized as follows. First, in Section 2 we describe the problem of routing vehicle components in car manufacturing workshops and we give a detailed overview of the standard MCTS algorithm in Section 3. Then, we present the novel aspects of our approach in Section 4. Finally, we give the specific implementation details for the considered problem in a MCTS framework in Section 5, and we report the results of extensive experiments on both industrial and synthetic data in Section 6. These experiments show that our adaptations of the MCTS method significantly outperforms previous methods, including the local search approach currently used in the industry.

2 Problem Description

The industrial assembly line consists of a set of m components to be moved across a workshop, from the point where they are produced to where they are consumed. Each component is produced and consumed by two unique machines, and it is carried from one to the other using four dedicated trolleys. Initially, there are two trolleys standing at the production point and two trolleys at the consumption point. On each side, one of them is full and the other is empty. However, the empty trolley at the production point is being filled, and the full trolley at the consumption point is being emptied. The full trolley at the production point must be brought to the consumption point before the initially full trolley there has been emptied, and symmetrically, the empty trolley at the consumption point must be brought to the production point before the initially empty trolley there has been filled. A production cycle is the time c_i taken to produce (resp. consume) component i , that is, to fill (resp. empty) a trolley. The two pickups and the two deliveries (of empty and full trolleys) described above must then be done within this time window. The end of a production cycle marks the start of the next, hence there are $n_i = \left\lfloor \frac{H}{c_i} \right\rfloor$ cycles over a time horizon H for the component i .

The problem is illustrated on a small example in Figure 1. In this example, there are 3 components having their own production and consumption machines, denoted by P_i and C_i in Figure 1(a). The lines between the machines represent the routes in the assembly line. The time cycles of each component and the time horizon (H) are given in Figure 1(b). In this example, there are 3 time cycles for the yellow component, 4 time cycles for the red component and 2 for the blue one.

For each component i , for each of its cycles k , there are two pickups and two deliveries: the pickup pe_i^k and delivery de_i^k of the empty trolley from the consumption area to the production one, and the pickup pf_i^k and delivery df_i^k of the full trolley from production to consumption. The processing time of an operation o is denoted pt_o and the travel time between operations o and o' is denoted $tt_{o,o'}$.

Let O be a set of all pickup and delivery operations with $|O| = n$. The problem is to compute a sequence $\omega : \{1, \dots, n\} \mapsto O$ of the operations O , where $\omega(j)$ is the j -th operation in the sequence, and $\chi = \omega^{-1}$ its inverse. The sequence ω must satisfy the following constraints:

Routing: For every $1 < j \leq n$, operation $\omega(j)$ must be given a start time $s_{\omega(j)}$ (and end time $e_{\omega(j)} = s_{\omega(j)} + p_{\omega(j)}$) taking into account duration and travel time: $s_{\omega(j)} \geq s_{\omega(j-1)} + p_{\omega(j-1)} + tt_{\omega(j-1), \omega(j)}$ (and $s_{\omega(1)} = 0$).

Time windows: An operation o occurring at period k for component i is given a release date $r_o = (k-1)c_i$ and a due date $d_o = kc_i$, with $r_o \leq s_o$ and $e_o \leq d_o$.

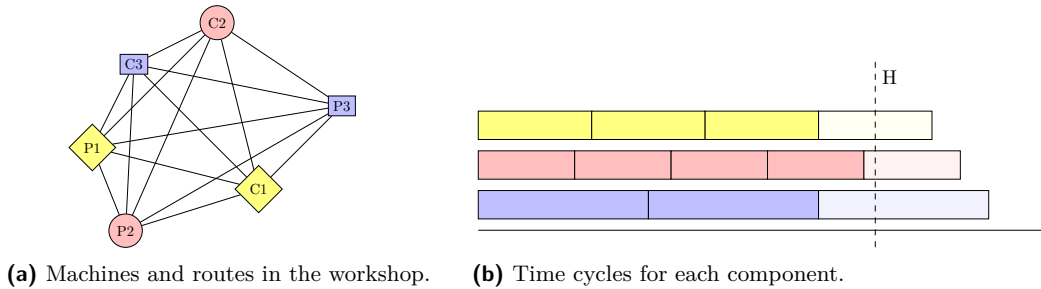
Precedences: Pickups must precede their deliveries in the same period.

$$\chi(pf_i^k) < \chi(df_i^k) \wedge \chi(pe_i^k) < \chi(de_i^k) \quad \forall i \in [1, m] \quad \forall k \in [1, n_i] \quad (1)$$

Train length: The operator may assemble trolleys into a train (trolleys can be extracted out of the train in any order), so a pickup need not be directly followed by its delivery. However, the total length of the train of trolleys must never exceed a length T_{\max} .

Notice that there are only two possible orderings for the four operations of a production cycle. Indeed, since the first delivery (which can be either the full or the empty trolley since they happen in parallel) and the second pickup take place at the same location, doing the second pickup before the first delivery is dominated: the train will needlessly contain both a full and an empty trolley for the same component, and this delivery will need to be done eventually and can only incur further time loss.

This industrial problem is a *repetitive single vehicle pickup and delivery problem with time windows and capacity constraint*. In this problem, the production-consumption cycles of each component entail a very particular structure: the four operations of each component must take place in the same time windows and all of these operations are repeated for every cycle. In addition, all operations are mandatory and there is no objective function for the industrial application, instead, feasibility is hard. As a result, the efficiency of the Large Neighborhood Search approaches proposed in [19] for such routing problems, are severely hampered since they rely on the length of the tour as the objective to evaluate the moves and the insertion of relaxed requests is often very constrained by the specific precedence structure. This problem was previously introduced in [2], and both exact and heuristic methods were proposed to solve it. These approaches rely on a fine tuned heuristic, and it was observed for some instances that greedy dives of the solvers were able to find a solution. The main motivation for a MCTS approach comes from this observation as the algorithm strongly rely on greedy dives, and is entirely guided by them.



■ **Figure 1** Illustrative example.

3 The Monte-Carlo Tree Search Method

In this section, we give some overview of the Monte-Carlo Tree Search method, and we introduce notations that will be used in the following.

MCTS is a tree search heuristic method based on multi-armed bandit principles to guide the tree expansion and to ensure a compromise between exploration and exploitation. This method was widely studied in the context of games but also for solving optimization problems [21, 20, 16, 14, 15, 5]. For a detailed survey on the MCTS method, the reader may refer to [4]. In a nutshell, the MCTS method develops a search tree where a node corresponds to a state of a given problem, with final states being solutions. Each node is associated with a set of feasible actions leading to child nodes in the tree. The aim is to find a path from the root node to a final state maximizing a reward. The MCTS method is based on four principles:

1. a reward can be computed at each final state;
2. a simulation process, also called rollout, is used to produce a path from a given node to a final state (for instance based on random sampling);
3. a backpropagation method to update node information after each new rollouts;
4. a selection mechanism, usually based on multi-armed bandit [12], for guiding the tree expansion and insuring a compromise between exploitation (select the most promising node) and exploration (visit different parts of the tree).

Let \mathcal{A} be a set of actions. A state $\sigma \in \mathcal{A}^*$ is a sequence of actions, and $|\sigma|$ denotes its length. We note $\sigma|a$ the state reached when applying action a in state σ , $\mathcal{A}(\sigma)$ denote the set of possible actions in state σ , and $p(\sigma)$ the parent state of σ . The MCTS method stores in memory the tree \mathcal{T} it has already explored, and for every state σ , it stores the triplet: $\langle N(\sigma), Pr(\sigma), V(\sigma) \rangle$, where $N(\sigma)$ is the number of time (σ) has been visited, $Pr(\sigma)$ is the prior probability or prior preferences to choose the state σ from its parent state $p(\sigma)$, and $V(\sigma)$ is the expected value of subtrees rooted at σ , and computed by averaging the outcomes of Monte-Carlo rollouts. Notice that $Pr(\sigma)$ was introduced in the MCTS in [22] but was not in the original form of MCTS.

The algorithm iterates over the four following phases until some stopping criteria are met.

Selection

The *selection* phase begins at the root node of \mathcal{T} , and finishes when we reach a node that has not yet been explored. At each node $\sigma \in \mathcal{T}$, an action is selected according to the statistics stored in σ :

$$a^* = \arg \max_{a \in \mathcal{A}(\sigma)} \tilde{V}(\sigma|a) + c * U(\sigma|a) \quad (2)$$

where $\tilde{V}(\sigma|a)$ is the exploitation term (based on the value of node $V(\sigma|a)$), $U(\sigma|a)$ is the exploration term, and c is a parameter which represents the balance between the two terms. This process continues from the state $\sigma|a^*$ until a non-visited node is reached, i.e. a leaf of the subtree \mathcal{T} .

In adversarial games, the value V of a node is the expected outcome, e.g., 1 for a win and 0 or -1 for a loss. In the context of combinatorial optimisation, however, several definitions have been used. A first possibility is to simply store the expected objective value, although this technique entails that rollouts must be complete, even when they are suboptimal early on. In [16] and [14], the authors consider a solution whose objective value is within a factor α of the best known solution as a “win” (the effective value is in $[0, 1]$ depending on the quality of solution) and all other outcomes as loss (0). The parameter α must therefore be carefully chosen, and the likelihood of a positive reward decreases when the best known

solution improves. In [13], the MCTS is frequently restarted (and hence the MCTS tree lost), then the authors store the outcomes of the rollouts on variable/value pair instead. In this technique the rollouts are depth first search calls stopped on the first fail, and the expected relative failure depth is stored for each variable/value pair instantiated in the selection phase. Finally, in [21], instead of a rollout, the lower bound of the LP relaxation is backpropagated instead.

Observe that it is important to normalize the value V stored on the node, to make the choice of the balance exploitation/exploration parameter c more robust. A state value $\sigma|a$ ending on the action a can be normalized in $[-1, 1]$ as follows:

$$\tilde{V}(\sigma|a) = \begin{cases} 2 * \frac{V^+ - V(\sigma|a)}{V^+ - V^-} - 1 & \text{if } N(\sigma|a) > 0 \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

Where $V^+ = \max\{V(\sigma|a) \mid a \in \mathcal{A}(\sigma), N(\sigma|a) > 0\}$ and $V^- = \min\{V(\sigma|a) \mid a \in \mathcal{A}(\sigma), N(\sigma|a) > 0\}$ are, respectively, the maximum and minimum values of any explored sibling state.

Finally, the exploration term is [22]:

$$U(\sigma) = Pr(\sigma) \frac{\sqrt{N(p(\sigma))}}{N(\sigma) + 1} \quad (4)$$

The rationale is to select the action a that maximizes $\tilde{V}(\sigma|a)$ plus a bonus that decreases with each visit in order to promote exploration. The prior probability $Pr(\sigma)$ biases the initial exploration by the knowledge we have on the state. The square root term could be replaced by a logarithmic term which is often used in MCTS, without changing this rationale.

Expansion

Let σ be the node returned by the selection procedure, during the *expansion* phase, for all $a \in \mathcal{A}(\sigma)$, a child $\sigma|a$ is added to σ and initialized its visit counter $N(\sigma|a)$ with its expected objective value $V(\sigma|a)$ set to 0. If no prior probability $Pr(\sigma|a)$ is available for this state, the uniform distribution $1/|\mathcal{A}(\sigma)|$ can be used instead.

Simulation

In the simulation phase, the state obtained by the selection phase is extended to a final state τ via a Monte-Carlo rollout. In the context of combinatorial optimization the final state is a feasible solution. The rollout is typically done by random sampling of the possible actions $\mathcal{A}(\sigma)$ from state σ following a stochastic policy. For instance, one can use the probability distribution given by $Pr(\sigma|a) \mid a \in \mathcal{A}(\sigma)$. Alternatively, this can be done by any randomized greedy heuristic tailored to the problem at hand [14, 16, 20]. As mention before, hybridization with existing technologies can take place in this phase, whether it is a linear relaxation [21], a local search [9] or a call to a CP solver [13].

Backpropagation

Finally for each node σ traversed during the selection procedure, we update its statistics regarding the final state τ obtained by the simulation phase:

$$\begin{aligned} V(\sigma) &\leftarrow V(\sigma) + \frac{z(\tau, \sigma) - V(\sigma)}{N(\sigma) + 1} \\ N(\sigma) &\leftarrow N(\sigma) + 1 \end{aligned}$$

with $z(\tau, \sigma)$ the outcome of the rollout τ evaluated from node σ . The first update rule allows to maintain the average outcome of the rollouts for each node traversed during the selection step. It is possible to change the rule to only keep the best outcome found when traversing the node, instead of the average [21, 20]. The rationale is the same as minimax algorithms for games, the optimistic view is that eventually search will find the best completion of a partial solution, and therefore its expected value is closer to the best rollout than to the average of all rollouts. However, the preferred choice may depend on the standard deviation of the outcomes of rollouts under a given node, and on the ratio of the whole search tree that the algorithm will eventually explore. For this reason, for larger problems, and when the heuristic used during the rollouts is robust, the average may be better.

4 Tailoring Monte-Carlo Tree Search to Combinatorial Optimization

In this section, we introduce three modifications of standard Monte Carlo Tree Search which we empirically found beneficial in the context of optimization problems. These modifications are generic, in the sense that they hold outside of our industrial application, as long as we have a lower bound computation technique for the objective function and a depth first search procedure for the target problem.

4.1 Evaluation based on the objective function

In game playing, the outcome of a Monte-Carlo rollout may only be known when the game ends. Typically, the rollout is given a value of 1 for a win, -1 for a loss and 0 for a draw. Standard adaptations to combinatorial optimization are to normalize the objective value in a way or another as described in Section 3.

When simulating long branches, however, a “mistake” on a single decision along the branch can make the final outcome irrelevant. In fact, look-ahead methods often exhibit diminishing returns. For instance, it was observed in Chess that the rate of wins in self-plays between an algorithm looking $k + 1$ plies ahead versus the same algorithm looking k plies ahead declines as k grows [10]. In the case of a greedy procedure, it is therefore natural to conjecture that as the length of the branch grows, the correlation between the quality of the initial decision and the overall outcome decreases.

In combinatorial optimization problems, however, we usually have a lower bound on the objective that monotonically grows with every decision. Therefore, the evolution of this value can provide a better insight into the quality of an initial decision. Let $LB : \mathcal{A}^* \mapsto \mathbb{R}$ be a lower bound on sequences of actions, with $LB(\sigma)$ equals to the objective value if σ is a final state. Then, for a given node σ we propose to evaluate a state σ' reachable from σ as the sum of the marginal increment of the lower bound LB in the path from σ to σ' , weighted by an exponentially decaying coefficient γ . Hence we can define this sum recursively as follows:

$$z(\sigma', \sigma) = \begin{cases} LB(\sigma) - LB(p(\sigma)) & \text{if } \sigma' = \sigma \\ \gamma^{|\sigma'| - |\sigma|} (LB(\sigma') - LB(p(\sigma'))) + z(p(\sigma'), \sigma) & \text{otherwise} \end{cases} \quad (5)$$

The evaluation of a final state τ obtained by a rollout is then simply $z(\tau, \sigma)$ and represents an upper bound of the optimal solution.

Algorithm 1 implements backpropagation following the reward defined in Equation 5. This algorithm takes as an input the sequence (R) of the lower bound increments given by the rollout, the node selected in the *selection* phase, and the decay rate.

■ **Algorithm 1** Backpropagation procedure.

Data: R : sequence of the lower bound increments, σ : selected node, γ : decay rate

```

1 // Sum of exponentially decaying marginal increment of the lower bound
2  $val \leftarrow \sum_{i=1}^{|R|} \gamma^{i-1} R_i$ 
3 // Backpropagation until the root node
4 repeat
5    $val \leftarrow \gamma * val + LB(\sigma) - LB(p(\sigma))$ 
6    $N(\sigma) \leftarrow N(\sigma) + 1$ 
7    $V(\sigma) \leftarrow V(\sigma) + \frac{val - V(\sigma)}{N(\sigma)}$ 
8    $\sigma \leftarrow p(\sigma)$ 
9 until  $\sigma = Nil$ ;
```

The proposed evaluation method puts more weight on the short-term impact of a decision, wagering on it being more reliable than long term observations. For $\gamma = 1$, the score reflects the objective value $LB(\tau)$ of the rollout, whereas greater weight is put on short-term impacts when γ tends towards 0.

Moreover, the lower bound computations can be used during the expansion phase to avoid expending into sequences whose objective value cannot be lower than the current upper bound (best known solution). Thus, a node σ' that cannot be expanded further (all potential children nodes are suboptimal) is removed from the search tree. In that case, the information is backpropagated along the branch that leads to this node, that is, each node σ containing the deleted node σ' in its subtree are updated:

$$V(\sigma) \leftarrow \frac{1}{N(\sigma) - N(\sigma')} (V(\sigma) * N(\sigma) - V(\sigma') * N(\sigma'))$$

and

$$N(\sigma) \leftarrow N(\sigma) - N(\sigma')$$

Indeed, all information contained in the deleted node is now irrelevant for the rest of the search as it is not in the tree anymore. Then previous iterations which have passed throughout this node should not have an impact on the future search.

Then, for the implementation of the proposed evaluation function, we should store $LB(\sigma)$ at each node σ in addition to the triplet $\{N(\sigma), Pr(\sigma), V(\sigma)\}$.

A potential limit with this evaluation method is that it may skew search towards postponing actions that greatly increase the lower bound, but must eventually be done. For instance, consider a Travelling Salesman Problem with an isolated city far away from all other cities. Rollouts where this city is visited last will be preferred to rollouts where it is visited early. Lower bounds that take into account the future decisions in a reasonable way (e.g., minimum spanning tree for the travelling salesman problem, or the prehemptive relaxation in scheduling) may prevent this phenomenon since the cost of an exceptionally remote city or of an exceptionally large task would contribute to the lower bound anyways. The lower bound we used in our industrial problem, however, is extremely basic and yet this did not seem to be an issue in our experiments.

4.2 Dynamic Exploitation vs Exploration Balance

Since the tree grows deeper as search progresses, the likelihood to deviate from the best branch increases. Therefore, we propose to dynamically adapt the parameter that control the balance between exploration and exploitation, depending on the depth of the tree, in order to promote exploitation on deeper nodes. Let $td(\mathcal{T})$ be the depth of the tree \mathcal{T} , then at step t of the selection phase, the exploitation/exploration coefficient will be

$$\beta^{td(\mathcal{T})-t} * c \quad (6)$$

with $\beta < 1$ a parameter. This mechanism has a similar effect as *committing to a move* at the root node. At the root $t = 0$ and thus $\beta^{td(\mathcal{T})-t} * c$ tends towards 0 when $td(\mathcal{T})$ grows, so the first decision is very unlikely to deviate from the most promising choice once the search tree has sufficiently grown. Conversely, at a leaf, this term tends towards the original value c and hence less promising – but less frequently visited – nodes will be selected more often. In the context of games, when a move is actually made, it makes sense to forget the siblings and parents of the corresponding state. In optimization, this mechanism has been implemented in several approaches in order to limit the combinatorial explosion [3, 14]. Since commits are irreversible, the algorithm is no longer complete, and budget parameters controlling such commits need to be carefully chosen. Instead, the mechanism we propose has a similar effect but in a “smooth” way: near the root, it is more likely that the best move will be chosen, however other states can still be reached.

4.3 Depth First Search as a rollout

Finally we propose to use a Depth First Search procedure instead of a randomized greedy heuristic in the simulation phase. More precisely, in order to intensify the search around promising areas, a budget is defined after a first greedy “dive” and a budget-limited DFS is performed. For this purpose, the simulation is split into three steps:

- The first step is a greedy randomized procedure from the selected node σ until a contradiction is detected. This contradiction can happen because a constraint is violated, or because the lower bound exceeds the upper bound. At this point, we define a budget for the DFS by evaluating the current state σ' . This budget will be larger if this is a promising state, and maximal if no contradiction was encountered (and hence a new upper bound was found). On the other hand, if the state σ' is not promising, then the budget will be smaller or null.
- The second step of the simulation is a DFS, from the state reached by the greedy procedure σ' . This search is only performed on the subtree rooted at the node σ selected in the selection phase. The DFS algorithm must be able to store the best branch discovered, that is, the best solution or the best partial sequence according to the evaluation we described previously.
- The third step begins when the budget is consumed (or the search is complete for the subtree rooted at the selected node σ in the selection phase). If no solution was found during the previous step, the greedy randomized procedure is used to extend the best branch found by the DFS to a complete solution which can be evaluated before backpropagation.

The evaluation procedures for the states and for the budget will be detailed in Section 5 as their definitions depends on the considered problem.

5 Adaptation to the industrial Workshop Scheduling Problem

Tree model

In the search tree of the MCTS method, a state σ represents a partial sequence of operations and the set of actions correspond to the set of operations of the routing problem described in Section 2, i.e., actions are operations $\mathcal{A} = \mathcal{O}$. In the search tree, a sequence $\sigma|a$ is the sequence σ extended by the action (operation) a . The set of possible actions $\mathcal{A}(\sigma)$ from a sequence σ contains every operation a such that the (partial) sequence $\sigma|a$ is feasible with respect to the constraints.

Objective function

Since our industrial application is a satisfaction problem (the existence of a tour without delay), we need to generalize it to an optimization problem to apply MCTS as described in Sections 3 and 4. Therefore, during the simulation phase, we relax the due date constraints and instead we minimize the maximum tardiness:

$$L(\sigma) = \max(0, \max_{1 \leq j \leq |\sigma|} (e_{\sigma(j)} - d_{\sigma(j)}))$$

Since in this case operations can finish later than their due dates, it is necessary to make the precedence constraints due to production cycles explicit:

$$\max(\rho(df_i^{k-1}), \rho(de_i^{k-1})) < \min(\rho(pe_i^k), \rho(pf_i^k)) \quad \forall i \in [1, m] \quad \forall k \in [2, n_i] \quad (7)$$

Furthermore, during the expansion phase we do not add a child node that would violate a due date constraint, as our primary goal is to find a solution σ without any late job, that is, such that $L(\sigma) = 0$.

We use a trivial lower bound, which is at state σ the maximum tardiness $L(\sigma)$ of the associated partial sequence also taking into account tardiness of all pending operations. Pending operations are all the operations that belong to a production cycle in which at least one operation is available to extend the current sequence, ignoring the train constraint. Therefore, Equation (5) is the sum of exponentially decaying marginal increments of the maximum tardiness with a small look ahead.

Heuristic

For the simulation phase as well as for the probabilities of the expansion phase, we use the heuristic tuned by reinforcement learning proposed in [2]. This heuristic is stochastic and provides a probability distribution over the set of available operations for a given state. More precisely, at a given state σ , each operation $a \in \mathcal{A}(\sigma)$ is evaluated using a fitness function $f(\sigma, a)$ defined as a linear combination of four criteria: $f(\sigma, a) = \theta^T \lambda(\sigma, a)$. These criteria λ_i correspond to:

1. The *emergency* of the operation: $lst(a, \sigma) - \max(r_a, e_{\sigma(|\sigma|)} + tt_{\sigma(|\sigma|), a})$, with $lst(a, \sigma)$ the latest starting time of the operation a in order to satisfy the due date constraints with respect to the operations belonging to σ and the precedences constraints;
2. The *travel/waiting time* of the operation: $\max(tt_{\sigma(|\sigma|), a}, (r_a - e_{\sigma(|\sigma|)}))$;
3. The (negated) *length* of the trolley;
4. The *type* of operation, equal to 1 for pickups and 0 for deliveries.

The parameter θ is set to the proposed learned values (0.251, 0.576, 0.148, 0.023). Then, a **softmax** function is applied to turn the fitness evaluation into a probability distribution for guiding the choice of the next node in the greedy heuristic:

$$\forall o \in \mathcal{A}(\sigma) \quad \pi_{\theta}(o \mid \sigma) = \frac{e^{(1-f(\sigma,o))/\delta}}{\sum_{o' \in \mathcal{A}(\sigma)} e^{(1-f(\sigma,o'))/\delta}} \quad (8)$$

where the parameter δ controls the “greedyness” of the heuristic, that is, a “low” value for δ encourages to select the best choice with high probability, whereas a more “neutral” value of δ produces more randomized choices. In the experiments, we will set a value of $\delta = 0.005$ in the simulation phase, and a value of $\delta = 0.1$ to initialize the prior probabilities of the new nodes in the expansion phase.

Simulation

The greedy procedures before and after the DFS simply consist in taking at random the next operation following the probability distribution defined by equation 8.

For the DFS we define a backtrack budget between 0 and \mathcal{B} , depending on when the first tardiness was detected during the first dive. If the first dive finds an improving solution, then the budget is maximum (\mathcal{B}), in order to find other related improving solutions. Otherwise, we rely on the rank ϕ where the lower bound became positive to define the budget. Let ϕ^* be the highest rank for any previous solution, the backtrack budget is then:

$$\begin{cases} \mathcal{B} & \text{if } \phi \geq \phi^* \\ \mathcal{B}(\frac{\phi^* - \phi}{\phi^* - \alpha * \phi^*})^2 & \text{if } \phi^* > \phi > \alpha * \phi^* \\ 0 & \text{otherwise} \end{cases} \quad (9)$$

with $\alpha \leq 1$, a threshold parameter.

During the DFS, we define a probability distribution over the children using the **softmax** function of the greedy heuristic and we limit the breadth of the tree by keeping only actions with a probability greater than 10^{-6} , which typically leaves all but 1 to 3 children approximately. Then, those children are sorted by their probabilities, and in order to randomize the DFS, a random child (again, using the same probability distribution) is swapped with the first one, to be branched on first by the DFS. As instances can be very large, this is sufficient to keep variety in the solutions, while removing many “bad” decisions. This is also why we rely on the backtrack count instead of the fail count to define the budget, as a lot of nodes may have only one child. We add a geometric restart policy in the DFS step, where the search is reset to the node selected in the selection phase. The growth factor is reset at each MCTS iteration. At the end, the DFS returns the longest (potentially partial) sequence for which the lower bound remains null (i.e., for as the largest number of operations). Then, the greedy procedure is called to extend this sequence to a complete solution.

6 Experimental Evaluation

We report in this section the results of our experiments. First, we assess the respective impact of using the new evaluation policy, the dynamic exploration/exploitation balance and the DFS in the simulation phase. In a second part, we compare our MCTS adaptations to state-of-the-art methods for this problem.

6.1 Experimental protocol

We use the same data set as in [2] composed of 120 synthetic instances. The data set is made of four categories characterized by the number of components (15 in category A, 20 in B, 25 in C and 30 in D). Moreover, all of these categories are associated to three time horizons: a work shift of an operator (7 hours and 15 minutes), a work day (made up of three shifts) and a full week (6 days).

The number of components is highly correlated with hardness, and directly related to the branching factor in the Monte-Carlo search tree. Indeed, each node has at most two children per component (ie., from 30 children for instances of category A to 60 children for instances of category D). In addition, the depth of the search tree grows with the number of operations, that depends both on the time horizon and on the number of components. This depth varies from 450 for the “shift” schedules, up to 14500 for the “weekly” schedules.

We ran every method 10 times for each of the 120 instances with a timeout of 1h. All experiments were run on a cluster composed of Xeon E5-2695 v3 @ 2.30GHz processors. Our methods were implemented using in C++ and compiled with GCC-8.0. The two methods from [2] were implemented using JAVA and were run in the same conditions, and Choco-4.10 [18] for CP.

6.2 Impact of the MCTS adaptations

In the first part of the experiments, the goal is to assess the respective impact of the proposed adaptations for the MCTS method. We evaluated 6 different versions of the MCTS, adding the adaptations we propose one at a time:

- MCTS is the standard MCTS method without any of the proposed adaptation. This baseline method uses the value of the objective function as the result of the rollouts, and backpropagates this value through the tree to the root node.
- MCTS+DFS is the same algorithm as MCTS except that it uses the DFS in the simulation phase.
- SEDMI is the variant of MCTS that uses the sum of exponentially decaying marginal increments of the lower bound to evaluate the nodes.
- SEDMI+DFS adds the DFS to SEDMI for the simulation phase.
- SEDMI+DFS+DC extends SEDMI+DFS with the dynamic exploitation/exploration compromise.
- SEDMI+SAT-DFS+DC is the variant of SEDMI+DFS+DC in which the upper bound on the objective function is fixed to 1 in the DFS, i.e. the DFS tries to solve the satisfaction version of the problem instead of trying to improve the global upper bound. However, the last part of the simulation still provides a complete solution via a greedy procedure, and hence this method also provides an upper bound.

All parameters for the proposed methods are given in Table 1. We recall that c is the exploitation/exploration tradeoff parameter. The higher value for this parameter, the more the MCTS will explore. Then, β is the decay rate for the adaptation of c , and γ is the decay rate of the evaluation function. Finally, α and \mathcal{B} are respectively the threshold parameter, and the maximum backtrack budget for the DFS. All the values for these parameters were chosen by preliminary experiments, and the chosen combination appears to give relatively good overall results.

The results are shown in Table 2 and 3, in which we report the number of solved runs, and the average maximum tardiness. For all the methods we consider that an instance is solved if and only if the value of the objective function is null i.e. there is no tardiness. Table 2 shows the performance of the different variants of the MCTS averaged by classes of

■ **Table 1** Parameters value.

c	1
β	0.995
γ	0.9977
α	0.9
\mathcal{B}	50000
Restart (base)	100
Restart (factor)	1.2

■ **Table 2** Comparison of the MCTS adaptations.

H		MCTS		MCTS+DFS		SEDMI		SEDMI+DFS		SEDMI+DFS+DC		SEDMI+SAT-DFS+DC	
		#S	L_{max}	#S	L_{max}	#S	L_{max}	#S	L_{max}	#S	L_{max}	#S	L_{max}
A	shift	100	0	100	0	100	0	100	0	100	0	100	0
	day	90	135	90	133	90	115	90	77	100	0	98	0
	week	68	1996	78	1850	70	1800	80	1839	77	1840	80	1858
B	shift	80	420	79	258	90	372	82	353	81	349	87	439
	day	50	2954	54	2959	60	2522	60	2439	63	2121	70	2134
	week	10	21070	29	20771	10	20572	32	20541	31	20355	36	20635
C	shift	49	1676	48	1708	40	1901	45	1727	40	1824	40	2012
	day	10	9503	11	9248	10	8683	26	8656	36	8747	35	9022
	week	0	64442	8	64713	0	64480	9	64584	9	64474	10	64445
D	shift	40	2154	33	2146	30	2338	33	2018	30	2304	30	2621
	day	0	13659	0	13664	0	12657	0	12723	13	12225	11	12340
	week	0	101474	0	101444	0	100533	0	100760	0	100954	0	100840
Average		41	18290	44	18241	42	17998	46	17976	48	17933	50	18029

instances, and by time horizons. In this table, for each method, a line corresponds to 100 runs (10 instances and 10 runs for every time horizon), then the number of solved runs is a sum over these 100 runs. In Table 3 the same results are presented aggregated by time horizons, and the number of solved instances is in percentage (over the 400 runs by line and by method).

In these tables, we can see the benefit of using the DFS in the simulation phase. Using DFS, as expected, allows the MCTS methods to solve more instances on the *week* horizon. In fact, those instances are too large to be solved via rollouts only, and the DFS allows to intensify the search on the deepest parts of the tree, that are not explored in the MCTS. Unfortunately, the effect of the DFS is not visible on the *shift* horizon. We can also see the benefit of using the sum of exponentially decaying marginal increments as node evaluation on *day* and *week* horizons in terms of objective value. However, this adaptation slightly degrades the performance on shorter horizon meaning that this time horizon is too short to

■ **Table 3** Results aggregated by time horizon.

H	MCTS		MCTS+DFS		SEDMI		SEDMI+DFS		SEDMI+DFS+DC		SEDMI+SAT-DFS+DC	
	#S	L_{max}	#S	L_{max}	#S	L_{max}	#S	L_{max}	#S	L_{max}	#S	L_{max}
Shift	0.67	1063	0.65	1028	0.65	1153	0.65	1024	0.63	1119	0.64	1268
Day	0.38	6562	0.39	6501	0.40	5994	0.44	5974	0.53	5773	0.54	5874
Week	0.20	47245	0.29	47195	0.20	46846	0.30	46931	0.29	46906	0.32	46944

take advantage of this mechanism. Overall, the combination of both mechanisms outperforms the two versions with only one of these mechanisms. Finally, the effect of the dynamic compromise can be seen on the *day* horizon. This time horizon is small, but not enough for the MCTS to advance deep enough in the search tree to find solutions. This mechanism forces the MCTS to explore the tree deeper and faster, and as a results, to improve the number of solved instances.

6.3 Comparison with previous methods

For the second part of the experiments, we compare the two best MCTS methods, namely **SEDMI+DFS+DC** (the method leading to the lowest objective function) and **SEDMI+SAT-DFS+DC** (the method with the highest number of solved instances) to the two best methods introduced in [2], that are both based on the stochastic branching policy described in Section 5:

- **CP**: a constraint programming approach with rapid restarts. This method solves the satisfaction version of the problem. As a result, it is slightly better for finding solutions without tardiness.
- **GRASP**: a multi-start local search procedure. This method considers the optimization problem with relaxed due dates as in the MCTS methods, hence we can compare the overall tardiness.

■ **Table 4** Comparison with previous methods.

	<i>H</i>	CP	GRASP		SEDMI+DFS+DC		SEDMI+SAT-DFS+DC	
		#S	#S	L_{max}	#S	L_{max}	#S	L_{max}
A	shift	90	90	10	100	0	100	0
	day	90	90	193	100	0	98	0
	week	80	70	2433	77	1840	80	1858
B	shift	60	60	467	81	349	87	439
	day	52	46	3218	63	2121	70	2134
	week	35	10	26915	31	20355	36	20635
C	shift	40	40	1941	40	1824	40	2012
	day	10	10	9498	36	8747	35	9022
	week	10	0	71104	9	64474	10	64445
D	shift	19	16	2677	30	2304	30	2621
	day	0	0	13994	13	12225	11	12340
	week	0	0	107186	0	100954	0	100840
Average		40.5	36	19969	48	17933	50	18029

The results, given in Table 4, show that overall, the proposed MCTS adaptations outperform the CP and the local search approaches on both criteria: the number of solved instances, and the maximum tardiness. More precisely, the dominance is clear for horizons *shift* and *day* in terms of number of instances solved, but we can see that our method does not outperform the CP model on the *week* horizon. Finally, between the CP approach and the **SEDMI+SAT-DFS+DC** variant, there is a difference of 9.5% of instances solved in favor of the latter. There is still half of the instances that are not solved to optimality. However, the instances of the data set were randomly generated without a guarantee of satisfiability, and, we believe that the majority of unsolved instances are not satisfiable (especially for the week horizon).

7 Conclusion

In this paper, we have presented and applied several variants of the Monte Carlo Tree Search method to solve a repetitive single vehicle pickup and delivery problem with time windows and capacity constraint, issuing from car manufacturing assembly lines. We defined a way of evaluating the rollouts based on the growth of the lower bound of the objective function. We also proposed an adaptation of the balance parameter between exploitation and exploration in order to be able to solve larger instances. Moreover, we proposed an hybridization of Monte Carlo Tree Search with Depth First Search used during the simulation phase. The experimental evaluation demonstrates that these proposals allow us to outperform previous approaches on the considered problem, and show the benefit of our contributions.

These three proposals, although well suited to a dedicated problem, are generic. The next step is then to demonstrate the genericity of these Monte Carlo Tree Search variants by considering their application to other combinatorial optimization problems. We also plan to integrate our MCTS method in existing constraint programming solvers to take advantage of their search tree exploration in the Depth First Search part, further reinforcing the hybrid nature of the approach. Finally, we would like to explore further the learning aspects of the method. Indeed, in the simulation phase, we are repeatedly dealing with similar subproblems in different part of the tree, and the policy used in a subtree could be adjusted after each iteration in order to have different policies adapted to different parts of the tree search.

References

- 1 David Allouche, Simon de Givry, George Katsirelos, Thomas Schiex, and Matthias Zytnicki. Anytime hybrid best-first search with tree decomposition for weighted CSP. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP)*, pages 12–29, 2015. doi:10.1007/978-3-319-23219-5_2.
- 2 Valentin Antuori, Emmanuel Hebrard, Marie-José Huguet, Siham Essodaigui, and Alain Nguyen. Leveraging Reinforcement Learning, Constraint Programming and Local Search: A Case Study in Car Manufacturing. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 657–672, 2020. doi:10.1007/978-3-030-58475-7_38.
- 3 Dimitris Bertsimas, J. Daniel Griffith, Vishal Gupta, Mykel J. Kochenderfer, and Velibor V. Misić. A comparison of Monte Carlo tree search and rolling horizon optimization for large-scale dynamic resource allocation problems. *European Journal of Operational Research*, 263(2):664–678, 2017. doi:10.1016/j.ejor.2017.05.032.
- 4 Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. doi:10.1109/TCIAIG.2012.2186810.
- 5 Guillaume Chaslot, Steven Jong, Jahn-Takeshi Saito, and Jos Uiterwijk. Monte-Carlo Tree Search in Production Management Problems. In *Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC)*, pages 91–98, January 2006.
- 6 Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proceedings of the 5th International Conference on Computers and Games (CG)*, pages 72–83, 2006. doi:10.1007/978-3-540-75538-8_7.
- 7 Gecode Team. Gecode: Generic constraint development environment, 2006. Available from <http://www.gecode.org>.
- 8 Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *Proceedings of the 24th International Conference on Machine Learning (ICML)*, pages 273–280, 2007. doi:10.1145/1273496.1273531.
- 9 Jack Goffinet and Raghuram Ramanujan. Monte-Carlo Tree Search for the Maximum Satisfiability Problem. In *Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming (CP)*, pages 251–267, 2016. doi:10.1007/978-3-319-44953-1_17.

- 10 Ernst A. Heinz. New Self-Play Results in Computer Chess. In *Proceedings of the Second International Conference on Computers and Games (CG)*, pages 262–276, 2000. doi:10.1007/3-540-45579-5_18.
- 11 Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning (ECML)*, pages 282–293, 2006. doi:10.1007/11871842_29.
- 12 Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning (ECML)*, ECML’06, page 282–293, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11871842_29.
- 13 Manuel Loth, Michèle Sebag, Youssef Hamadi, and Marc Schoenauer. Bandit-Based Search for Constraint Programming. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 464–480, 2013. doi:10.1007/978-3-642-40627-0_36.
- 14 Jacek Mandziuk and Cezary Nejman. UCT-Based Approach to Capacitated Vehicle Routing Problem. In *Proceedings of the 14th International Conference on Artificial Intelligence and Soft Computing (ICAISC)*, pages 679–690, 2015. doi:10.1007/978-3-319-19369-4_60.
- 15 Shimpei Matsumoto, Noriaki Hirosue, Kyohei Itonaga, Nobuyuki Ueno, and Hiroaki Ishii. Monte-carlo tree search for a reentrant scheduling problem. In *Proceedings of the 40th International Conference on Computers Industrial Engineering (CIE)*, pages 1–6, 2010. doi:10.1109/ICCIE.2010.5668320.
- 16 Minh Anh Nguyen, Kazushi Sano, and Vu Tu Tran. A monte carlo tree search for traveling salesman problem with drone. *Asian Transport Studies*, 6:100028, 2020. doi:10.1016/j.eastsj.2020.100028.
- 17 Alessandro Previti, Raghuram Ramanujan, Marco Schaerf, and Bart Selman. Monte-Carlo Style UCT Search for Boolean Satisfiability. In *Proceedings of the 12th International Conference of the Italian Association for Artificial Intelligence (AI*IA)*, pages 177–188, 2011. doi:10.1007/978-3-642-23954-0_18.
- 18 Charles Prud’homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Solver Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. URL: <http://www.choco-solver.org>.
- 19 Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006. doi:10.1287/trsc.1050.0135.
- 20 Thomas Philip Runarsson, Marc Schoenauer, and Michèle Sebag. Pilot, Rollout and Monte Carlo Tree Search Methods for Job Shop Scheduling. In *Proceedings of the 6th International Conference on Learning and Intelligent Optimization (LION)*, pages 160–174, 2012. doi:10.1007/978-3-642-34413-8_12.
- 21 Ashish Sabharwal, Horst Samulowitz, and Chandra Reddy. Guiding combinatorial optimization with UCT. In *Proceedings of the 9th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, pages 356–361, 2012. doi:10.1007/978-3-642-29828-8_23.
- 22 David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. doi:10.1038/nature16961.
- 23 David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017. doi:10.1038/nature24270.

Practical Bigraphs via Subgraph Isomorphism

Blair Archibald ✉ 🏠 

School of Computing Science, University of Glasgow, UK

Kyle Burns ✉ 

School of Computing Science, University of Glasgow, UK

Ciaran McCreesh ✉ 🏠 

School of Computing Science, University of Glasgow, UK

Michele Sevegnani ✉ 🏠 

School of Computing Science, University of Glasgow, UK

Abstract

Bigraphs simultaneously model the spatial and non-spatial relationships between entities, and have been used for systems modelling in areas including biology, networking, and sensors. Temporal evolution can be modelled through a rewriting system, driven by a matching algorithm that identifies instances of bigraphs to be rewritten. The previous state-of-the-art matching algorithm for bigraphs with sharing is based on Boolean satisfiability (SAT), and suffers from a large encoding that limits scalability and makes it hard to support extensions. This work instead adapts a subgraph isomorphism solver that is based upon constraint programming to solve the bigraph matching problem. This approach continues to support bigraphs with sharing, is more open to other extensions and side constraints, and improves performance by over two orders of magnitude on a range of problem instances drawn from real-world mixed-reality, protocol, and conference models.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Theory of computation → Models of computation

Keywords and phrases bigraphs, subgraph isomorphism, constraint programming, rewriting systems

Digital Object Identifier 10.4230/LIPIcs.CP.2021.15

Supplementary Material *Software (Source Code)*: <https://doi.org/10.5281/zenodo.5161185>

Funding *Blair Archibald*: supported by the EPSRC under grant S4: Science of Sensor Systems Software (EP/N007565/1).

Kyle Burns: supported by the EPSRC under a Doctoral Training Partnership (EP/R513222/1).

Ciaran McCreesh: supported by the EPSRC under grant Modelling and Optimisation with Graphs (EP/P026842/1).

Michele Sevegnani: supported by the EPSRC under PETRAS SRF grant MAGIC (EP/S035362/1).

1 Introduction

Bigraphs are a universal modelling formalism, used to represent both the spatial relationships of entities and their global interactions. Since their introduction by Milner [24], they have been used to model, amongst others: IoT/sensor systems [27, 6], Mixed-Reality systems [9], networking protocols [10, 11], security [1], and biological systems [18]. A bigraph consists of two graph-based structures over the same set of vertices: a *place graph* describing the nesting of entities, e.g. a device within a room, and a *link graph* describing non-local relationships through hyperedges, e.g. a device connected to (numerous) other devices regardless of location. A Bigraphical Reactive System (BRS) allows bigraphs to evolve over time through a set of reaction rules, of the form $L \longrightarrow R$, that *find*, through a matching algorithm, an instance of bigraph L in a bigraph B and replace it with bigraph R . With a BRS, model verification is performed either through reachability analysis over the transition system generated by the reaction rules, or by simulation.



© Blair Archibald, Kyle Burns, Ciaran McCreesh, and Michele Sevegnani;
licensed under Creative Commons License CC-BY 4.0

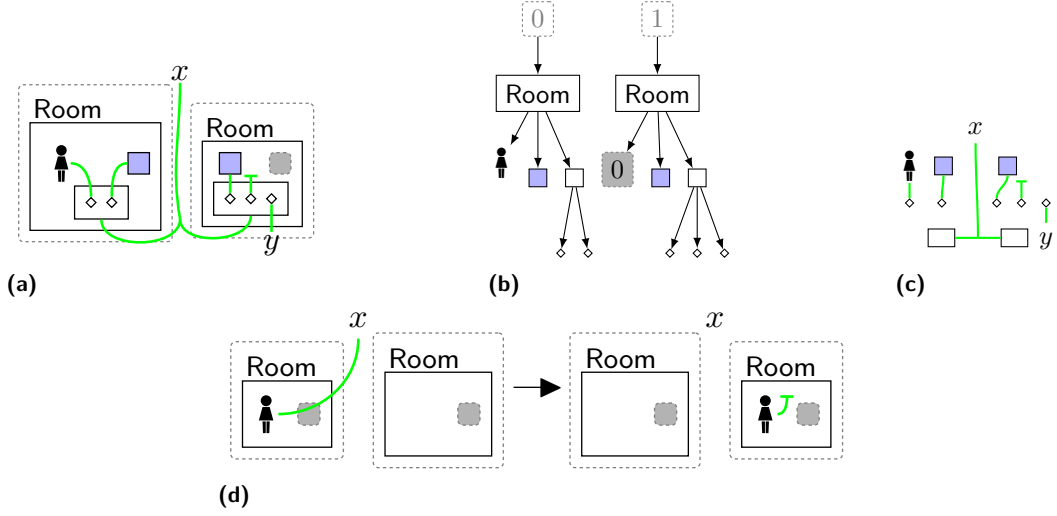
27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 15; pp. 15:1–15:17

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** (a) Bigraph example with Rooms, People, Computers (blue squares), Routers (clear rectangles), and Sockets (diamonds); (b) Place graph for (a); (c) Link graph for (a), unlinked entities not drawn; (d) Reaction rule to move people between rooms.

Efficient matching and rewriting routines are essential for practical analysis of large models, and even small improvements per match can have significant impacts on the overall analysis time, given the huge number of matches. In this work we show that a bigraph matching algorithm implemented on top of a constraint programming solver for the subgraph isomorphism problem provides a *performant* and *extensible* basis for bigraph matching. We provide an encoding of bigraphs to graphs, in such a way that we can solve the bigraph matching problem using a variant of the subgraph isomorphism problem (SIP) with additional constraints. This process supports both standard bigraphs, and the bigraphs with sharing extension. Using a set of real-world models, we show empirically that in *all cases* the SIP solver outperforms the previous state-of-the-art SAT solver found in the open-source BigraphER [26] toolkit, with a speedup of more than two orders of magnitude.

2 Background

We begin by giving the necessary background to present our contributions. In this section we describe the concepts and theory underlying bigraphs, and then explain bigraph matching and subgraph matching problems. The following section will explain how these matching problems can be related.

2.1 Bigraphs

Bigraphs simultaneously model systems based on both spatial and non-local relationships between entities. Throughout this paper we use bigraphs to refer to bigraphs with sharing, which allow entities to have multiple parents.

Bigraphs have equivalent algebraic and diagrammatic notation. Throughout this paper we use the diagrammatic notation when possible. An example bigraph is shown in Figure 1a. This simple model represents people and computers within rooms and their links to specific *sockets* (shown as diamonds) on a router. We use shapes and colour to denote different entity types. Nesting and adjacency of entities represents spatial relationships, e.g. the

person is in the first room. The green hyperlinks represent non-spatial relationships, e.g. links between routers in different rooms. Importantly, entities have *fixed* arity (number of links), e.g. sockets have arity 1, although links might not be linked elsewhere as shown by the second to last socket.

Spatial relationships are captured by the place graph, shown in Figure 1b, that forms a *directed-acyclic-graph* (DAG)¹ over entities. Top-level places, shown as dashed rectangles, are called *regions* that represent unknown (or empty) parent(s). The grey dashed rectangle is called a *site*; similar to regions, sites represent an unknown (or empty) child bigraph.

Non-spatial relationships are captured by the link graph, shown in Figure 1c, that forms a *hypergraph* over entities. Here, regions/sites are replaced by outer/inner names, i.e. they represent (potentially) other entities on the same link. We draw outer names above the diagrams and inner names below. A link is *open* if it connects to a name, and *closed* otherwise (i.e. it connects only the specified entities).

Regions/sites and outer/inner names – called *interfaces* – allow us to build bigraphs compositionally. That is, we can build larger bigraphs from smaller bigraphs. This is done by placing regions into sites and connecting on like-names. For example, the bigraph in Figure 1a accepts a bigraph with a single region and outer name y – adding it to the second room and linking up the incoming link to the rightmost socket – and can be composed with a bigraph with two sites (one for each region) that accepts a name x . Composition of bigraphs is shown in more detail in Section 2.3 to describe the bigraph matching problem. We denote the composition of bigraphs B_0 and B_1 as $B_0 \circ B_1$ (placing the regions of B_1 in the sites of B_0). Alternatively, we can build larger bigraphs through a tensor operation $B_0 \otimes B_1$ that places bigraphs side-by-side. In general, bigraphs form a specific type of symmetric monoidal category [24], although we do not need the full power of this fact in this paper.

2.2 Bigraph Definitions

We give enough definitions for bigraphs with sharing to explain our encoding and matching routines; full details are available elsewhere [25]. We use *concrete* bigraphs, where each entity and closed link is named. Models are usually defined over *abstract* bigraphs that represent an equivalence class of all bigraphs that have the same structure regardless of concrete names. We always perform *matching* on concrete bigraphs so this is sufficient for our purposes.

We assume a set \mathcal{K} of entity types (e.g. Room), an arity function $ar : \mathcal{K} \rightarrow \mathbb{N}$, \mathcal{V} a set of entity identifiers v_0, \dots, v_n , \mathcal{E} a set of link identifiers e_0, \dots, e_n , and \mathcal{X} a finite set of names x, y, z, \dots , such that all names and identifiers are disjoint.

► **Definition 1** (concrete place graph with sharing). A concrete place graph with sharing

$$B = (V_B, ctrl_B, prnt_B) : m \rightarrow n$$

is a triple having m sites and n regions (treated as ordinals)². B has a finite set $V_B \subset \mathcal{V}$ of entities, a control map $ctrl_B : V_B \rightarrow \mathcal{K}$, and a parent relation

$$prnt_B \subseteq (m \uplus V_B) \times (V_B \uplus n)$$

that is *acyclic* i.e. $(v, v) \notin prnt_B^+$ for any $v \in V_B$, with $prnt_B^+$ the transitive closure of $prnt$.

¹ A forest in standard bigraphs, and a DAG for sharing.

² The function notation is used as place graphs (resp. link graphs, bigraphs) are arrows in a category with ordinals, e.g. m, n (resp. sets of names, ordinal/name pairs) as objects.

► **Definition 2** (concrete link graph). *A concrete link graph*

$$B = (V_B, E_B, ctrl_B, link_B) : X \rightarrow Y$$

is a quadruple having (finite) inner name set $X \subset \mathcal{X}$ and an outer name set $Y \subset \mathcal{X}$. B has finite sets $V_B \subset \mathcal{V}$ of entities and $E_B \subset \mathcal{E}$ of links, a control map $ctrl_B : V_B \rightarrow \mathcal{K}$ and a link map

$$link_B : X \uplus P_B \rightarrow E_B \uplus Y$$

where $P_B \stackrel{def}{=} \{(v, i) \mid v \in V_B, i = ar(ctrl_B(v))\}$ is the set of ports of B .

Closed links are those where the domain is restricted to P_B and the image is in E_B . Otherwise they are open.

A concrete bigraph with sharing joins these two structures on V_B .

► **Definition 3** (concrete bigraph with sharing). *A concrete bigraph*

$$B = (V_B, E_B, ctrl_B, prnt_B, link_B) : \langle k, X \rangle \rightarrow \langle m, Y \rangle$$

consists of a concrete place graph with sharing $B^P = (V_B, ctrl_B, prnt_B) : k \rightarrow m$ and a concrete link graph $B^L = (V_B, E_B, ctrl_B, link_B) : X \rightarrow Y$. The inner and outer interfaces of B are $\langle k, X \rangle$ and $\langle m, Y \rangle$, respectively.

2.3 Bigraph Matching Problem

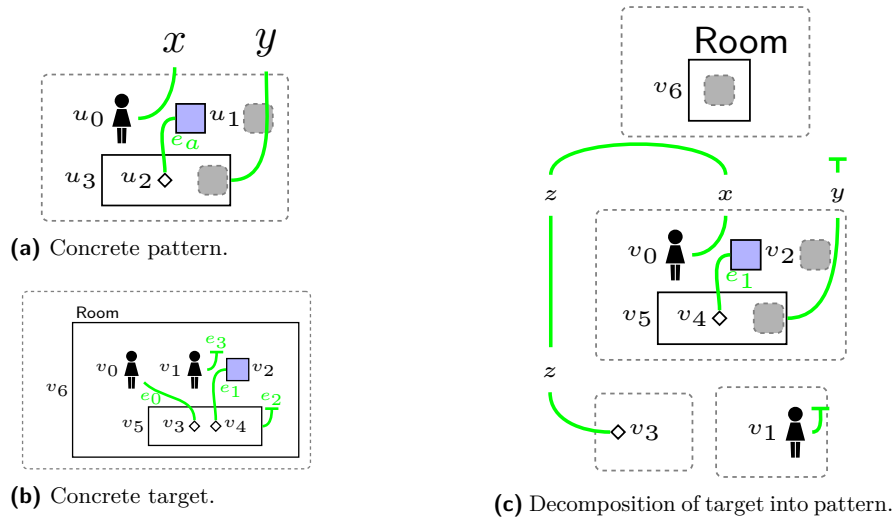
We denote the identity bigraph over an interface $I = \langle m, X \rangle$ by $id_I : I \rightarrow I$. It maps names in X to themselves and places m sites in m regions.

A Bigraphical Reactive System (BRS) allows bigraphs to evolve through a set of reaction rules of the form $L \rightarrow R$. Intuitively a reaction rule replaces an occurrence of a bigraph L with R . An example reaction rule, allowing a person to move between rooms, is in Figure 1d. The use of sites within the rooms allows them to contain any other entities. Reaction rules can update both the place and link graph simultaneously as shown by the connection being severed.

The central operation when computing over a BRS is the ability to *match* the left-hand-side of a reaction rule. Matches are also used to define state predicates, i.e. as bigraph patterns [9], and multiple matches per rewrite step are required for conditional rewriting [4]. If stochastic semantics are required [18] then the *number* of matches is required to correctly normalise rates. Bigraph matching is defined formally in terms of occurrences:

► **Definition 4** (concrete occurrence). *Let P and T be two concrete bigraphs representing a pattern and target. We say there is a concrete occurrence of P in T if there is a valid decomposition $T = C \circ (id_I \otimes P) \circ D$ for some interface I , and concrete bigraphs C and D . We call C the context and D the parameter. Two concrete occurrences are equal if they differ only by a permutation or a bijective renaming on the inner interface of C and the outer interface of D .*

It is always possible to determine an abstract occurrence starting from a concrete one. In other words, a bigraph P occurs in T (both abstract) only if an arbitrary concretisation of P occurs in an arbitrary concretisation of T , where *concretisation* means the assignment of distinct identifiers (drawn from \mathcal{V} and \mathcal{E}) to all entities and closed links.



■ **Figure 2** Example matching instance with $\{u_0 \rightarrow v_0, u_1 \rightarrow v_2, u_3 \rightarrow v_5, u_2 \rightarrow v_4, e_a \rightarrow e_1\}$. An alternative match with $u_0 \rightarrow v_1$ is possible.

An example match through decomposition is in Figure 2. We use the same entities as before, but this time give a concrete pattern/target bigraph with identifiers v_n, u_n, e_n . The pattern is as given, while all additional entities are placed either in the context (i.e. the Room) or the parameter (the additional socket/person). The additional wiring of z is through the id_I component of the decomposition. Because of the ability to loop outer names back to the parameter, for matching, we only need to consider open links and not distinguish between inner and outer.

Treating matching as a decomposition is essential since, while it is necessary to find an isomorphism between entities in the pattern and target graph, it is not sufficient. To have a *valid* match we *must* also be able to form a valid context/parameter. For example, we cannot have the same entity appear in both the context and the parameter.

Existing approaches

The first bigraph matching algorithm [15] made use of structural induction on the algebraic representation of bigraphs in order to find a valid match through an inference system. The algorithm supports both standard bigraphs and binding bigraphs that allow names to have locality. A similar inductive approach was used to provide the matching algorithm for directed bigraphs [8] that allows directed link graphs.

Like the approach we outline, other algorithms encode the bigraph matching problem as an instance of a combinatorial search problem, allowing re-use of existing tools for efficiency. For example, jLibBig [12], which also supports directed bigraphs, formulates matching as a constraint satisfaction problem, while BigraphER [26] is the only implementation to support bigraphs with sharing through the use of SAT solvers.

The closest approach to ours encodes bigraphs as *ranked graphs* [14]. Ranked graphs can be seen as graphs-with-interfaces, mirroring the sites/regions/names of bigraphs. Rather than perform the encoding only for matching, ranked graphs are used to do the *rewriting* (as an instance of double-pushout graph transformation) and then the entire structure is converted back to a bigraph. Assuming negligible encoding/decoding time, the performance of this approach depends on the underlying graph transformation framework.

2.4 Subgraph Isomorphism

The Subgraph Isomorphism Problem (SIP) is a classic NP-complete decision problem that determines whether a pattern graph is a subgraph of (i.e. is present in) a target graph. Because of its broad applicability, many dedicated solving algorithms exist, with the current state of the art being the *Glasgow Subgraph Solver* [22]. This solver adopts a constraint programming approach, combining inference and intelligent backtracking search, but with special data structures and algorithms designed specifically for subgraph problems. The solver supports variants of the problem, including non-induced and induced subgraph finding, graphs involving directed edges, and labelling schemes defining vertex and edge compatibilities. It can also explicitly enumerate all solutions, rather than just deciding whether one solution exists.

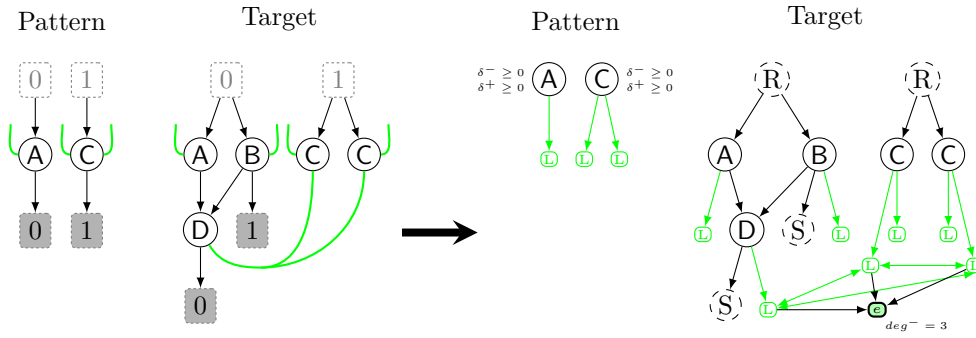
Formally, the problem we will be solving is as follows. Given a pattern directed graph $G = (V_G, E_G)$, a target directed graph $H = (V_H, E_H)$, and a vertex compatibility function³ $\ell : V_G \times V_H \rightarrow \{t, f\}$, a *non-induced subgraph isomorphism with vertex compatibility constraints* from G to H is an injective mapping $i : V_G \rightarrow V_H$ such that edges are mapped to edges, $(u, v) \in E_G \implies (i(u), i(v)) \in E_H$, and where vertex compatibility is respected, $\ell(v, i(v)) = t \ \forall v \in V_G$. We wish to enumerate *all* such mappings.

3 Bigraph Matching as Subgraph Isomorphism

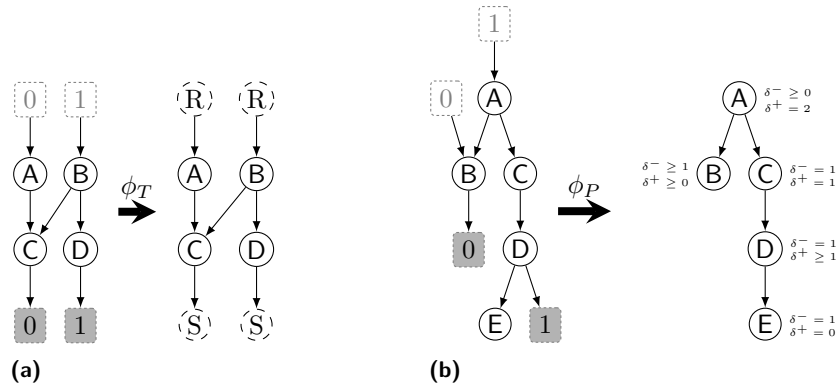
The key observation underlying our new approach is that bigraph matching instances can be seen as a SIP problem with additional constraints to handle abstractions (regions/sites, and names), and ensure valid decompositions (Section 2.3). Using a subgraph model rather than a SAT model is potentially beneficial for three reasons. Firstly, subgraph solvers can carry out stronger reasoning than SAT solvers, by using implied constraints based upon degrees and neighbourhood degree sequences [29], cardinality [28], distances [7], and path counts [20]. Secondly, we know how to design very good variable- and value-ordering heuristics for graph problems [21, 5]. And thirdly, subgraph solvers can potentially deal with much larger problem instances due to a compact representation of adjacency constraints.

However, most existing SIP solvers do not directly support hypergraphs, let alone multiple overlaid graph structures like bigraphs. In order to use existing tools we encode the pattern/target bigraphs into a traditional graph structure (with additional degree constraints). This is a two step process. First we encode place graphs by removing/replacing abstractions and then we *flatten* the link graphs into these encoded structures by replacing hyperedges with cliques resulting in a single *flattened* graph. This flattened graph representation is accepted by existing SIP tools, with small changes required to allow the vertex compatibility function (encoding labelling and additional degree constraints). These are required as the bigraph variant of SIP can be considered a special case somewhere between induced and non-induced SIP, that is, the use of sites/regions/names swaps the matching semantics from *must have all edges matching* to *must have at least n edges matching*. This encoding is slightly under-constrained, however: although every bigraph matching corresponds to a subgraph isomorphism, some subgraph isomorphisms do not give valid bigraph matchings, and there can be multiple subgraph isomorphisms for a given bigraph matching. Rather

³ Note that this kind of compatibility scheme is more general than the typical “oxygen atoms must be mapped to oxygen atoms, and fluorine atoms must be mapped to fluorine atoms” labelling scheme which occurs in many applications, but nearly all subgraph isomorphism solvers can easily be modified to support this in practice through simple unary constraints.



■ **Figure 3** An illustration of a full encoding of a bigraph matching instance as a SIP instance. Unconnected links are open.



■ **Figure 4** (a) Example target place graph encoding – regions/sites are replaced with unmatchable vertices (shown as R and S but have no label in the encoding). Vertices show control labels; (b) Example pattern place graph encoding – regions/sites removed and degree constraints introduced.

than attempting to handle these details through an awkward encoding, we will instead make use of additional constraints (Section 3.4) and projection nogoods (Section 3.5) on solutions to obtain the desired one-to-one correspondence between solver outputs and solutions.

Importantly there is not a single encoding function between bigraphs and flattened graphs. Instead, to allow matching constraints to be specified, we require different encoding functions for the pattern and target graphs. A diagrammatic overview of the encodings is shown in Figure 3. In the following sections we detail how the encodings are constructed. The encodings are not total but are defined for all cases where matching is non-trivial – for example, they are not designed to be used for node-free bigraphs, i.e. bigraphs containing only hyperedges between names and/or roots/sites for example the identity bigraph.

3.1 Place Graph Encoding

The encodings take bigraphs and produce graphs (V, E) where V is a set of vertices and E a set of edges. Additional constraints are specified with a compatibility function ℓ_p that we define in Section 3.1.1.

The target place graph

Let $T^P = (V_T, ctrl_T, prnt_T) : m \rightarrow n$ be a concrete place graph representing the target of a matching instance. The target place graph encoding function $\phi_T : T^P \mapsto (V, E)$ is shown diagrammatically in Figure 4a. Intuitively, we take the original place graph and extend it with additional vertices for the sites and regions. Formally, the encoding produces a graph with $V = V_T \uplus \{r_i \mid i \in n\} \uplus \{s_i \mid i \in m\}$ and $E = prnt_T^{-1}$ (the child relation). The site/region vertices are added to ensure the *structure* is maintained, i.e. parent/child entities have correct in/out degrees, but regions/sites are never compatible and so do not appear in mappings.

The pattern place graph

Let $P^P = (V_P, ctrl_P, prnt_P) : i \rightarrow j$ be a concrete place graph representing the pattern of a matching instance. The pattern encoding function $\phi_P : P^P \mapsto (V, E)$ is shown diagrammatically in Figure 4b. Intuitively we take the original place graph and remove sites/regions as we do not want to map abstract nodes into the target. The encoding produces a graph with vertices $V = V_P$, edges $E = \{(u, v) \in prnt_P^{-1} \mid v \notin i, u \notin j\}$. As we still need to remember the structure, we replace these with unary in/out degree constraints in the compatibility function (Section 3.1.1). For sites/regions we introduce \geq constraints to allow additional incoming/outgoing edges, while all other entities must match in/out degrees exactly.

3.1.1 Place Compatibility Function

Additional bigraph specific constraints are handled by a place compatibility function $\ell_p : V_G \times V_H \rightarrow \{t, f\}$ that specifies when a pattern-target pair is allowed in a mapping. When defining ℓ_p we assume bigraph definitions such as V_T , $ctrl_P$ and $prnt_T$ are available for pattern/target bigraphs. For clarity we define ℓ_p over *bigraph* nodes, e.g. $u \in V_P$, although formally ℓ_p is over SIP graph vertices (e.g. V_G) and these are inverse-mapped into their bigraph representation for checking compatibility.

We define ℓ_p as the logical conjunction of two sub-functions, i.e. $\ell_p(u, v) = \ell_{p_1}(u, v) \wedge \ell_{p_2}(u, v)$. ℓ_{p_1} ensures the bigraph controls are maintained, while ℓ_{p_2} introduces cardinality constraints based on sites/regions.

$$\ell_{p_1}(u \in V_P, v \in V_T) = \begin{cases} t & \text{if } ctrl_P(u) = ctrl_T(v) \\ f & \text{otherwise} \end{cases}$$

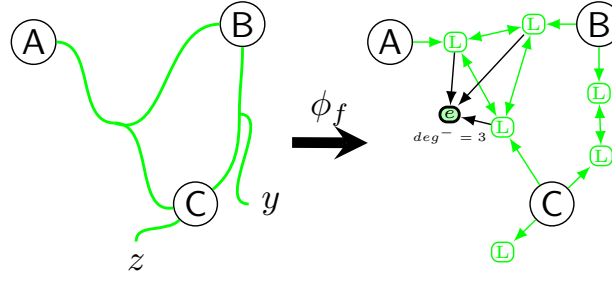
Simply states that entities must maintain their controls.

For ℓ_{p_2} , we assume $\delta^- : V_p \rightarrow \mathbb{N}$ is the function determining the number of in-edges for an entity *ignoring* regions (ordinal n), and likewise $\delta^+ : V_p \rightarrow \mathbb{N}$ as the number of out-edges of an entity *ignoring* sites (ordinal m)⁴.

We then define:

$$\ell_{p_2}(u \in V_P, v \in V_T) = \begin{cases} t & \text{if } prnt(u) \cap n \neq \emptyset \wedge \delta^-(v) \geq \delta^-(u) \\ t & \text{if } prnt(u) \cap n = \emptyset \wedge \delta^-(v) = \delta^-(u) \\ t & \text{if } prnt(u)^{-1} \cap m \neq \emptyset \wedge \delta^+(v) \geq \delta^+(u) \\ t & \text{if } prnt(u)^{-1} \cap m = \emptyset \wedge \delta^+(v) = \delta^+(u) \\ f & \text{otherwise} \end{cases}$$

⁴ As described in Definition 1, ordinals are used to represent the place graph interfaces, such that, for example, $n = 2 = \{0, 1\}$ works as an (ordered) set with two points (roots) that can connect to the wider context.



■ **Figure 5** An example of flattening – links become cliques between port vertices, and closure nodes, shown as solid green, are added for closed links.

Where m and n are the sites/regions of the pattern bigraph. The vertex compatibility function replaces the regions/sites with the semantics of how they should be matched, e.g. that entities connecting sites can have any number of additional children including none, but entities without sites must match out-degrees exactly.

3.2 Link Graph Encoding

Once we have an encoded place graph we *flatten* the hyperedges in the link graph into it so we can treat links as vertices in our encoded graph. The key challenge is allowing non-injective matches for open links to cover the case where, for example, two open links are merged in the context. Unlike the place graph encoding, link graphs are always flattened the same way regardless of whether they are targets or patterns.

Let B^L be a concrete link graph: $(V_B, E_B, ctrl_B, link_B) : X \rightarrow Y$, and $\phi_{\{P,T\}}(D^P) : (V_D, E_D)$ be a (pattern or target) encoding of a place graph D^P . We define the flattening function $\phi_f : \phi_{\{P,T\}}(D^P) \times B^L \mapsto (V, E)$ that given an encoded place graph produces a new flattened graph.

Intuitively flattening creates a new vertex for every *port* (Definition 2) in the link graph, and connects these as a child of their corresponding entities. As arities are fixed, as is the number of port vertices to be added. The vertex compatibility function ensures that port nodes can only match with other port nodes in a subgraph isomorphism. As ports are treated separately to entity nodes, they do not contribute towards the δ^-, δ^+ values in the place graph encoding. Port vertices are wired based on the existing links, i.e. if they shared a link in the bigraph they share a link in the flattened representation. However as graphs do not support hyperedges the edges are encoded by forming a *clique* between the ports.

Converting links to cliques between port vertices is sufficient to encode open links. For closed links we have the following additional constraints:

1. Closed links in the pattern cannot map to open links in the target.
2. Closed to closed link mappings can only be one-to-one and have identical connected sets.
3. Closed links in the target can still be mapped to by many open links.

We implement these constraints by adding additional *closure* vertices e_n for each closed link clique encoding, to represent the “closing off” of these links. The closure vertex is linked to all port nodes in a closed link clique, and we add an equal-degree constraint to each closure node to enforce injectivity, i.e. each closed link in the pattern can only map to a single closed link with an identical adjacency set. The encoding for a link graph featuring both open and closed edges is shown diagrammatically in Figure 5.

Formalising flattening

Given concrete link graph: $B^L : (V_B, E_B, ctrl_B, link_B) : X \rightarrow Y$, and $\phi_{\{P,T\}}(D^P) : (V_D, E_D)$ an encoding of a (pattern or target) place graph D (where $V_B = V_D$), we define $\phi_f : \phi_{\{P,T\}}(D^P) \times B^L \mapsto (V, E)$. We let $\widehat{E}_B = \{e \in E_B \mid link_B(p) = e, p \notin X\}$ be the set of closed links in B^L , and we have $V = V_D \uplus P_B \uplus \widehat{E}_B$, where P_B are the ports of B^L (defined in Definition 2), and one closure node is added for all closed links. We re-use the bigraph *edge* identifier as a vertex identifier in the flattened graph. For edges, $E = E_D \uplus \{(v, p) \mid p = (v, i) \in P_B\} \uplus \{(p_1, p_2) \mid p_1, p_2 \in P_B, link_B(p_1) = link_B(p_2)\} \uplus \{(p, e) \mid e \in \widehat{E}_B, link_B(p) = e\}$. As we build edges with $link_B(p_1) = link_B(p_2)$ we always get two directed edges e.g. (p_1, p_2) and (p_2, p_1) between two linked ports, and the clique structure is constructed automatically through this restriction to binary edges. Additional edges point port vertices to the closure nodes for the closed link representation.

Finally we extend the vertex capability function for place graphs ℓ_p (Section 3.1.1) to include extra link constraints:

$$\ell(u, v) = \ell_p(u, v) \wedge \begin{cases} t & \text{if } u \in P_P \wedge v \in P_T \\ t & \text{if } u \in \widehat{E}_P \wedge v \in \widehat{E}_T \wedge \deg^-(u) = \deg^-(v) \\ f & \text{otherwise} \end{cases}$$

This expresses that ports can only map to ports, and closure nodes can only map to closure nodes with the *exact* same in degree, where \deg^- is the standard in-degree function.

3.3 Encoding Size

A key challenge with the existing SAT solver based algorithm is the large number of clauses required to encode the problem. On the other hand our SIP encoding requires a modest number of nodes and edges, with nodes growing linearly and edges quadratically (due to the clique representation). The number of nodes and edges for a pattern and target bigraph is as follows, where $|e| = |link_G^{-1}(e) \cap P_G|$ is the cardinality of a hyperedge $e \in E_G$ when counting only ports.

Pattern bigraph $P : \langle i, X \rangle \rightarrow \langle j, Y \rangle$

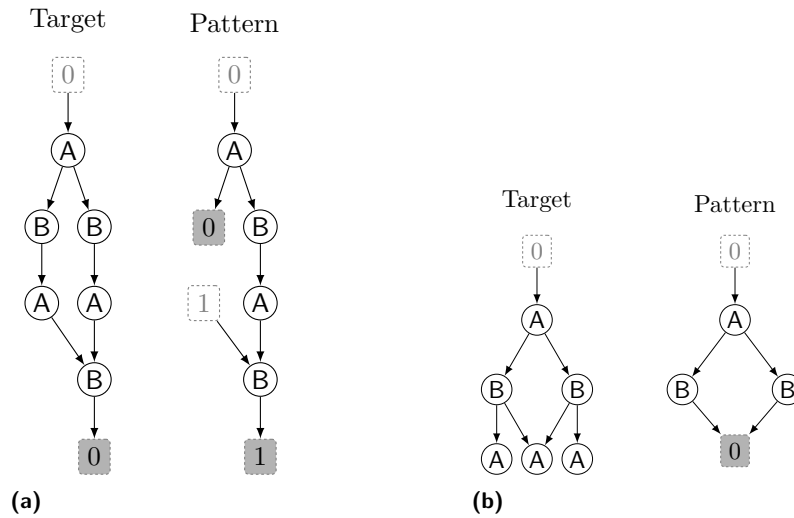
$$\begin{aligned} |V| &= |V_P| + |P_P| + |\widehat{E}_P| \\ |E| &= \sum_{v \in V_P} \delta^-(v) + |P_P| + \sum_{e \in E_P} |e| \cdot (|e| - 1) + \sum_{e \in \widehat{E}_P} |e| \end{aligned}$$

Target bigraph $T : \langle n, X' \rangle \rightarrow \langle m, Y' \rangle$

$$\begin{aligned} |V| &= |V_T| + n + m + |P_T| + |\widehat{E}_T| \\ |E| &= \sum_{v \in V_P} \deg^-(v) + \sum_{s \in n} \deg^-(s) + |P_T| + \sum_{e \in E_T} |e| \cdot (|e| - 1) + \sum_{e \in \widehat{E}_T} |e| \end{aligned}$$

3.4 Checking Constraints

To complete the encoding for standard bigraph matching, we require one additional constraint in the case where a region may have multiple children in the pattern graph – this introduces the rule that all of the child nodes must remain siblings relative to each resultant parent node



■ **Figure 6** (a) Example instance showing need for transitive closure to avoid the parameter also appearing as a context; (b) Example instance with shared site in pattern. No matches are possible in either example.

that substitutes the abstract region during bigraph composition. We cannot validate this during the encoding stage due to the loss of this information from stripping regions in the pattern graph, and as a result the returned solution set may return a superset containing false solutions where the assigned target vertices do not share a parent. An encoding workaround would require the underlying SIP algorithm to be able to support pattern vertices that can encapsulate multiple target nodes in a matching assignment, however this would stray too far from the idea that any existing SIP solver that supports direction and labelling can be used for bigraph matching. We instead deal with this underconstrainedness by implementing a checking constraint on top of the constraint programming model through analysis of the input graphs and their vertex assignments whenever this case occurs – this enforces that a solution is only valid when the set of children of each region all share the same set of parent vertices once mapped to their corresponding target vertices. A similar constraint is also required in the case where we wish to support bigraphs with sharing, where we consider sites with multiple parents rather than regions with multiple children.

Similar constraints are required for the existing SAT encoding to handle the same conditions, so although these are not implemented directly in SIP, evaluating the two approaches against one another remains a fair comparison.

Constraints Imposed By Sharing

Our encoding in addition to the constraint described above is enough to perform matching for Milner’s original bigraph formalism. However, bigraphs with sharing cannot be fully supported⁵ without additional constraints that ensure firstly, that we do not try to form a bigraph where a site also ends up as a region i.e. the DAG property is violated from a cycle being introduced; and secondly that shared sites contain exactly the same elements.

⁵ Sharing within a pattern would be possible, but not on the interfaces.

Figure 6a shows an example where it seems there are valid matches, but sharing leads to an invalid context/parameter. Regardless of which of the two possible matches we choose, the remaining vertices are captured by site 0 in the *parameter*. However, this same path needs to rejoin the pattern through the *context* (region 1). As the same (concrete) bigraph cannot appear in both the context and parameter at the same time, both matches are invalid. To compute this constraint we check that no vertex in the match is *transitively* connected through *prnt* to any vertices in the parameter, that is, you never go upwards to reach the parameter.

Figure 6b shows the second sharing constraint, which is symmetric to the region constraint required for the standard bigraph formalism. As the site is shared it must include *only*, and all, entities shared by the two parents. In this example that means it must contain only a single A entity. This means there is nowhere in the parameter for the additional A children to go and so there is no match.

We implement both these constraints through analysis of the input graphs and their vertex assignments, ensuring all solutions returned are valid. These constraints always hold for standard bigraphs so it is safe to use them in all cases, although as an optimisation we detect sharing and only enable them for bigraphs with sharing.

The ease of implementing sharing constraints demonstrates a further advantage of our SIP encoding over the existing SAT algorithm: we can easily implement further variants of the bigraph matching problem by specifying additional high-level constraints instead of configuring the low-level set of clauses to support new conditions.

3.5 Nogood Recording

When enumerating solutions, the bigraph matching problem does not consider a permutation of open link assignments to be a new separate solution if there already exists a solution with the same place graph vertex and closed link assignments – this will result in the SIP solver returning “duplicate” solutions in its solution set which, whilst still technically valid solutions that differ in their vertex assignments, should not increment the total number of solutions found. We thus make use of the constraint solver’s inbuilt nogood recording functionality where we insert a nogood upon finding a new valid solution, which records the current assignments of the place graph vertices and closure vertices such that any future solutions found with the same set of assignments are disregarded by the solver. This ensures that the set of solutions found by the SIP solver for an encoded bigraph matching instance will always bijectively match the set of solutions found by existing bigraph tools.

4 Implementation and Evaluation

We implemented [3] the encoding and SIP solving process within the Glasgow Subgraph Solver [22] due to it being the state of the art for subgraph solving. However, our approach could be implemented using any solver which supports solution enumeration, directed graphs, and a way of specifying vertex compatibilities.

We compare our SIP implementation to BigraphER’s existing SAT approach on systems with dual Xeon E5-2687A v4 CPUs and 512GB RAM, running Ubuntu 18.04. To allow experimenting with a large number of instances, we perform up to 30 matching problems in parallel on the same machine. The SIP solver is compiled with GCC 7.5 while BigraphER is compiled with OCaml 4.10, statically linked to MiniSAT [13] compiled with GCC 9.3, and

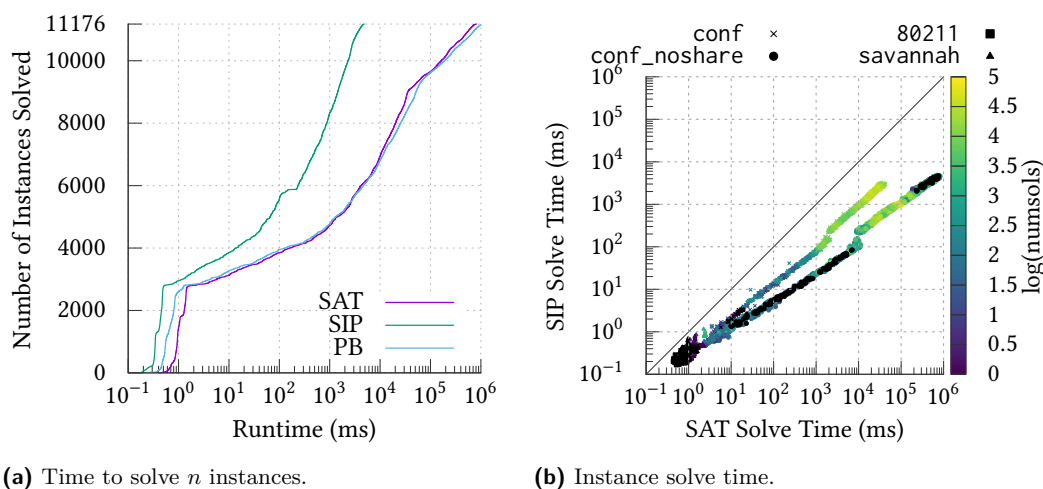


Figure 7 Comparing the performance of the SIP, SAT, and Pseudo-Boolean (PB) approaches. On the left, the cumulative number of instances solved over time for all three approaches. On the right, comparing SAT and SIP on an instance by instance basis; point colour indicates the number of solutions found, and shape the benchmark family.

then copied to the benchmarking machine. For both solvers we measure the steady-clock time required to call the main solver function that includes generating clauses or constraints, but not the time taken to read input files from disk.⁶

As an additional point of comparison we have also implemented a pseudo-Boolean variant of the SAT algorithm, i.e. with direct cardinality constraints when encoding sites. The implementation is through BigraphER with MiniCARD [19] as the underlying solver.

Instances are drawn from two real-world models: `savannah` models a mixed-reality system [9] where children must work together in the physical world to hunt virtual impala, and `80211` that models the 802.11 MAC protocol [11]. In each case instances are recorded from steps to compute the full transition system of the model. Existing public datasets contain only relatively small bigraphs due to limitations of earlier solvers, and so to test scalability we additionally generate larger instances based on the conference call example of Milner [24, chapter 1]. In this case we generate larger instances by not only allowing the existing bigraph to reconfigure, e.g. for agents to join calls, but also rules that add additional agents, computers, rooms and buildings at random. We have two variants of the conference example `conf` and `conf_noshare` that allow/disallow sharing.

Instances use BigraphER’s text based bigraph representation, which is essentially an adjacency matrix with additional entity type and link information. There are 11,176 matching instances in total and we are making these freely available, along with the results in this paper [2], to allow future comparative work. Of the 11,176 instances, 1,660 are unsatisfiable (i.e. no rewriting can be applied). To give us confidence in our implementation, we verified that the SAT and SIP approaches returned the same set of solutions for all of these 11,176 instances.

⁶ In an application context, calls to a solver would be made directly.

4.1 Performance

Figure 7a shows the cumulative number of instances that can be solved *individually* with a timeout of t (x-axis), and so the curve further to the top and left shows the better solver. The horizontal distance between the lines shows the increase in timeout required for both solvers to solve the same number of (but not necessarily the exact same set of) instances, and measures *aggregate speedup* [17]. In this case, no instance takes the SIP solver more than 4,516ms to solve, while the SAT solver requires 820,117ms to solve its hardest instance, giving an aggregate speedup of 181. The pseudo-Boolean solver performs similarly to SAT especially for the harder instances, although there does seem to be a significant benefit for smaller instances. When solving many instances, i.e. when generating a transition system, we expect the pseudo-Boolean solver to be beneficial as it generates less clauses that would need to be garbage collected by OCaml. As expected given the encoding, the number of SAT clauses increases rapidly as the problem instances increase in size.

Figure 7b compares the per-instance runtime for both the SAT and SIP solver, where any point below the mid-line implies SIP outperforms SAT for that instance; the colour of each point gives the log of the number of solutions, with unsat instances shown in black. We see the SIP *always* outperforms the SAT solver for all instances. The consistency of the speedups seen is interesting, especially given the difference in techniques used by the two approaches. Further experiments with non-default configurations of the Glasgow Subgraph Solver show that for bigraph instances, neither neighbourhood degree sequence filtering nor supplemental graph constraints make much of a difference to performance. One might guess that perhaps all instances are computationally easy for any reasonable solver, and that long-running instances are due to either initialisation costs or the cost of enumerating large numbers of solutions. However, things are not this simple: some of the hardest instances are unsatisfiable, take hundreds of thousands of decisions to solve, and spend most of their runtime doing this search. In fact, close inspection of solver statistics suggests that the hard unsatisfiable instances contain very many *near*-solutions, that fail only on the checking constraints discussed in Section 3.4. This suggests that most of the performance gain comes down to the smaller encoding size and faster propagation speeds of the SIP solver, rather than any algorithmic cleverness – although there is still the risk that an insufficiently advanced solver will perform extremely badly on some instances [21]. As future work, we intend to investigate ways of speeding the solver up on these instances, either by using a propagating constraint, or through generation of small conflict clauses.

Finally, although the difference in solve time for the real-world instances, i.e. **savannah** and **80211**, can seem modest, when generating transition systems for verification the solve routine can easily be called thousands of times. As such any speedup is likely to have a high impact on total model generation time.

5 Conclusion

We have shown that the bigraph (with sharing) matching problem can be considered a special case of the subgraph isomorphism problem with additional constraints to handle site/regions, open/closed links and sharing. Through an encoding from bigraphs to graphs with a vertex compatibility function, we can integrate with existing SIP solvers such as the Glasgow Subgraph Solver. We use this to improve on the state of the art SAT-based solver for bigraphs (with sharing), and show significant improvements to performance and scalability including an aggregate speedup of 181 when performance is compared for over 11,000 instances.

Future work

While the new solver already significantly outperforms the SAT approach, there is scope to optimise further, for example using symmetry breaking to reduce the impact of cliques in the encoding, adding further inference that exploits the structure of the labelling function, and using propagation rather than solution checking for sharing.

We expect to see our approach integrated into BigraphER as the new default matching algorithm, allowing a wider range of models to be efficiently manipulated.

Further afield, we wish to extend the algorithm to capture additional bigraph variants – something that was particularly difficult to do through the low-level CNF encoding of SAT. There are many extensions to the bigraph theory, such as local bigraphs [23] that support locality of names, e.g. to model restriction in the π -calculus, and directed bigraphs [16]. Just as sharing introduced a small number of additional constraints, e.g. transitive closure (Section 3.4), we believe that supporting additional bigraph extensions is possible, and requires significantly less effort than the SAT encoding, due to high-level constraint-based reasoning. Once extra variants are supported we will be able to perform a comparative study with existing solvers for these variants (e.g. [12]), to learn from and share new solving techniques.

References

- 1 Faeq Alrimawi, Liliana Pasquale, and Bashar Nuseibeh. On the automated management of security incidents in smart spaces. *IEEE Access*, 7:111513–111527, 2019. doi:10.1109/ACCESS.2019.2934221.
- 2 Blair Archibald, Kyle Burns, Ciaran McCreesh, and Michele Sevegnani. Practical Bigraphs via Subgraph Isomorphism – Benchmark Instances and Results. doi:10.5281/zenodo.4597074.
- 3 Blair Archibald, Kyle Burns, Ciaran McCreesh, and Michele Sevegnani. Practical Bigraphs via Subgraph Isomorphism – Glasgow Subgraph Solver Bigraph Source Code. doi:10.5281/zenodo.5161185.
- 4 Blair Archibald, Muffy Calder, and Michele Sevegnani. Conditional bigraphs. In Fabio Gadducci and Timo Kehrer, editors, *Graph Transformation - 13th International Conference, ICGT 2020*, volume 12150 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2020. doi:10.1007/978-3-030-51372-6_1.
- 5 Blair Archibald, Fraser Dunlop, Ruth Hoffmann, Ciaran McCreesh, Patrick Prosser, and James Trimble. Sequential and parallel solution-biased search for subgraph algorithms. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Proceedings*, 2019. doi:10.1007/978-3-030-19212-9_2.
- 6 Blair Archibald, Min-Zheng Shieh, Yu-Hsuan Hu, Michele Sevegnani, and Yi-Bing Lin. Bi-graphTalk: Verified design of IoT applications. *IEEE Internet Things J.*, 7(4):2955–2967, 2020. doi:10.1109/JIOT.2020.2964026.
- 7 Gilles Audemard, Christophe Lecoutre, Mouny Samy Modeliar, Gilles Goncalves, and Daniel Cosmin Porumbel. Scoring-based neighborhood dominance for the subgraph isomorphism problem. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014*, 2014. doi:10.1007/978-3-319-10428-7_12.
- 8 Giorgio Bacci, Davide Grohmann, and Marino Miculan. DBtk: A toolkit for directed bigraphs. In *Algebra and Coalgebra in Computer Science, Third International Conference, CALCO 2009. Proceedings*, pages 413–422, 2009. doi:10.1007/978-3-642-03741-2_28.
- 9 Steve Benford, Muffy Calder, Tom Rodden, and Michele Sevegnani. On lions, impala, and bigraphs: Modelling interactions in physical/virtual spaces. *ACM Trans. Comput.-Hum. Interact.*, 23(2):9:1–9:56, 2016. doi:10.1145/2882784.

- 10 Muffy Calder, Alexandros Koliouisis, Michele Sevegnani, and Joseph S. Sventek. Real-time verification of wireless home networks using bigraphs with sharing. *Science of Computer Programming*, 80:288–310, 2014. doi:10.1016/j.scico.2013.08.004.
- 11 Muffy Calder and Michele Sevegnani. Modelling IEEE 802.11 CSMA/CA RTS/CTS with stochastic bigraphs with sharing. *Formal Asp. Comput.*, 26(3):537–561, 2014. doi:10.1007/s00165-012-0270-3.
- 12 Alessio Chiapperini, Marino Miculan, and Marco Peressotti. Computing embeddings of directed bigraphs. In Fabio Gadducci and Timo Kehrer, editors, *Graph Transformation - 13th International Conference, ICGT 2020*, volume 12150 of *Lecture Notes in Computer Science*, pages 38–56. Springer, 2020. doi:10.1007/978-3-030-51372-6_3.
- 13 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. doi:10.1007/978-3-540-24605-3_37.
- 14 Amal Gassara, Ismael Bouassida Rodriguez, Mohamed Jmaiel, and Khalil Drira. Executing bigraphical reactive systems. *Discret. Appl. Math.*, 253:73–92, 2019. doi:10.1016/j.dam.2018.07.006.
- 15 Arne John Glenstrup, Troels Christoffer Damgaard, Lars Birkedal, and Espen Højsgaard. An implementation of bigraph matching. Technical Report TR-2010-135, IT University of Copenhagen, 2010.
- 16 Davide Grohmann and Marino Miculan. Directed bigraphs. In Marcelo Fiore, editor, *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics, MFPS 2007*, volume 173 of *Electronic Notes in Theoretical Computer Science*, pages 121–137. Elsevier, 2007. doi:10.1016/j.entcs.2007.02.031.
- 17 Ruth Hoffmann, Ciaran McCreesh, Samba Ndojh Ndiaye, Patrick Prosser, Craig Reilly, Christine Solnon, and James Trimble. Observations from parallelising three maximum common (connected) subgraph algorithms. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th International Conference, CPAIOR 2018*, pages 298–315, 2018. doi:10.1007/978-3-319-93031-2_22.
- 18 Jean Krivine, Robin Milner, and Angelo Troina. Stochastic bigraphs. *Electr. Notes Theor. Comput. Sci.*, 218:73–96, 2008. doi:10.1016/j.entcs.2008.10.006.
- 19 Mark H. Liffiton and Jordyn C. Maglalang. A cardinality solver: More expressive constraints for free - (poster presentation). In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 485–486. Springer, 2012. doi:10.1007/978-3-642-31612-8_47.
- 20 Ciaran McCreesh and Patrick Prosser. A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Proceedings*, 2015. doi:10.1007/978-3-319-23219-5_21.
- 21 Ciaran McCreesh, Patrick Prosser, Christine Solnon, and James Trimble. When subgraph isomorphism is really hard, and why this matters for graph databases. *J. Artif. Intell. Res.*, 61:723–759, 2018. doi:10.1613/jair.5768.
- 22 Ciaran McCreesh, Patrick Prosser, and James Trimble. The Glasgow Subgraph Solver: Using constraint programming to tackle hard subgraph isomorphism problem variants. In *Graph Transformation - 13th International Conference, ICGT 2020*, pages 316–324, 2020. doi:10.1007/978-3-030-51372-6_19.
- 23 Robin Milner. Local bigraphs and confluence: Two conjectures: (extended abstract). *Electr. Notes Theor. Comput. Sci.*, 175(3):65–73, 2007. doi:10.1016/j.entcs.2006.07.035.
- 24 Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

- 25 Michele Sevegnani and Muffy Calder. Bigraphs with sharing. *Theor. Comput. Sci.*, 577:43–73, 2015. doi:10.1016/j.tcs.2015.02.011.
- 26 Michele Sevegnani and Muffy Calder. BigraphER: Rewriting and analysis engine for bigraphs. In *Computer Aided Verification - 28th International Conference, CAV 2016. Proceedings, Part II*, pages 494–501, 2016. doi:10.1007/978-3-319-41540-6_27.
- 27 Michele Sevegnani, Milan Kabác, Muffy Calder, and Julie A. McCann. Modelling and verification of large-scale sensor network infrastructures. In *23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018*, pages 71–81. IEEE Computer Society, 2018. doi:10.1109/ICECCS2018.2018.00016.
- 28 Christine Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artif. Intell.*, 174(12-13):850–864, 2010. doi:10.1016/j.artint.2010.05.002.
- 29 Stéphane Zampelli, Yves Deville, and Christine Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 15(3):327–353, 2010. doi:10.1007/s10601-009-9074-3.

The Hybrid Flexible Flowshop with Transportation Times

Eddie Armstrong

Johnson & Johnson Research Centre, Limerick, Ireland

Michele Garraffa ✉

Confirm SFI Research Centre for Smart Manufacturing, Limerick, Ireland
School of Computer Science, University College Cork, Ireland

Barry O’Sullivan ✉

Confirm SFI Research Centre for Smart Manufacturing, Limerick, Ireland
School of Computer Science, University College Cork, Ireland

Helmut Simonis ✉

Confirm SFI Research Centre for Smart Manufacturing, Limerick, Ireland
School of Computer Science, University College Cork, Ireland

Abstract

This paper presents the hybrid, flexible flowshop problem with transportation times between stages, which is an extension of an existing scheduling problem that is well-studied in the literature. We explore different models for the problem with Constraint Programming, MILP, and local search, and compare them on generated benchmark problems that reflect the problem of the industrial partner. We then study two different factory layout design problems, and use the optimization tool to understand the impact of the design choices on the solution quality.

2012 ACM Subject Classification Software and its engineering → Constraints; Computing methodologies → Planning and scheduling

Keywords and phrases Constraint Programming, scheduling, hybrid flowshop

Digital Object Identifier 10.4230/LIPIcs.CP.2021.16

Supplementary Material *Dataset:* <https://zenodo.org/record/5168966>

Funding This project has received funding from a research grant from Science Foundation Ireland (SFI) under Grant Number 16/RC/3918.

1 Introduction

An important category of production systems comprises hybrid flowshop environments, where the input jobs need to be processed by different production stages, and multiple identical machines are available at each stage. In this context, the problem consisting of finding schedules of minimum length is a well-known NP-hard combinatorial problem, known as Hybrid Flowshop Scheduling (HFS) [36]. An extension of this problem, which has been the object of recent studies, is the case where a subset of the jobs are required to skip some of the stages. This problem is usually denoted as Hybrid Flexible Flowshop Scheduling (HFF) [33]. Since many decades, the solution of HFS/HFFs problems has been fundamental to increase efficiency of many real world manufacturing systems, related to a wide variety of production areas like electronics [25, 10], glass [32], textile [22], packaging [1], etc.

Recently, an increasing number of production systems are structured such that the machines are located in different buildings of the same production site, or even at separate sites. This may occur for multiple reasons, such as the risk of contamination of materials during some production stages. As a consequence, the machines may be located at diverse locations, requiring a non-negligible time to transport a job from one stage to the next



© Eddie Armstrong, Michele Garraffa, Barry O’Sullivan, and Helmut Simonis;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

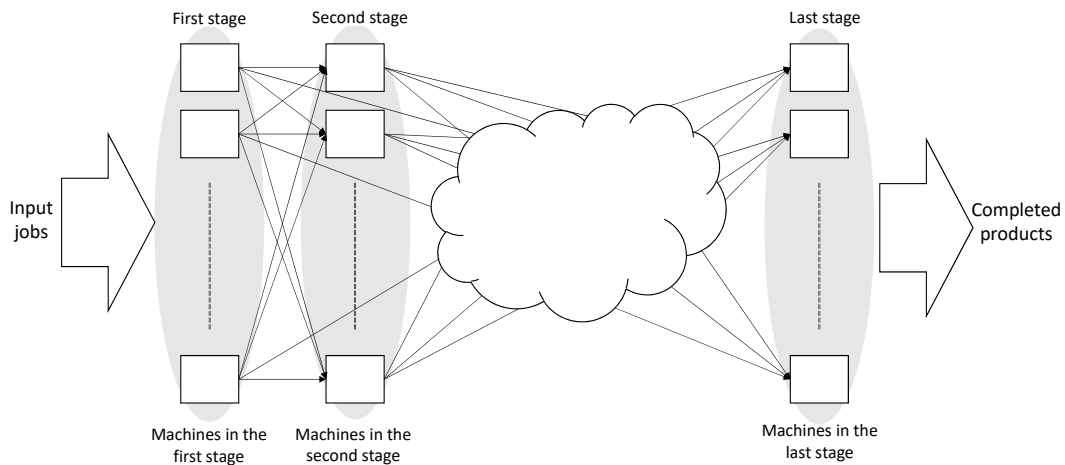
Editor: Laurent D. Michel; Article No. 16; pp. 16:1–16:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

16:2 The HFP with Transportation Times



■ **Figure 1** Logical structure of a hybrid flowshop.

one. These transportation times are known as inter-stage transportation times, in order to make a distinction with the ones required to deliver the products or acquire raw materials. Moreover, these transportation times are usually machine-dependent, in the sense that they are proportional to the physical distance between the two machines involved. In this work, we introduce the Hybrid Flexible Flowshop with Transportation Times (HFFTT), as an extension of the HFF where inter-stage, machine-dependent transportation times are taken into account.

The diagram depicted in Figure 1 summarizes the logical structure of a HFFTT problem, where multiple machines are associated with each stage and each arrow connecting a couple of machines (m_a, m_b) is indicating the fact that a certain time is needed to transport a job from m_a to m_b . The main characteristics of the HFFTT are:

- An arbitrary number of production stages are considered.
- Each stage has multiple identical machines working in parallel.
- All jobs are following the same production flow, with the exception that some jobs may skip some stages.
- Each job has a fixed processing time for each production stage, independent of the machine used.
- Transportation times are required to move a job from one machine of a stage to another machine of the following stage.
- The vehicles transporting the jobs are not taken into consideration, since they are assumed to be enough to avoid any waiting delay. This is justified by the fact that current production lines are fully automated and the schedules are defined a priori, so that the transporters can move the jobs in time with no delays.
- The objective function value is the maximum completion time, i.e. the makespan of the schedule. The makespan is a measure of the system efficiency, it is considered in the standard definition of the HFS and widely used in scheduling.

Other assumptions are that any machine can process only one operation at a time with no preemption, while all jobs can be processed by only one machine at a time. All tasks are released at time 0, setup times are negligible or can be aggregated to the jobs' processing times. The capacity of the buffers between stages is unlimited, there are no additional resources needed to run a job on a machine and, finally, the processing times and the transportation times are deterministic parameters that are known a priori.

Surprisingly, there are no previous studies focusing exactly on this problem definition, to the best of the authors' knowledge. However, studies tackling similar problems are discussed in Section 2. The HFFTT can be indicated with $HFFm|tt|C_{max}$ by means of the three field notation, which is the most common notation used to identify scheduling problems [21].

Our motivation for studying the HFFTT lies in a real-world practical application arising in fully-automated pharmaceutical manufacturing, where millions of lots are produced every year. One of the authors, working for the research and development team of the firm, provided requirements that we reproduced when we defined the problem and generated our instances. The aim of this work is twofold. On the one hand, increasing the throughput of a production site by a small percentage (say 1%) brings a significant economic benefit, due to the very large volume of production. We formulate the problem in terms of Constraint Programming (CP), Mixed Integer Linear Programming (MILP), and Local Search in order to develop optimization approaches based on off-the-shelf solvers and test them on these realistic scenarios.

On the other hand, the introduction of transportation times leads to another research question, which is of particular interest for the industry partner. It consists of determining the positions to be assigned to the machines in order to achieve a high system efficiency. Such problem belongs to a wide category of optimization problems known as facility layout [14, 4], with hundreds of papers published in recent years [41]. We address this challenge by using CP-based scheduling to quantify the performances of different layouts alternatives, hence providing insights for driving business-related decisions in the real world scenario.

The rest of this document is arranged as follows. Section 2 collocates the problem in the state of the art, by analyzing similar scheduling problems studied in the past.

The problem admits multiple alternative formulations, which are presented and discussed in Section 3. Section 3.2 presents a CP formulation for the HFFTT. Section 3.3 discusses how to extend the MILP formulations available in the state of the art for the HFF, in order to take into account transportation times. Section 3.4 provides a local search procedure, which is used to compute good feasible solutions and initialize the solvers. Section 4 describes the computational experiments that we conducted. These experiments are performed on realistic instances, whose structure comes from the real world scenarios discussed with the pharmaceutical firm, as mentioned before. The instance generation and the scenarios are discussed in Section 4.1 and Section 4.2, respectively. The computational results, describing the performance of the proposed models, are presented in Section 4.3. Section 5 concludes the paper.

2 Related Work

As discussed in the introduction, the HFFTT is an extension of the HFF, which is a HFS problem where certain jobs are required to skip some production stages. The HFS is a well-studied problem, whose literature includes many problem variants. It is NP-hard in its general form and in most of its restrictions. As an example, the case where there is one machine per stage is a flowshop problem, which is known to be NP-hard [20]. Some polynomially solvable special cases are arising when some special properties and precedence relationships are verified [13].

The reader is referred to [36] for an in-depth survey, which describes papers published up to 2010. Another, more recent review [40] is covering the works published from 2010 to 2020, with a special focus on metaheuristic approaches. The literature of the HFF is less rich with respect to the HFS. Four different MILP formulations are proposed in [33]

and assessed computationally. A general discussion about the impact of including different realistic constraints is given in [37]. The most recent metaheuristic developed for the HFF is an iterated greedy approach based on a hyperheuristic [8]. Finally, the use of CP approaches in HFS/HFF problems is tackled by [26] and [45]. [26] considers a HFS with multiprocessor tasks and proposes a memetic algorithm, which uses a CP-based branch and bound algorithm as a local search engine. [45] studies a HFF problem inspired by a real-world application, formulates it in terms of CP and solves it utilizing Gecode.

The HFFTT is more general than the HFF and the HFS, hence it is NP-Hard as its special cases. The HFFTT includes inter-stage, machine-dependent transportation times, but they are not the only type of transportation times studied by the scheduling community. In fact, the first scheduling paper taking transportation into account is [31], which dates back to 1980, and considers purely job-dependent transportation times. The authors consider a two-machine flowshop problem with a sufficient number of transporters, so that whenever a job is completed on the first machine it can be transported to the second machine immediately. Another paper [29] focuses purely on the job transportation in a HFS with two stages. Other HFSs with some form of job transportation have been considered in other works, the reader is referred to Section 2 of [30] for a recent and detailed review about this topic. A recent paper [44] studies two problems where there is only one transporter in a HFS with two stages. These problems are very close to the HFFTT, with the two main differences being the fact that the transporter is seen as a further resource to be shared among the jobs and the number of stages is restricted to 2. Another similar problem to the one defined in this paper is presented in [30]. The main differences are that buffers are assumed to be finite, there is only one transporter per stage, and the jobs are not skipping any stage. [42] deals with another HFS with transportation times, differing from our problem because it considers a limited number of transporters with finite capacities and because the transportation times are purely stage-dependent, since they do not depend on the machine positions. [15] studies a single-transporter HFS with machine eligibility where the transportation times are not machine-dependent. The same authors propose in [16] and [17] a MILP model and a metaheuristic for a cyclic HFS, with multiple robots devoted to perform the transportation operations. The main difference with the HFFTT is the fact that they deal with a cyclic problem and they use as an objective a throughput rate maximization, that is the number of parts completed in the cycle time. [24] presents a two-centers hybrid flowshop, where the machines of each stage are aggregated in centers. Contrarily to our problem, here only two stages are considered and the transportation times are purely job-dependent.

Other recent works are dealing with scheduling problems with transportation times, which consider different types of production environment, such as a groupshop [3], a flexible jobshop [35] and a batch-flowshop [6].

A large number of studies have been devoted to the design of facility layouts in order to improve the system productivity and/or reduce costs. However, we did not find any study related to designing facility layouts for HFS. A survey about facility layout problems is provided in [4], where a survey of the mathematical optimization approaches is presented. These approaches are based on solving a variety of optimization problems, that are expressed in terms of MILP or conic programming. In some cases, simulation and optimization interoperate to find the best layouts. [5] presents a simulation-based genetic algorithm encoding each single layout as a string and evaluating the performances of each layout by means of simulation. [19] discusses the case where simulation-based optimization is used for facility layout optimization under uncertainty. CP has been adopted to solve different types of layout problems. An example is [39], where the authors show the effectiveness of CP for

designing manufacturing cells. Another related study is [43], where the authors propose a CP approach for multi-objective wind farm layout optimization. To the best of the authors’ knowledge, there are no previous studies using constraint-based scheduling for the design of facility layouts.

3 Models

We now present different models for the selected problem, using various solver technologies. We only explain one CP model in full detail due to space considerations.

3.1 Notation

In this section, we introduce the notation used in the different problem formulations. Let J be a set of jobs, M be a set of machines and S be a set of production stages. Each job $j \in J$ requires a processing time $p_{j,s}$ to complete a stage $s \in S$. The set of all jobs that need to be processed in stage $s \in S$ is indicated with $J_s \subseteq J$, while the stage before $s \in S$ where a job $j \in J$ needs to be processed is indicated with $prec(s, j)$. The transportation time $\delta_{m,m'}$ is required to move a job from machine $m \in M_s$ to $m' \in M_{s'}$, where $s, s' \in S$ and s' is successive to s in the production system. By following the general concept used in CP for scheduling problem including the HFS [44], we define a task as the execution of a job at some production stage. Hence, one task per stage used is associated to each job. We indicate the set of all tasks with T . Finally, the set of all the tasks associated with a certain stage $s \in S$ is denoted as T_s and the set of all tasks associated with a certain job $j \in J$ is denoted as T_j .

3.2 CP

For this type of scheduling problem with precedence and machine choice there are two main choices of modelling in Constraint Programming. The first approach uses a two-dimensional `diffn` [7] constraint, where the first (x) dimension represents time, and the second (y) dimension represents the machine allocated to a task. Each task is represented by a rectangle whose width is the duration of the task, and the height is one, while the x start position represents the start time, and the y value indicates the machines on which it is run. This constraint states that tasks are either scheduled on different machines, or do not overlap in time. This type of model goes back to CHIP [12], and can be expressed in many CP systems, we use it in our MiniZinc [34] and SICStus Prolog [9] formulations. The transport time between two consecutive tasks of the same job is expressed by a `table` constraint, which links the machine variables of the tasks to the transport time, by enumerating all possible combinations. This provides maximum flexibility in expressing the transport time constraint, while allowing for efficient, domain consistent constraint propagation.

The second approach uses optional interval task variables [28] to represent the machine choices. For each task and each machine on which the task can be scheduled, there is an optional interval variable which represents the choice whether the task is run on this machine. Exactly one of the optional variables for a task must be selected (using an `alternative` constraint), while all optional tasks on the same machine are constrained to be non-overlapping. This model is required by IBM’s CPOptimizer [28], but can also be expressed with MiniZinc ¹. Expressing the transport time between tasks is more complex,

¹ Unfortunately, the MiniZinc model with interval variables is not competitive solving even the smallest problem instances, and is not used in the experiments.

IBM provides a specific constraint to deal with sequence dependent setup times on the same machine, but not for expressing transport times between consecutive tasks of the same job on different machines. We have to link additional machine assignment variables via `presenceOf` constraints to the optional tasks, and use `allowedAssignments` constraints to handle the transport times, linking the machine assignment variables and the transport time of consecutive tasks. The default search in CPOptimizer automatically switches between different search methods during the search phase, and is, like our custom SICStus Prolog search described below, independent of the selected time resolution.

3.2.1 Diffn Model

We now describe the first constraint model of the problem in full detail, using MiniZinc as the description language. We first define the constants to use, and the arrays that hold the input data. The set T is the index-set of all tasks, for every task we have the fixed duration, the stage to which it belongs, and the set of machines on which it can run. We also have a set P of precedence relations between tasks that will be used to set up precedence constraints. Note that this formulation is more generic than the flowshop problem, and allows to handle other generic scheduling problems as well.

```

1 include "globals.mzn";
2 % constant definitions
3 set of int:T; % tasks
4 set of int:P; % precedences between tasks
5 set of int:M; % machines
6 set of int:S; % stages
7 set of int:tuples; % transport time table size
8 int:lb; % lower bound on the cost
9 int:ub; % upper bound on the cost
10 % constant arrays
11 array[T] of int:duration;
12 array[T] of int:stage;
13 array[P,1..2] of int:prec; % precedence pairs
14 array[T] of set of int:machines; % machine domains
15 array[tuples,1..3] of int:transportTime;
16 % data file
17 include "data/data.dzn";

```

For every task, we have three variables, the start of the task, the machine on which it is run, and the travel time from the machine of the previous task to its own machine. In addition, we use the variable *cmax* as the overall cost.

```

1 % variables
2 array[T] of var 0..ub:start;
3 array[T] of var M:machine;
4 array[T] of var 0..ub:travel;
5 var lb..ub:cmax;

```

We have five constraint types in the model. The first links the makespan variable to the end of all tasks. The second handles the `precedence` constraints within a job, linking the first to the second task by the duration of the first task, and the required travel time between the machines used. The third constraint expresses that each task can only be run on a specific subset of the machines. The fourth constraint is the `diffn` constraint for each stage, representing the tasks as rectangles. Note that a single `diffn` constraint for all tasks would suffice, but multiple constraints with disjoint subsets of tasks lead to better performance. In

addition, a cumulative [2] constraint linking the tasks and the number of available machines of each stage may lead to additional propagation in some systems. The final constraint generates the `table` constraint for the travel times, linking the machines assigned and the time required, via a single table of all allowed travel time combinations.

```

1 % constraints
2 constraint forall (i in T)
3   (cmax >= start[i]+duration[i]);
4 constraint forall (p in P)
5   (start[prec[p,2]] >= start[prec[p,1]]+
6     duration[prec[p,1]]+
7     travel[prec[p,2]]);
8 constraint forall(i in T) (machine[i] in machines[i]);
9 constraint forall(s in S)
10  (diffn([start[i]|i in T where stage[i]=s],
11         [machine[i]|i in T where stage[i] = s],
12         [duration[i]|i in T where stage[i] = s],
13         [1|i in T where stage[i] = s]));
14 constraint forall (p in P)
15   (table([machine[prec[p,1]],
16         machine[prec[p,2]],
17         travel[prec[p,2]]],transportTime));
18 % solve
19 solve minimize cmax;

```

With the free search of Chuffed [11] we find solutions for smaller problem sizes, but it initially finds solutions with high costs. When giving a conservative value for *ub*, Chuffed spends most of its time decreasing the cost value one by one from that initial high value. We need an alternative to find good solutions quickly, the strategy chosen will assign tasks from left to right, fixing start and machine of the task together. Note that other strategies, like assigning all start values before the machine assignment or vice versa, do not lead to viable solutions, as the machine assignment determines the transport time between tasks, and therefore has a direct effect on the start times. Conversely, if we fix the start times to their smallest value, we remove too much flexibility in the machine assignment to complete the assignment.

We use the `priority_search` annotation of Chuffed [18] to assign start and machine together. We select a task with the smallest value in the domain of its start, and fix the start and then the machine of the task to values in their domain. Note that the sequence here is important to place tasks to their left-most position.

```

1 include "chuffed.mzn";
2
3 solve ::priority_search(start,
4   [int_search([start[i],machine[i]],input_order,indomain_min)| i
5     in T],
6   smallest,complete) minimize cmax;

```

For the search in the SICStus Prolog [9] model, which uses the same constraints, we define a custom search routine that iteratively increases the start time considered, and for each time point, decides if the remaining tasks should start at this time or not. Once the start is fixed, we try to find a machine on which the task will be run. The search times out after a limited amount of backtracking, and restarts with a different initial task ordering, until either the lower bound is reached, or the global time budget is exceeded.

3.3 MILP

The HFFTT admits multiple MILP formulations. It is possible to extend the HFF formulations in [33] so that they include inter-stage transportation times. In fact, Model 1, Model 3 and Model 4 of [33] can be extended by including continuous time variables $\Delta_{j,s} \in \mathbb{R}_+$, representing the transportation time required by job $j \in J$ to reach stage $s \in S$. Analogously, Model 2 of [33] can be adapted by including continuous time variables $\Delta_{j,s,m} \in \mathbb{R}_+$, representing the transportation time required by a job $j \in J$ to reach a machine $m \in M$, belonging to stage $s \in S$. The variables $\Delta_{j,s}$ and $\Delta_{j,s,m}$ need to be linked with the real transportation times $\delta_{m,m'}$. This can be done by imposing that they are greater than or equal to $\delta_{m,m'}$, in the case the job $j \in J$ is processed by the machines $m, m' \in M$, which is determined by the binary variables related to the machines' assignments. Finally, the constraints linking the jobs' start and completion times at the different stages need to be modified in order to include the transportation times.

We implemented all four adapted models and performed a set of preliminary tests, where Model 4 yielded the best results. Hence, we focus on the modified version of Model 4. To simplify notation, let us define a dummy stage s_0 preceding all other stages, with no machines $M_{s_0} = \emptyset$ and performed by all the jobs with 0 duration. The decision variables are:

- $x_{i,j,s} \in \{0, 1\}$ equal to 1 if job $i \in J_s$ comes after job $j \in J_s$ at a certain stage $s \in S$, 0 otherwise.
- $y_{j,s,m} \in \{0, 1\}$ equal to 1 if the job $j \in J_s$ is processed by machine $m \in M_s$ at stage $s \in S \cup \{s_0\}$, 0 otherwise.
- $\Delta_{j,s} \in \mathbb{R}_+$ is the inter-stage transportation time required by job $j \in J_s$ to reach stage $s \in S$ from the previous one.
- $C_{j,s} \in \mathbb{R}_+$ is the completion time of job $j \in J_s$ in stage $s \in S \cup \{s_0\}$.
- $C_{max} \in \mathbb{R}_+$ represents the makespan.

The following formulation of the HFFTT holds:

$$\min C_{max} \tag{1}$$

subject to:

$$C_{j,s_0} = 0 \quad \forall j \in J \tag{2}$$

$$y_{j,s_0,m} = 0 \quad \forall j \in J, m \in M \tag{3}$$

$$\sum_{m \in M_s} y_{j,s,m} = 1 \quad \forall j \in J_s, s \in S \tag{4}$$

$$C_{j,s} \geq C_{j,prec(s,j)} + p_{j,s} + \Delta_{j,s} \quad \forall j \in J_s, s \in S \tag{5}$$

$$C_{j,s} \geq C_{i,s} + p_{j,s} - \Lambda(3 - x_{j,i,s} - y_{j,s,m} - y_{i,s,m}) \quad \forall i \neq j \in J_s, s \in S, m \in M_s \tag{6}$$

$$C_{i,s} \geq C_{j,s} + p_{i,s} - \Lambda x_{j,i,s} - \Lambda(2 - y_{j,s,m} - y_{i,s,m}) \quad \forall i \neq j \in J_s, s \in S, m \in M_s \tag{7}$$

$$C_{max} \geq C_{j,s} \quad \forall j \in J_s, s \in S \tag{8}$$

$$\Delta_{j,s} \geq \delta_{m',m}(y_{j,s,m} + y_{j,prec(s,j),m'} - 1) \quad \forall m' \in M_{prec(s,j)}, m \in M_s, j \in J_s, s \in S \tag{9}$$

Please note that the parameter $\Lambda \in \mathbb{R}_+$ indicates a large enough real constant. Furthermore, $prec(s, j)$ indicates the stage where $j \in J$ was processed before stage $s \in S$. The objective is to minimize the makespan (1). Constraints (2) and (3) are related to the dummy stage, which is the first stage for all the jobs and requires no time for its completion and no machine usage. Constraints (4) ensure each job $j \in J_s$ is processed by exactly one machine at each stage $s \in S$. Constraints (5) link the completion time of a job in a stage with the

completion time of the same job in the previous stage. Please note that such constraints reduce to $C_{j,s} \geq p_{j,s}$ in the case where $\text{prec}(s, j) = s_0$, since $C_{j,s_0} = 0$ and no time is required to reach the first stage after s_0 ($\Delta_{j,s} = 0$). Constraints (6) and (7) determine the completion times of the jobs in a stage, by taking into account the precedence and the machines availability. Constraints (7) enforce that C_{max} is greater than or equal to the completion time of all the jobs at any stage. Constraints (8) link the values of the transportation time variables with the time needed to transport each job in order to achieve a machine at a certain stage. Please note that $\Delta_{j,s}$ is not involved in any constraint if $\text{prec}(s, j) = s_0$, since $M_{s_0} = \emptyset$.

3.4 Dispatch Rule and Local Search

In order to evaluate the results of our models in a broader context, we also introduced two incomplete methods of finding solutions. The Dispatch model takes the jobs in some given order, and schedules each job in sequence, placing the tasks at the first available time, on the first available machine. This can be done efficiently by keeping track of the allocated time on each machine, and by considering the required precedences and travel time when looking for the next available machine. In our system, we randomly permute the job order, and rerun the Dispatch model repeatedly, keeping improved solutions. We stop the search when we either reach the given lower bound, or hit a timeout limit.

We've added a Local Search variant of this model, by allowing swaps of two jobs and insertion of jobs into the job sequence at a different place. When we run out of possible moves, we restart the search with a new permuted job order, and continue until we reach the lower bound or the time limit.

Note that neither variant uses constraints, or estimates about the achievable makespan given a partial solution, while both rely on the simplicity of the method to evaluate many possible job sequences.

3.5 Lower Bound Calculation

A good lower bound on the makespan can be useful to stop the search having found an optimal solution reaching that lower bound, or to understand the remaining optimality gap of the instances tested. We have extended the bounds in [23] to deal with the transportation time between stages, while adding one more bound based on the job type distribution. The following gives an intuition of the bounds used.

An obvious lower bound on the makespan is the minimum duration of any job to be scheduled. This consists of the sum of the duration of the tasks of the different stages, and an estimate of the transport time required between consecutive tasks. The easiest approximation of this is using the smallest transport time between each pair of stages as an estimate, a more complex model can use Dynamic Programming to find the shortest path from start to finish over all machines. This job related lower bound is quite strong if the number of jobs is small, but will become useless as the number of jobs in the schedule increases. Note that the constraint models will infer the first version of this lower bound from the precedence and travel time constraints, this then becomes the initial lower bound of the makespan variable.

For larger instances we can do better than this, by considering a stage based bound. If we consider any set of tasks belonging to the same stage, we can compute a lower bound on the overall makespan as the minimum time to start one of the tasks of the set (**minStart**), the time to process all tasks in the set on the machines for that stage, and the minimum time that is required from the end of a task in the set to the overall project end (**minEnd**).

16:10 The HFP with Transportation Times

While it is clearly not possible to evaluate this bound for every subset of the tasks for each of the stages, we can group tasks by their `minStart` and `minEnd` values.

We still have to define how to estimate the time required to process all tasks of a set on the machines of the selected stage. Ideally, we can solve a separate bin-packing constraint to find the best solution, but that itself is a hard problem, and therefore not really appropriate for a lower bound. We use three lower bounds on the bin packing problem instead:

- The maximum duration of any task in the set is a bound on the time required to process all tasks of the set.
- The total duration of all tasks in the set, divided by the number of available machines, is another lower bound.
- If we order the tasks of the stage by decreasing duration, then the sum of the k th and $k + 1$ th element of the task list is a lower bound on the total duration, if the stage has k machines.

Finally, we consider the most common product types, for which there are multiple jobs in the order set. When minimizing the overall makespan, these jobs are identical, and in our test scenarios, quite common. The most common product accounts for up to 25% of all orders. It is worthwhile to obtain a lower bound for these jobs. We consider the sub problem of scheduling only a single product type, in that case we do not have to allow all permutations of the jobs, since they are all identical. The solution to this simpler problem is a valid lower bound for the overall makespan. In the selected scenarios, this bound very rarely dominates the other bounds, but it becomes useful when the power-law distribution of product types is changed to consider fewer, more common products.

Given these bounds, we still observe rather wide optimality gaps (up to 20%) for some medium sized instances. Further improvements could consider:

- The total surface estimate assumes that there are no other tasks that overlap the selected period, but there may be longer tasks starting earlier that are still active at this time. A study of energetic reasoning for the cumulative constraint [2] may be appropriate to understand if we can apply some of the reasoning here.
- Some tasks may be required to achieve the shortest `minStart` and shortest `minEnd` time at the same time. But it may not be possible to use the same tasks for both roles, so that more time either at the start or the end is required.
- There can be a conflict between lower bounds for different stages, which require different sets of jobs to be scheduled early to avoid losing too much production capacity at the start or end of the schedule. By considering multiple, non-consecutive bottleneck stages at the same time we may be able to improve the lower bounds significantly, without having to solve hard combinatorial sub-problems to optimality.

4 Computational Experiments

In this section we describe how we generate sample problem instances based on assumptions on the manufacturing process, which design alternatives we want to explore, and which results we achieve. The results allow us to understand the impact of the different models and solving technologies, and how the factory design choices affect the overall solution quality for different solvers. A final step, not presented here, then integrates our results with a financial analysis of the costs and risks of the design alternatives, to help the decision making process for the industrial partner.

4.1 Instance Generation

We have built an instance generator for the problem, which produces fully parametrized instances of the problem. The generated test cases will be made available as an appendix in the final publication. In our experiments we selected some parameters to be typical of the situation of the industrial partner, while allowing for flexibility in other parameters. All experiments shown use 8 stages, where stage 4 and 8 can be skipped with 10% probability. We allow 10 machines at each stage, and consider two scenarios, one, where each stage has the same number of machines, and one where the number of machines per stage is adjusted downwards based on the total workload for each stage. For space reasons we only report results for the uniform machine number case. We vary the number of jobs between 20 and 400, noticing that the larger instances may require large amounts of memory for some solvers.

For machines within the same building, we consider a lane model (see Figure 2a), where machines are arranged in a grid pattern, all machines of the same stage being placed in the same column, and machines in the same row requiring the smallest transport time between them. Transport time increases with the difference of the lanes in which the machines are placed. Note that this layout design is only one of the choices in the instance generator, and does not directly affect the models, which use a table of transport times between machines.

Each job belongs to a given job type, jobs of the same type have the same manufacturing constraints. We select the job type based on a discrete power law (Zipf) distribution, with exponent 1.05. This means that some products are more common than others, but many products only have a single job in the order set. This choice corresponds to the semi-custom production model for which the factory is being designed. The task duration is randomly chosen within a range of 1 to 10, while the transport times vary from 1 to 9, and an inter-building transport requires 10 time units. The instances generated are available at [38].

4.2 Scenarios

We use our models to answer two design questions, one designing the transport inside a single factory, the other dealing with the potential split of the production between sites.

4.2.1 Reach of Transport between Stages

This problem considers the lane based layout inside a single factory as a starting point. The question is how far the transport should reach between lanes. If each job stays within its initial lane during production, then transport requirements are minimal, but there is little flexibility in the scheduling. If, on the other hand, we allow transport between any two lanes, a lot of infrastructure needs to be provided, which may or may not pay off in improved solution quality. The result will be an understanding of how transport flexibility affects solution quality, and if using different solvers for the experiments lead to different results.

4.2.2 Single Factory vs. Multiple Locations

Another important design question is whether it is better to have a single facility where production is concentrated, or whether (for example) two locations should be selected. We consider five alternatives:

1. We use a single facility where all products are made. Transport is between lanes of consecutive stages of production within the location.

2. Consider two facilities that run sequentially, all production of the early stages is done in one location, the intermediate products are then moved to another location, where the later stages of production are performed.
3. We use two facilities which run in parallel in relatively close proximity, each handling part of the workload. Intermediate products can be moved between facilities at a (high) inter-building transport cost.
4. Take two facilities in different geographical regions, with no intermediate product movement between them. For each order, the choice of facility where it is manufactured is left to the scheduling algorithm.
5. We choose the layout as in the previous case, but 80% of the orders are preassigned to a facility due to customer location, only 20% of the orders can be assigned freely.

4.3 Results

We now present some results on the solution quality obtained with the different models and solvers.

4.3.1 Selected Solvers

We compare the results for the following solver alternatives, all running on a Windows 10 laptop with i7-6920HQ CPU running at 2.90GHz, and 64 GB of memory. All solvers are using a single core, to ease comparison. CPOptimizer and Cplex allow to use multiple cores, while Chuffed and SICStus do not provide this functionality out of the box. Both MiniZinc and SICStus solutions are run as separate solver processes from the main Java application.

CP Optimizer by IBM, Version 20.1.0, task interval model, default search

SICStus Prolog Version 4.3.5, diffn model, with a custom search routine

MiniZinc Chuffed, Free Search Version 2.5.5, diffn model, free search

MiniZinc Chuffed, Priority Search Version 2.5.5, diffn model, priority search on start and machine variable

MiniZinc Cplex, Free Search Version 20.1.0, diffn model, free search

Cplex Version 20.1.0, MILP model of Section 3.3

The **Dispatch Rule** and **Local Search** solvers of Section 3.4 were implemented in Java, and are run inside the main application.

We use a timeout of 300 seconds for each solver and each instance. The best computed lower bound from Section 3.5 is provided to each solver, and we also provide an initial upper bound by running the Dispatch rule solver for 10 seconds for each instance.

Both the Cplex version of the MiniZinc model, and our own MILP model of Section 3.3 were not able to improve the given upper bound within the timeout, even for the smallest instances. This seems due to a poor initial LP relaxation, probably due to the added machine assignment choice. This contrasts with the results in [27], where MILP models were shown to be competitive for smaller problem instances of the job-shop problem type, where the machine for each task is known a priori.

Table 1 shows a comparison of the different solution approaches for a sample set of instances, ranging from 20–400 jobs.

While for the smaller instances the difference in solution quality is quite small, the difference increase as the number of jobs increases. We can see that the initial upper bound obtained in 10 seconds is quite good, with only modest further improvement made by the Dispatch and Local Search solvers. CPOptimizer with its default search finds better solutions, and is the best overall on medium sized instances, but SICStus Prolog with a custom search

■ **Table 1** Average Cmax Value over 25 instances dependent on size (number of jobs), 8 stages, Lanes Transport, Zipf Exponent 1.05 Job Type Distribution, 10 machines/stage, best solver marked in green, – indicates no solution better than upper bound found, timeout 300s.

Size	Lower Bound	Upper Bound	CP Opt	Chuffed Free	Chuffed Priority	Dispatch Rule	Local Search	SICStus
20	61.88	63.56	62.72	63.48	63.04	63.28	63.20	62.72
25	62.84	65.96	64.24	–	64.76	65.20	64.84	64.16
30	64.12	70.24	66.68	–	68.44	69.16	68.24	66.84
40	65.32	77.36	72.56	–	75.40	76.08	75.28	73.28
50	67.24	84.52	78.40	–	82.24	83.16	82.24	79.40
100	94.72	120.12	115.16	–	116.96	118.28	118.92	113.04
200	153.08	185.16	180.48	–	181.32	182.80	184.76	176.72
300	214.96	249.12	248.96	–	248.76	246.96	248.88	240.96
400	275.36	311.60	311.28	–	–	308.76	311.40	303.16

on average provides the best solution for larger instances. The 32 bit version of SICStus used here runs out of memory when we increase the problem size even further. The Chuffed free search only finds a better solutions than the given upper bound for the smallest problem size, the priority search in Chuffed scales better, but also times out for larger problem instances.

4.3.2 Scenario 1

Table 2 shows the result for a scenario with 200 jobs solved with different solvers. For each solver, we show the average makespan for a given parameter value, as well as the percentage increase over the best result obtained for that solver. We see that increasing the reach of the transport improves the quality of the schedule that can be reached for all solvers except CPOptimizer. CPOptimizer is much less affected by the transport restriction, and finds the best solutions for the strongly constrained cases, but is not as successful for the more relaxed cases. The custom search for SICStus finds the best solutions in the more relaxed cases, but only finds rather poor solutions in the more restricted scenarios. Comparing the best results for all solvers, not allowing transport beyond the initial lanes (limit 0) incurs a cost of 4.92%, while only allowing movement to neighboring lanes (limit 1) costs 2.66% over the unrestricted case. Note that for some instances better solutions were found by imposing a limit on the transport.

4.3.3 Scenario 2

Figure 2 illustrates the different layout alternatives studied in Scenario 2, with the relevant transport times indicated for one machine in stage 3, which is linked to machines in stages 4 and 5, as stage 4 is skipped for certain products. In Scenario 2a, we consider a single facility, where the transport time between machines is determined by their distance. In Scenario 2b, there are two facilities, transport between the facilities requires a longer inter-building transport time, marked in red. In Scenario 2c, there are two facilities working in parallel, where jobs can be moved between the buildings, at a higher cost. This transport is not allowed in Scenario 2d and 2e, jobs can only move between machines in the same facility.

Table 3 shows the results for a run of 25 instances with 200 jobs each for the different layout scenarios. We concentrate on the results with SICStus, which are better than those for the other solvers, and which differentiate the different scenarios more clearly. Splitting the production between two facilities sequentially (Scenario 2b) increases the makespan by 4.5% on average, as every job is delayed by the long transport between the facilities. On the

16:14 The HFP with Transportation Times

■ **Table 2** Results for Scenario 1 – Changing transportLimit parameter for different solvers, average Makespan over 25 instances, 8 stages, 10 machines/stage, Zipf exponent 1.05 Job Type Distribution, average lower bound is 153.08, best average solution marked in green.

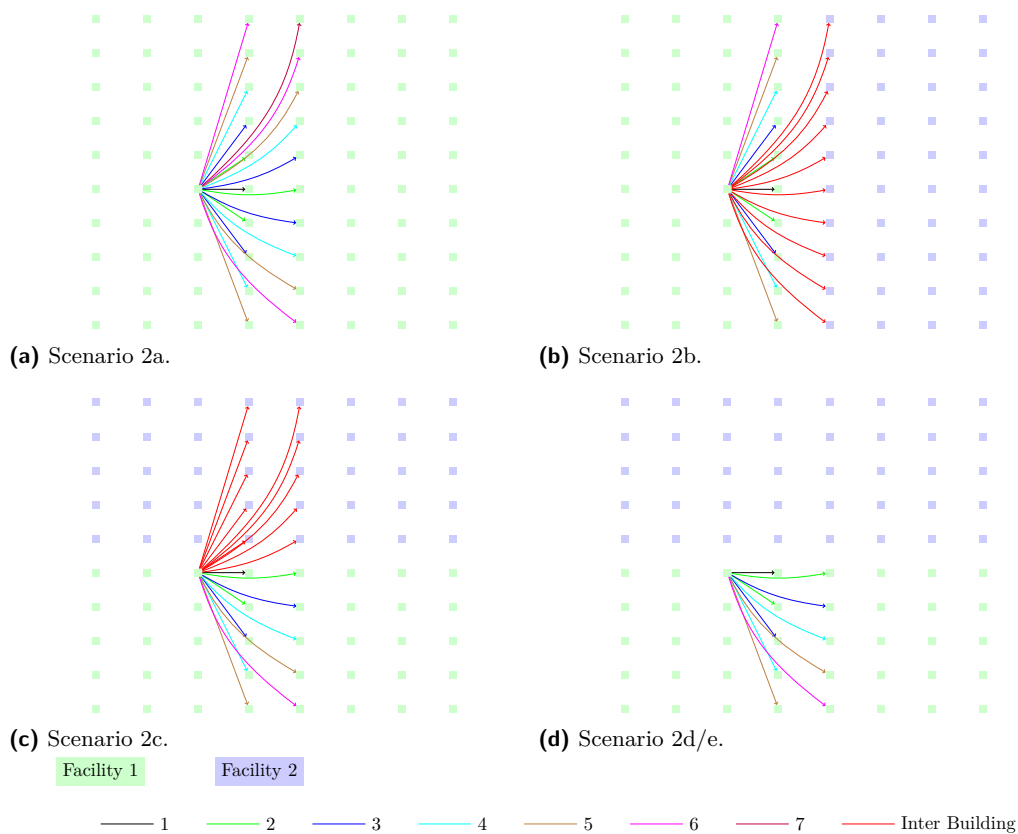
Solver	Transport Limit Between Lanes						No Limit
	0	1	2	3	4	5	
SICStus	204.40	184.20	180.44	178.52	177.88	177.40	177.36
% over Best	15.51	4.09	1.97	0.88	0.52	0.25	
CPOptimizer	186.08	182.96	180.60	180.72	180.80	180.84	180.88
% over Best	3.89	2.14	0.83	0.89	0.94	0.96	
Dispatch Rule	200.84	186.72	184.08	183.72	183.32	182.92	182.92
% over Best	9.99	2.26	0.81	0.61	0.39	0.18	
Local Search	199.32	188.56	186.20	185.48	185.12	184.68	184.80
% over Best	8.21	2.37	1.09	0.69	0.50	0.26	0.33

other hand, splitting the work between two facilities in parallel has a smaller impact. If we are still able to transport intermediate products between locations (Scenario 2c), then the makespan increases by less than 1%. This doubles to 2% if transport between facilities is no longer allowed in Scenario 2d. Restricting the problem further, by considering the assignment of jobs to facility as given for 80% of the orders, causes no further degradation in result quality. Instead, for a small number of cases, the scheduler finds an improved solution, as the domain of the machine assignment variables is cut in half, reducing the search complexity. We are of course not able to explore any of these search spaces completely. But the results for the SICStus solver are statistically significant, when tested with paired t-tests. With the exception of scenarios 2d and 2e, which cannot be distinguished, all scenarios produce significantly different results over the generated 25 instances.

The change of the schedule quality based on the layout scenario is only one aspect in evaluating the alternatives, other parts use customer specific data, and are not reported here.

■ **Table 3** Results for Scenario 2 – Average Makespan over 25 instances with 200 jobs, different Solvers, 10 machines/stage, Zipf 1.05 Job Type Distribution, average lower bound is 159.80.

Solver	Size	Scenario				
		2a	2b	2c	2d	2e
SICStus	200	176.84	184.84	178.28	180.52	180.48
% over Best		0.00	4.52	0.81	2.08	
CPOptimizer	200	184.40	190.92	186.00	183.52	183.52
% over Best		1.23	4.81	2.11	0.75	
Dispatch	200	182.76	190.44	184.28	184.60	184.64
% over Best		0.00	4.20	0.83	1.01	
Local Search	200	184.68	192.24	185.76	186.08	185.96
% over Best		0.13	4.23	0.72	0.89	



■ **Figure 2** Scenario 2 - Layout Alternatives and Sample Transport Times.

5 Future Work and Conclusion

In this paper we have presented a novel variant of a hybrid, flexible flowshop problem which adds transportation time between stages. We have explored different models for the problem, using CP and other technologies, and compared them on a number of generated problem instances, based on parameters given by the industrial partner. While some solvers seem promising on small instances, several of the solvers considered failed to handle the problem sizes required for a realistic scenario. We then considered two factory design problems and compared the solutions obtained for different problem settings. These results can be used to evaluate the impact of the design choices on the operational efficiency of a plant layout. While the results obtained are very encouraging, there is still a rather big gap between the lower bound found and the best solutions. Future work will be focused on improving the bounds, while also considering hybridisation techniques that might allow us to find better solutions for large problem instances. Finally, we believe that the problem of designing optimized facility layouts, where the positions of all the machines are computed to increase efficiency, may be of interest for future research.

References

- 1 Leonard Adler, Nelson Fraiman, Edward Kobacker, Michael Pinedo, Juan Carlos Plotnicoff, and Tso Pang Wu. BPSS: A scheduling support system for the packaging industry. *Operations Research*, 41(4):641–648, 1993. doi:10.1287/opre.41.4.641.
- 2 Abder Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17:57–73, 1993. doi:10.1016/0895-7177(93)90068-A.

- 3 Fardin Ahmadizar and Parmis Shahmaleki. Group-shop scheduling with sequence-dependent set-up and transportation times. *Applied Mathematical Modelling*, 38(21):5080–5091, 2014. doi:10.1016/j.apm.2014.03.035.
- 4 Miguel F. Anjos and Manuel V.C. Vieira. Mathematical optimization approaches for facility layout problems: The state-of-the-art and future research directions. *European Journal of Operational Research*, 261(1):1–16, 2017. doi:10.1016/j.ejor.2017.01.049.
- 5 Farhad Azadivar and John(Jian) Wang. Facility layout optimization using simulation and genetic algorithms. *International Journal of Production Research*, 38(17):4369–4383, 2000. doi:10.1080/00207540050205154.
- 6 Javad Behnamian, Seyyed Mohammad Taghi Fatemi Ghomi, Fariborz Jolai, and Omid Amirtaheri. Realistic two-stage flowshop batch scheduling problems with transportation capacity and times. *Applied Mathematical Modelling*, 36(2):723–735, 2012. doi:10.1016/j.apm.2011.07.011.
- 7 Nicolas Beldiceanu and Evelyne Contejean. Introducing Global Constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, 1994. doi:10.1016/0895-7177(94)90127-9.
- 8 Fehmi Burcin Ozsoydan and Mijgan Sağır. Iterated greedy algorithms enhanced by hyper-heuristic based learning for hybrid flexible flowshop scheduling problem with sequence dependent setup times: A case study at a manufacturing plant. *Computers & Operations Research*, 125:105044, 2021. doi:10.1016/j.cor.2020.105044.
- 9 Mats Carlsson and Per Mildner. SICStus Prolog-the first 25 years. *Theory and Practice of Logic Programming*, 12(1-2):35–66, 2012. doi:10.1017/S1471068411000482.
- 10 Hyun-Seon Choi, Ji-Su Kim, and Dong-Ho Lee. Real-time scheduling for reentrant hybrid flow shops: A decision tree based mechanism and its application to a TFT-LCD line. *Expert Systems with Applications*, 38(4):3514–3521, 2011. doi:10.1016/j.eswa.2010.08.139.
- 11 Geoffrey Chu. *Improving Combinatorial Optimization*. PhD thesis, The University of Melbourne, 2011.
- 12 Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf, and Françoise Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems, FGCS 1988, Tokyo, Japan, November 28-December 2, 1988*, pages 693–702. OHMSHA Ltd. Tokyo and Springer-Verlag, 1988.
- 13 Housni Djellab and Khaled Djellab. Preemptive hybrid flowshop scheduling problem of interval orders. *European Journal of Operational Research*, 137(1):37–49, 2002. doi:10.1016/S0377-2217(01)00094-7.
- 14 Amine Drira, Henri Pierreval, and Sonia Hajri-Gabouj. Facility layout problems: A survey. *Annual Reviews in Control*, 31(2):255–267, 2007. doi:10.1016/j.arcontrol.2007.04.001.
- 15 Atabak Elmi and Seyda Topaloglu. Scheduling multiple parts in hybrid flow shop robotic cells served by a single robot. *International Journal of Computer Integrated Manufacturing*, 27(12):1144–1159, 2014. doi:10.1080/0951192X.2013.874576.
- 16 Atabak Elmi and Seyda Topaloglu. Multi-degree cyclic flow shop robotic cell scheduling problem: Ant colony optimization. *Computers & Operations Research*, 73:67–83, 2016. doi:10.1016/j.cor.2016.03.007.
- 17 Atabak Elmi and Seyda Topaloglu. Multi-degree cyclic flow shop robotic cell scheduling problem with multiple robots. *International Journal of Computer Integrated Manufacturing*, 30(8):805–821, 2017. doi:10.1080/0951192X.2016.1210231.
- 18 Thibaut Feydy, Adrian Goldwaser, Andreas Schutt, Peter Stuckey, and Kenneth Young. Priority search with MiniZinc. In *ModRef 2017: The Sixteenth International Workshop on Constraint Modelling and Reformulation*, 2017.
- 19 Erik Flores Garcia, Enrique Ruiz Zúñiga, Jessica Bruch, Matias Urenda Moris, and Anna Syberfeldt. Simulation-based optimization for facility layout design in conditions of high uncertainty. *Procedia CIRP*, 72:334–339, 2018. 51st CIRP Conference on Manufacturing Systems. doi:10.1016/j.procir.2018.03.227.

- 20 Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1979.
- 21 Ronald Graham, Eugene Lawler, Jan Karel Lenstra, and Alexander H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979. doi:10.1016/S0167-5060(08)70356-X.
- 22 Alain Guinet. Textile production systems: a succession of non-identical parallel processor shops. *Journal of the Operational Research Society*, 42(8):655–671, 1991. doi:10.1057/jors.1991.132.
- 23 Mohamed Haouari and Lotfi Hidri. On the hybrid flowshop scheduling problem. *International Journal of Production Economics*, 113:495–497, May 2008. doi:10.1016/j.ijpe.2007.10.007.
- 24 Lofti Hidri, Saneur Elkosantini, and Mohammed M. Mabkhot. Exact and heuristic procedures for the two-center hybrid flow shop scheduling problem with transportation times. *IEEE Access*, 6:21788–21801, 2018. doi:10.1109/ACCESS.2018.2826069.
- 25 Zhihong Jin, Katsuhisa Ohno, Takahiro Ito, and Salah E. Elmaghraby. Scheduling hybrid flowshops in printed circuit board assembly lines. *Production and Operations Management*, 11(2):216–230, 2002. doi:10.1111/j.1937-5956.2002.tb00492.x.
- 26 Antoine Jouglet, Ceyda Oğuz, and Marc Sevaux. Hybrid flow-shop: a memetic algorithm using constraint-based scheduling for efficient search. *Journal of Mathematical Modelling and Algorithms*, 8(3):271–292, 2009. doi:10.1007/s10852-008-9101-1.
- 27 Wen-Yang Ku and Christopher Beck. Mixed integer programming models for job shop scheduling: A computational analysis. *Computers & Operations Research*, 73:165–173, 2016. doi:10.1016/j.cor.2016.04.006.
- 28 Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP optimizer for scheduling. *Constraints*, 23(2):210–250, 2018. doi:10.1007/s10601-018-9281-x.
- 29 Michael Langston. Interstage transportation planning in the deterministic flowshop environment. *Operations Research*, 35(4):556–564, 1987. doi:10.1287/opre.35.4.556.
- 30 Chuanjin Lei, Ning Zhao, Song Ye, and Xiuli Wu. Memetic algorithm for solving flexible flow-shop scheduling problems with dynamic transport waiting times. *Computers & Industrial Engineering*, 139:105984, 2020. doi:10.1016/j.cie.2019.07.041.
- 31 P.L. Maggu and G. Das. On $2 \times n$ sequencing problem with transportation times of jobs. *Pure and Applied Mathematical Sciences*, 12:1–6, 1980.
- 32 Byungsoo Na, Shabbir Ahmed, George Nemhauser, and Joel Sokol. A cutting and scheduling problem in float glass manufacturing. *Journal of Scheduling*, 17(1):95–107, 2014. doi:10.1007/s10951-013-0335-z.
- 33 Bahman Naderi, Sheida Gohari, and Mehdi Yazdani. Hybrid flexible flowshop problems: Models and solution methods. *Applied Mathematical Modelling*, 38(24):5767–5780, 2014. doi:10.1016/j.apm.2014.04.012.
- 34 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.
- 35 Andrea Rossi. Flexible job shop scheduling with sequence-dependent setup and transportation times by ant colony with reinforced pheromone relationships. *International Journal of Production Economics*, 153(C):253–267, 2014. doi:10.1016/j.ijpe.2014.03.006.
- 36 Rubén Ruiz and José Antonio Vázquez Rodríguez. The hybrid flow shop scheduling problem. *European Journal of Operational Research*, 205:1–18, 2010. doi:10.1016/j.ejor.2009.09.024.
- 37 Rubén Ruiz, Funda Sivrikaya Şerifoğlu, and Thijs Urlings. Modeling realistic hybrid flexible flowshop scheduling problems. *Computers & Operations Research*, 35(4):1151–1175, 2008. doi:10.1016/j.cor.2006.07.014.

- 38 Helmut Simonis, Michele Garraffa, Barry O’Sullivan, and Eddie Armstrong. Dataset for Article: Hybrid Flexible Flowshop with Transportation Times at CP 2021, August 2021. doi:10.5281/zenodo.5168966.
- 39 Ricardo Soto, Hakan Kjellerstrand, Juan Gutiérrez, Alexis López, Broderick Crawford, and Eric Monfroy. Solving manufacturing cell design problems using constraint programming. In He Jiang, Wei Ding, Moonis Ali, and Xindong Wu, editors, *Advanced Research in Applied Artificial Intelligence*, pages 400–406, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- 40 Ömür Tosun, Mariappan Kadarkarainadar Marichelvam, and Nedret Tosun. A literature review on hybrid flow shop scheduling. *International Journal of Advanced Operations Management*, 12(2):156-194, 2020. doi:10.1504/ijaom.2020.108263.
- 41 Srisatja Vitayasak, Pupong Pongcharoen, and Christian Hicks. Robust machine layout design under dynamic environment:dynamic customer demand and machine maintenance. *Expert Systems with Applications*, 3:100015, 2019. doi:10.1016/j.eswax.2019.100015.
- 42 Hua Xuan. Hybrid flowshop scheduling with finite transportation capacity. *Applied Mechanics and Materials*, 65:574-578, 2011. doi:10.4028/www.scientific.net/amm.65.574.
- 43 Sami Yamani Douzi Sorkhabi, David Romero, Christopher Beck, and Cristina Amon. Constrained multi-objective wind farm layout optimization: Novel constraint handling approach based on constraint programming. *Renewable Energy*, 126:341–353, 2018. doi:10.1016/j.renene.2018.03.053.
- 44 Weiya Zhong and Zhi-Long Chen. Flowshop scheduling with interstage job transportation. *Journal of Scheduling*, 18(4):411–422, 2015. doi:10.1007/s10951-014-0409-6.
- 45 Jinlian Zhou, Ying Guo, and Guipeng Li. On complex hybrid flexible flowshop scheduling problems based on constraint programming. In *12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, pages 909–913, 2015. doi:10.1109/fskd.2015.7382064.

CLR-DRNets: Curriculum Learning with Restarts to Solve Visual Combinatorial Games

Yiwei Bai ✉

Cornell University, Ithaca, NY, USA

Di Chen ✉

Cornell University, Ithaca, NY, USA

Carla P. Gomes ✉

Cornell University, Ithaca, NY, USA

Abstract

We introduce a curriculum learning framework for challenging tasks that require a combination of pattern recognition and combinatorial reasoning, such as single-player visual combinatorial games. Our work harnesses Deep Reasoning Nets (DRNets) [4], a framework that combines deep learning with constraint reasoning for unsupervised pattern demixing. We propose CLR-DRNets (pronounced Clear-DRNets), a curriculum-learning-with-restarts framework to boost the performance of DRNets. CLR-DRNets incrementally increase the difficulty of the training instances and use restarts, a new model selection method that selects multiple models from the same training trajectory to learn a set of diverse heuristics and apply them at inference time. An enhanced reasoning module is also proposed for CLR-DRNets to improve the ability of reasoning and generalize to unseen instances. We consider Visual Sudoku, i.e., Sudoku with hand-written digits or letters, and Visual Mixed Sudoku, a substantially more challenging task that requires the demixing and completion of two overlapping Visual Sudokus. We propose an enhanced reasoning module for the DRNets framework for encoding these visual games. We show how CLR-DRNets considerably outperform DRNets and other approaches on these visual combinatorial games.

2012 ACM Subject Classification Computing methodologies → Machine learning

Keywords and phrases Unsupervised Learning, Combinatorial Optimization

Digital Object Identifier 10.4230/LIPIcs.CP.2021.17

Funding This research was supported by NSF awards CCF-1522054 (Expeditions in computing), CNS-1059284 (Infrastructure), and an ARO award (DURIP, W911NF-17-1-0187, AIDA compute cluster).

Acknowledgements We want to thank Wenting Zhao and anonymous reviewers for their valuable feedback.

1 Introduction

Deep learning has surpassed human-level performance on many perception tasks, ranging from object recognition to language translation. However, these successes heavily rely on the availability of large datasets and corresponding labels. In contrast, humans often only have access to a few examples, and therefore they resort to meticulous reasoning about prior knowledge to compensate for the lack of labeled data and fill in the data information gaps. Herein we consider unsupervised or weakly supervised single-player visual combinatorial games. Humans tackle such challenging tasks by combining pattern recognition with reasoning about prior knowledge (the games' rules). Visual combinatorial games capture various real-world applications, particularly scientific data interpretation tasks, which are in general unsupervised or weakly supervised but for which rich prior knowledge is often available [4]. Consider the case of Visual Sudoku [22], a variant of the standard Sudoku with hand-written



© Yiwei Bai, Di Chen, and Carla P. Gomes;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 17; pp. 17:1–17:14

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

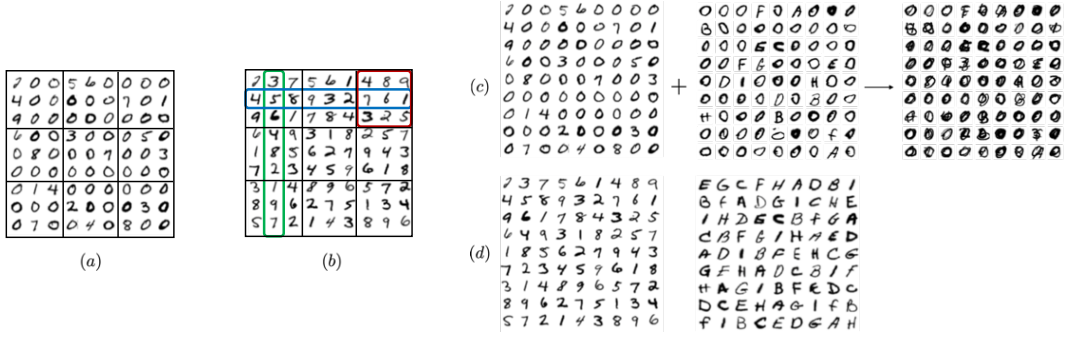


Figure 1 (a) Visual Sudoku: 0 denotes the empty cell, other digits are hints. Goal: replace the empty cells with digits from 1 to 9 to obtain a valid Sudoku, i.e., the cells in each row (blue rectangle), column (green rectangle), and any of the nine marked 3x3 boxes (red square) have to have all-different digits. (b) The solution to (a). (c) Visual Mixed Sudoku: 0 and O denote empty cells. The goal is to replace the empty cells (denoted by overlapping 0 and O characters) in the mixed Sudoku on the right with digits from 1 to 9 and letters from A to I, and obtain two valid demixed Sudokus. Note overlapping here is the max operator. (d) The solution to (c).

digits (see Fig. 1). We solve Visual Sudokus **without any Sudoku labels** by combining our perception skills, for digit recognition, with logical reasoning about Sudoku rules, to disambiguate noisy digits and fill in the missing digits. The missing digits simulate real-world settings in which there are missing data. Visual Mixed Sudoku (Fig. 1) is even more challenging than Visual Sudoku as it requires identifying the missing digits or letters of two partially filled overlapping Sudokus. Visual Mixed Sudoku involves demixing the partially filled Sudokus and inferring the missing digits. As we show in the experimental section, a straightforward approach for Visual Sudoku that first uses a well-trained state-of-the-art deep-learning digit classifier and passes the digit information as input to a powerful combinatorial solver, such as a SAT solver, does not lead to satisfactory results. Even though the classifier has high accuracy (99.4%), it still makes mistakes and therefore, when there are many hints in the Visual Sudoku, it is likely that the classifier will make a few mistakes, leading to noisy data that SAT solvers cannot handle. This approach performs even more poorly for the Visual Mixed Sudoku, given the higher probability of making digit/letter classification mistakes due to a combination of factors (additional challenging demixing task, lower accuracy of letter classifiers, and the fact that we double the number of hints corresponding to the two overlapping Sudokus). Therefore an approach integrating digit/letter recognition with reasoning about the Sudoku rules in an end-to-end fashion is required. Deep Reasoning Networks (DRNets) [4] is a framework proposed recently that seamlessly integrates deep learning with constraint reasoning via an interpretable latent space, to incorporate prior knowledge (such as Sudoku rules). DRNets were shown to be effective for unsupervised demixing tasks, such as the demixing of two solved (i.e., all the digits filled in) overlapping Visual Sudokus. Nevertheless, DRNets have limited reasoning generalization capabilities, in particular for completion tasks. As we will show, Visual Sudoku and Visual Mixed Sudoku are substantially challenging for DRNets since that in addition to the demixing task, they involve the demanding completion task that requires inferring missing digits. **Our contributions:** (1) We propose **CLR-DRNets** (pronounced clear DRNets), a **curriculum learning** framework to boost the performance of DRNets. Our approach is inspired by how humans learn to solve complex problems, starting with easy instances and gradually increasing the instances' difficulty. Another intuition behind our

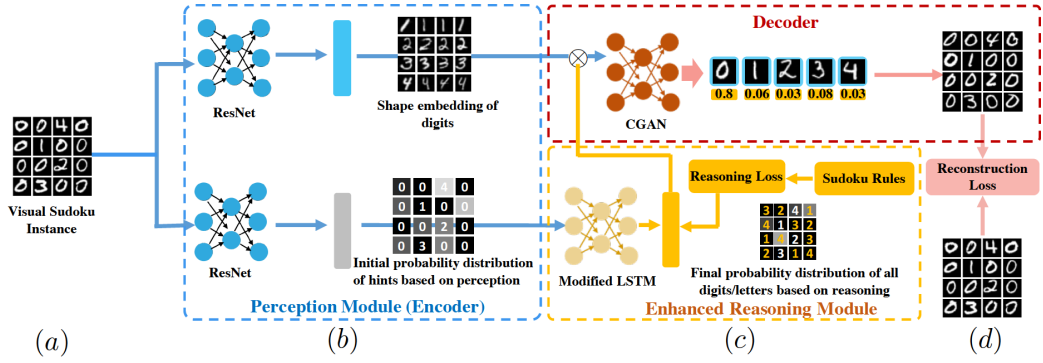
approach is that the perception task is relatively easier than the reasoning task in many visual combinatorial games. Therefore, it is crucial to carefully select a sequence of training examples to balance the reasoning and perception tasks to prevent the model from focusing only on the perception task. **(2)** We propose an **enhanced reasoning module** to improve the power of reasoning and generalization to unseen data. **(3)** We propose **restarts**, a new model selection strategy for improving the performance at test time. We note that it is usually hard to solve a complex visual combinatorial problem in a single inference step of a neural network. Since CLR-DRNets are trained without label supervision, only with prior knowledge supervision (e.g., domain rules), we can still improve and customize the learned model with respect to the specific test data by further optimizing the loss function. We also find that the reasoning module of CLR-DRNets can learn different heuristics during training. Thus, saving several different models from the training phase and applying each of them during test time sequentially is really helpful. **(4)** We propose encodings for **(4.1) Visual Sudoku** and **(4.2) Visual Mixed Sudoku** using the CLR-DRNets framework. and **(5)** show how **CLR-DRNets substantially outperform DRNets and other approaches on these visual combinatorial games.**

2 Related Work

Combining perception and reasoning. CLR-DRNets leverage Deep Reasoning Networks (DRNets) [4]. DRNets were shown to be effective for unsupervised demixing tasks. **Here we use the DRNets framework for unsupervised visual combinatorial games, which are more challenging reasoning tasks than demixing,** and therefore we develop a **strengthened reasoning module for DRNets.** Furthermore, **CLR-DRNets further boost the power of DRNets with its curriculum-learning-with-restarts approach and generalize to unseen instances, in contrast to DRNets, designed to solve instances.** For Visual Sudoku, other approaches focus on learning the Sudoku rules from the labeled data [1, 24, 16, 9]. For example, SATNET [22] introduces a differentiable maximum satisfiability (MAXSAT) solver that can be incorporated with deep learning models to capture the reasoning rules efficiently. [3] learns problem related cost functions for Constraint Networks, which are solved with a specialized constraint solver. [18] integrates the perception and reasoning through exposing the predicted probability to the constraint solver. However, except for DRNets and [18], previous approaches require labeled training Sudoku data.

Curriculum learning (CL) for solving hard tasks. CL is widely used in deep reinforcement learning [19, 7, 21]. For example, Feng et al. [7] let the model firstly learn from the easy sub-task created from the hard original task. Then gradually increase the hardness of the sub-task until the agent can solve the original task. Our CL method shares a similar idea but we do not use it in a RL setting and we do not require that the easy instances are generated by the hard instances. Some works also employ CL for deep learning [23, 12]. For instance, Hacohen et al. [12] introduce a CL framework for image classification.

Restarts in deep learning. *Restarts* are widely used in the CP/SAT community (e.g., [10, 2]) and also in stochastic optimization to solve non-convex problems (e.g., [6, 8]). In the deep learning training phase, learning rate restart [11] is proven to increase the performance. DRNets use simple restarts, basically to randomly group different instances into one mini-batch to compute various gradients, CLR-DRNets in addition leverage different models acquired from one training trajectory and apply them sequentially to test (unseen) data.



■ **Figure 2** (a) The digit visual Sudoku instance. (b) The perception model (encoder) of CLR-DRNets. The top blue rectangle is the latent space capturing the shape information of all possible digits per cell. The bottom gray rectangle is the initial estimation of input digit visual Sudoku instance only based on perception. Here we employ the heat map to represent the predicted confidence of each cell. (c) The enhanced reasoning module of CLR-DRNets (not in the original DRNets) consists of a modified LSTM model. It takes the initial estimation as the input, constrained by the relaxed Sudoku rules' loss function, computing the completion results. (d) The decoder part leverages a pre-trained cGAN to generate all the possible digit images w.r.t the shape embeddings of (b) per cell. We can reconstruct each cell guided by the completion results of (c). (The figure uses 4×4 Sudokus for easier visualization. All our experiments are with 9×9 Sudokus.)

3 CLR-DRNets

We start by providing a high-level description of DRNets.[4].

DRNets. DRNets perform end-to-end unsupervised deep reasoning using a perception module (*encoder*) to produce the *initial estimation of the visual inputs*, which are constrained to adhere to prior knowledge via a *reasoning module*. The reasoning module encodes the constraint loss function using the *initial estimation*. A *generative decoder* uses the initial estimation of the visual inputs to generate the reconstruction of the input. DRNets solve the problem by jointly optimizing the reconstruction loss, encouraging the reconstruction to be similar to the input, and the constraint loss function, to enforce the domain rules.

CLR-DRNets borrow the general framework from DRNets and further strengthen it with an *enhanced reasoning module*. We propose a *curriculum learning framework* to tackle the difficult visual combinatorial completion tasks, which is beyond the capability of the original DRNets. Moreover, *restarts*, a new model selection strategy is proposed to boost the performance at test time. The enhanced reasoning module in CLR-DRNets is much more powerful and allows generalization to unseen data, in contrast to DRNets. Adapting from the DRNets [4] framework, CLR-DRNets formulate the entire process as a **data-driven optimization problem**:

$$\min_{\theta_p, \theta_r} \frac{1}{N} \sum_{i=1}^N \underbrace{\lambda^p \psi^p(f_{\theta_p}(x_i))}_{\text{regularize the initial estimation}} + \underbrace{\psi^r(f_{\theta_r}(f_{\theta_p}(x_i)))}_{\text{regularize the reasoning output new in CLR-DRNets}} + \underbrace{\lambda^l \mathcal{L}(cGAN(f_{\theta_r}(f_{\theta_p}(x_i))), x_i)}_{\text{regularize both the initial estimation and the reasoning output modified from DRNets}} \quad (1)$$

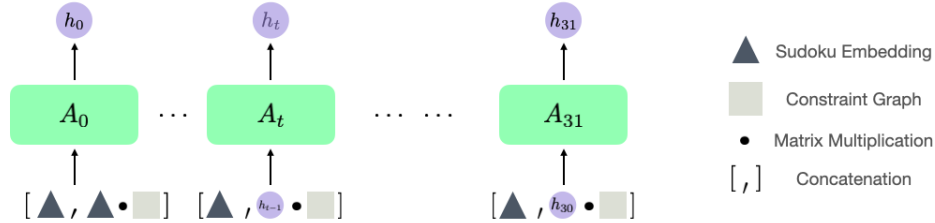
where f_{θ_r} and f_{θ_p} are the reasoning module and the perception module respectively, $cGAN$ is a pre-trained conditional generative adversarial network (cGAN)[17], x_i is the i -th data point of the input, ψ^p, ψ^r are the penalty functions of continuously relaxed constraints

related to the perception module and the reasoning module, λ^p, λ^l are weight scalars and \mathcal{L} is any distance metric. In equation 1, the term labeled by “*regularize both the initial estimation and the reasoning output*” is a **modified version of DRNets’ original loss function** and the term labeled by “*regularize the reasoning output*” is **completely new in CLR-DRNets**. These modified terms are associated with CLR-DRNets’ enhanced reasoning module. Below we describe CLR-DRNets highlighting the main differences with respect to the general DRNets framework.

CLR-DRNets. We illustrate the CLR-DRNets framework with our proposed models (see Fig. 2) for the visual Sudoku games (Fig. 1 (a)). Similarly to DRNets, CLR-DRNets employ two ResNets for the perception module to predict the distribution of all digits in each cell along with the shape embedding (*initial estimation*). The constraints for the initial estimation (ψ^p) are that the predicted digits probability distribution of each cell should converge to only one digit.

CLR-DRNets’ enhanced reasoning module is a modified long short term memory (LSTM) model [14] that can compute all the possible digits’ distribution for each cell, capturing the Sudoku structure, and with sufficient power to reason about missing digits (more details below). The constraints for the reasoning module (ψ^r) are the relaxed Sudoku constraint losses, which can regularize the reasoning outputs to satisfy the Sudoku rule (more details below).

The decoder is similar to DRNets’ and consists of a conditional Generative Adversarial Networks (cGAN) [17] pre-trained on the prototypes, which are images of all possible single digits. The decoder generates all possible digit images w.r.t the shape embeddings of the initial estimation. Then per each cell, all possible digit images are remixed based on the distribution of digits from the reasoning and the perception module. Cells that are predicted as hints are encouraged to have a reconstructed image similar to the input image, which can further regularize the initial estimation and the reasoning outputs.



■ **Figure 3** Modified LSTM model. The Sudoku embedding replaces each digit of a Sudoku with a learnable embedding. The constraint graph is a 81×81 matrix capturing Sudoku constraints between every pair of cells (i.e., there is a constraint between any pair of cells in the same row, same column, and same block). h_{31} goes through a fully connected layer to get the final reasoning results.

CLR-DRNets’ enhanced reasoning module. DRNets’ reasoning ability entirely relies on the continuous relaxation of the combinatorial constraints, i.e., it converts the discrete constraints into a differentiable loss function and solves the problem by optimizing it. However, this reasoning method is *stateless*, i.e., it is difficult to generalize to unseen data. CLR-DRNets’ enhanced reasoning module employs a constraint graph and a modified LSTM (see Fig. 3) to learn from the training examples and generalize to unseen data. The constraint graph is a square matrix where each entry represents the constraints between these two elements.

■ **Algorithm 1** Curriculum Learning framework for solving data-driven optimization problems.

Input: Training problem instances D , Target problem instances T , CLR-DRNets model M , Difficulty Gap G , Hard set of constraints ψ^h .
 Select or generate a set of instances of proper difficulty level D_{train} based on ψ^h ;
while D_{train} is not as hard as T **do**
 Train M using D_{train} via optimizing equation 1 ;
 do
 Select/generate a harder set of instances D_{train}^\dagger based on D_{train}, ψ^h ;
 $D_{\text{train}} \leftarrow D_{\text{train}}^\dagger$;
 while D_{train}^\dagger meets G ;
end

For example, the constraint graph for a 4×4 Sudoku is a 16×16 matrix. If entry (x, y) is one, it means the cell x and cell y are in the same row, column, or block otherwise zero. Then the input of each LSTM block is the concatenation of the Sudoku embedding and the multiplication of the hidden state and the constraints graph, where the Sudoku embedding replaces each digit of a Sudoku with a learnable embedding.

Now we explain how the unsupervised differentiable Sudoku loss function is derived, using DRNets' continuous relaxation. The reasoning module computes the digits' distribution for each Sudoku's cells and we denote the distribution of cell (i, j) as $P_{i,j}$ (row i , column j). The loss for encoding the Sudoku's row constraint is: $L_r = -\sum_{i=1}^9 H(\frac{1}{9} \sum_{j=1}^9 P_{i,j})$, where H is the entropy function. Similarly, we can define the column and block constraint loss L_c and L_b respectively. Since each cell contains one digit, the semantic constraint loss for cells is: $L_{\text{cell}} = \sum_{i=1}^9 \sum_{j=1}^9 H(P_{i,j})$. These loss functions minimize the entropy of each cell's digits distribution while maximizing the entropy of the average distribution of digits in each row, column, and box, forcing the distribution of each cell to converge to one digit while each row, column and box has different digits. Formally, the unsupervised **Sudoku constraint loss** is defined as: $L_{\text{Sudoku}} = \lambda_1(L_c + L_r + L_b) + \lambda_2 L_{\text{cell}}$, where λ_1 and λ_2 are weight scalars. Moreover, we show how the reconstruction loss for cell (i, j) is derived. Denote the embeddings of cell (i, j) as $z_{i,j}[t]$, where t can be any possible object, e.g., t can be digit 1 to 9 in visual Sudoku games. The predicted digit distribution from the perception module for the cell (i, j) is $P'(i, j)$. Note $P_{i,j}$ is defined above as the digits' distribution predicted by the reasoning module. These two distributions are mixed to form the $P_{\text{mix}}(i, j)$ for reconstruction: $P_{\text{mix}}(i, j)[t] = (1 - P'(i, j)[0]) * P'(i, j)[t] + P'(i, j)[0] * P(i, j)[t]$, where digit 0 represents the missing digits of the Sudoku. The reconstructed input for cell (i, j) is: $X_{\text{recon}}(i, j) = \sum_{t=1}^9 P_{\text{mix}}(i, j)[t] * cGAN(z_{i,j}[t])$. The reconstruction loss is then defined as $L_{\text{recon}} = \sum_{i=1}^9 \sum_{j=1}^9 \mathcal{L}(X(i, j), X_{\text{recon}}(i, j))$, where X is the input image and \mathcal{L} is any distance metric. This mixed distribution contributes to the reasoning ability of CLR-DRNets in which the wrong perception estimation can be corrected by the reasoning outputs while the initial estimation of images also help the reasoning module make decisions.

Note that we relax the hard discrete constraints into continuous constraint loss functions. The different difficulty of the relaxed constraint loss functions ψ^r and ψ^p is a key challenge for training, which could cause the optimization to focus on the easy part and converge to some local minimum quickly. Thus, we proposed the curriculum learning to tackle the difficulty imbalance among different losses.

Curriculum learning for CLR-DRNets. We introduce a curriculum learning framework (see Alg. 1) to manage the difficulty imbalance of the perception and reasoning tasks in the data-driven optimization problem. Based on the prior knowledge and problem structure, we can identify a set of constraints ψ^h that are quite hard to optimize. We can generate/find some problems that are easier enough in terms of the hard constraints set ψ^h . For example, in Visual Sudoku, completing the Sudoku is much harder than classifying digit images, so we set ψ^h as the Sudoku rules. Visual Sudoku instances with many hints are easier than those with fewer hints. A model that perfectly solves the easier instances can be easily trained. Based on the problem structure, we define a difficulty gap G to guide the generation/selection of instances that are slightly harder. For example, in Visual Sudoku, the difficulty gap G is set to be 5 hints: we remove 5 hints to gradually increase the instance difficulty. We then use these harder instances to continue training our model. This process can be repeated several times to finally solve the instances of desired difficulty level. More details in the appendix.

CLR-DRNets' training strategy. The success of CLR-DRNets relies on the seamless cooperation of the perception module, the reasoning module, and the generative decoder. We propose two training strategies for achieving this goal, different from DRNets', since DRNets do not have parameters for the reasoning module. Both strategies employ a pre-trained cGAN and a pre-trained classifier. The first strategy is *joint training*: we train the entire CLR-DRNets at the same time, i.e., optimizing the loss function (see equation 1) and the gradients affect both the perception module and the reasoning module. This joint strategy can handle the case where only noisy, e.g., handwritten, input data are available. However, for some challenging reasoning problems, the noisy input may harm the ability of the reasoning module. Thus, we propose a second strategy, *separate training*: we separately train the reasoning module with non-noisy input (i.e., input values are known), i.e., optimizing only the second term of the equation 1. This *separate training* strategy can tackle more intricate problems, but it requires the non-noisy input training data (no labels required though).

It is challenging to generalize a *single* data-driven optimization model to unseen problem instances, capturing all the logical relations across instances, given the combinatorial search space. So we introduce *restarts* to remedy this issue.

Restarts, a new model selection strategy. The reasoning effort required for solving visual combinatorial games can be huge. A single inference step of the deep learning model is unlikely to be able to solve all the instances with different levels of difficulty. We derive our objective function **without label supervision**, where the only supervision is based on the prior knowledge, so we can still optimize our CLR-DRNets model for a few steps in the test phase to further improve the model and customize it with respect to the test data. Also, we observe that the accuracy metric is increasing smoothly during the training process, but the set of training instances that can be solved varies quite a bit. Thus, we postulate that our model is doing local search to solve the problem and the model parameters can be loosely interpreted as heuristic of the search algorithm. Inspired by the *restart* scheme broadly used by the combinatorial optimization community [10, 2], we propose a new model selection method based on restarts (see Alg. 2). Since we have observed that we can get very different heuristics during the training process, so the restart scheme starts by collecting several models with top validation performance to form a model pool M . Then we start from one model and for each unsolved test case the loss function is optimized until the instance is solved or for a maximum of restart gap g steps, switching to the next model when all the

Algorithm 2 Restart scheme for CLR-DRNets.

Input: Test instances T , CLR-DRNets model pools M , Restart Gap g , metric ϕ to evaluate whether the instance is solved

```

 $idx \leftarrow 0$ ;
while  $T = \emptyset$  or  $idx < \text{len}(M)$  do
   $m \leftarrow M[idx]$ ;
  for  $i \leftarrow 1$  to  $g$  do
     $R \leftarrow m(T)$ ;  $R$ : the solution of  $T$ .
     $S \leftarrow \phi(R)$ ;  $S$ : correctly solved instances of  $T$ .
     $T \leftarrow T \setminus S$ 
    update parameter of  $m$  w.r.t equation 1;
  end
   $idx \leftarrow idx + 1$ ;
end

```

instances are processed. The scheme is supposed to fine-tune the model for the underlying test data, so the restart gap g is typically small. Thus the restart procedure takes much fewer time compared with the training. Note that this scheme does not require labels.

4 Experiments

4.1 Visual-Sudoku

As an example of Visual Sudoku, using digits, see Fig. 1(a). We also considered Visual Sudoku, using letters. The CLR-DRNets model for the Visual Sudoku is illustrated in Section 3 and Fig. 2. We prepare two training sets for the digit Visual Sudoku: one contains only noisy training data (denoted as noisy dataset), i.e., the digits of the Sudoku are images, another consists of non-noisy training data (denoted as non-noisy dataset), i.e., the value of digit/letter images are known. Note the training data for CLR-DRNets **do not include** the solution of the Sudoku. The noisy dataset consists of seven difficulty levels: 51, 46, 42, 36, 31, 25 and 20 hints. We generate 10,000 Visual-Sudoku instances for each difficulty level. For the non-noisy dataset, we generate 100,000 standard Sudoku instances with a uniform distribution of 18 hints to 25 hints. There is no difficulty imbalance issue for non-noisy dataset, so we do not separate the dataset based on its difficulty.

We explain our training and test settings here. For the *restarts* scheme, the size of the model pool M is 100, i.e. we select and save the top 100 models in terms of the validation performance. And the restart gap g is set to 10 steps, i.e., we move to the next model after optimizing one model 10 steps. The metric ϕ is whether the input Sudoku is solved. Note that we do not assign partial credits for the Sudoku solution. The loss we optimize in the training phase and the test phase is $L_{\text{Sudoku}} + \lambda_3 L_{\text{recon}}$. The distance metric we used for L_{recon} is \mathcal{L}_1 loss. We use Adam as the optimizer, and the learning rate is $3e-4$ (including for restarts (test)). The weight scalars $\lambda_1 = 1.0$, $\lambda_2 = 0.01$ (for training and testing) and $\lambda_3 = 0.001$.

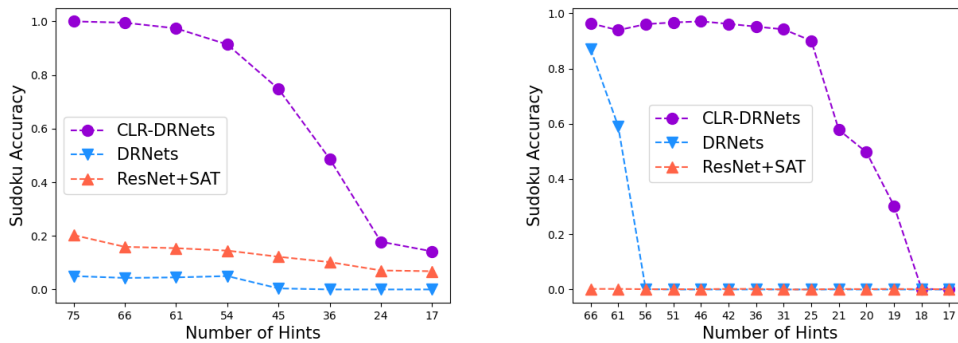
We compare our model with DRNets [4], SATNET [22], a higher-order constraint optimization based approach [18] (referred to as HOCOP/HOCOP (C) where C refers to do the model calibration on the validation set) and a Cost Function Network based approach [3] (referred to as CFN). The learning settings of these methods are different. SATNET and

CFN both require the solutions (labels) of the data. HOCOP and CLR-DRNets do not require the solutions, however, HOCOP leverages a high-efficient Sudoku solver and this kind of solver may not exist for other problems. **CLR-DRNets use the fewest supervision (no solutions (labels)) to learn to solve the problem and generalize to unseen data.** We use two test datasets. The first dataset (denoted as $D_{\text{avg-36hints}}$) is SATNET's dataset (CNF and HOCOP also use this dataset) consisting of Sudokus with a mean of 36 hints, and the second dataset (denoted as $D_{17\text{hints}}$) is formed of 1000 Sudoku with 17 hints. The CLR-DRNets model is trained on the noisy dataset (*joint training*) to solve the $D_{\text{avg-36hints}}$ and on the non-noisy dataset (*separate training*) to solve the $D_{17\text{hints}}$.

■ **Table 1** The test set Sudoku accuracy on the Digit Visual Sudoku task for different approaches.

Dataset	CLR-DRNets	DRNets	SATNET	CFN	HOCOP	HOCOP (C)	ResNet+SAT
$D_{\text{avg-36hints}}$	0.996	0.81	0.632	0.763	0.929	0.996	0.821
$D_{17\text{hints}}$	0.88	0	0	NA	NA	NA	0.918

CLR-DRNets outperform SATNET, DRNets, HOCOP and CFN on both datasets (see Table. 1) and CLR-DRNets do not require the labels. ResNet+SAT denotes a sequential coupling of ResNet with a SAT Solver, i.e., passing the digit classification of the input handwritten Sudoku, using ResNet, as input to a modern SAT-Solver to get the final results. For the dataset with an average of 36 hints, CLR-DRNets significantly surpass the performance of ResNet+SAT since CLR-DRNets can correct some perception mistakes guided by the Sudoku rules. Note though that CLR-DRNets do not outperform ResNet+SAT on 17 hint instances, given the high digit accuracy of ResNet for few number of hints, which results in high probability of perfect recognition of the input Sudoku. Nevertheless, often in many tasks we may not get an almost-perfect perception model, which is the case of e.g., the letter Visual Sudoku (the classifier accuracy is only 97.5%). In fact, when tested on the letter Visual Sudokus (trained using *separate training* on the non-noisy dataset), CLR-DRNets consistently outperform ResNet+SAT and DRNets for all the instances (see Fig. 4, left panel).



■ **Figure 4** Accuracy of CLR DNRNets, DRNets and ResNet+SAT on Letter Visual Sudoku (left) and Visual Mixed Sudoku (right).

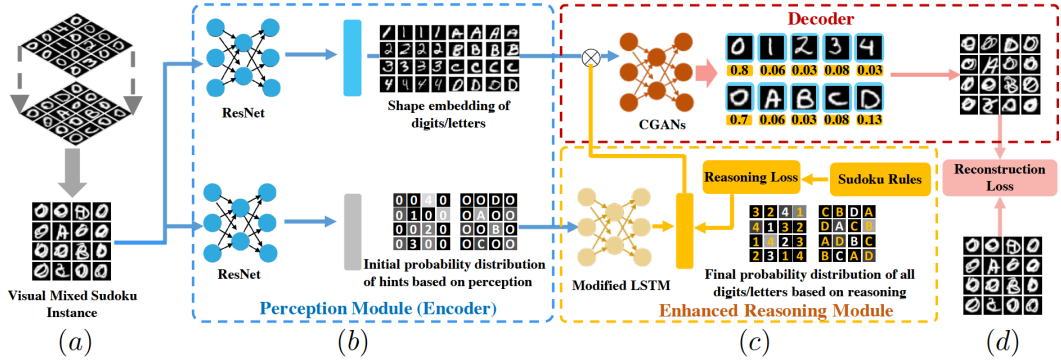


Figure 5 (a) A visual mixed Sudoku instance. We use a max operator to mix two visual Sudokus. (b) The perception module (encoder) of CLR-DRNets. The top blue rectangle is the latent space capturing the shape information of all possible digits and letters per cell. The bottom gray rectangle is the initial estimation of the input visual mixed Sudoku instance, only based on perception. Here we employ a heat map to represent the predicted confidence of each cell. (c) The enhanced reasoning module consists of a modified LSTM model. It takes the initial estimation as the input, constrained by the relaxed Sudoku rules loss function, computing the completion results. (d) The decoder leverages two pre-trained cGANs to generate all the possible digit and letter images w.r.t the shape embeddings of (b) per cell. Then we can reconstruct each cell guided by the completion results of (c). (We use 4×4 Sudokus for easier visualization. All our experiments are with 9×9 Sudokus.)

4.2 Visual Mixed Sudoku

The Visual Mixed Sudoku task is computationally more challenging than Visual Sudoku, requiring a tighter combination between perception and reasoning. We mix (using max operator) two Visual Sudokus (see Fig. 1(c)) to form a Visual Mixed Sudoku instance, offsetting the digits and letters to the top left and bottom right direction by two pixels. One Visual Sudoku consists of digits 0 to 9 (0 denotes the empty cell). The other one is formed of letters A to J (J denotes the empty cell). All the digit images are sampled from MNIST [15] and all the letter images are sampled from EMNIST [5]. The sizes of the training set, with different difficulty, and test set are all 10,000. We employ the *joint training* strategy to train the CLR-DRNets model since the pre-trained classifier can only achieve an accuracy of around 90% (due to the complexity of classifying mixed digits and images). The CLR-DRNets model for Visual Mixed Sudoku is similar to that for Visual Sudoku and its framework is illustrated in the Fig. 5. The perception module (see Fig. 2(b)) consists of two ResNet-18 [13] models. One model is used to generate the shape embeddings for all possible letters and digits per cell. The other model is employed to generate the initial probability distribution of each cell. The reasoning module (see Fig. 2(c)) consists of a modified LSTM, the same as for Visual Sudoku. The initial probability distributions are fed into the modified LSTM to compute the final probability distribution. The decoder (see Fig. 2(d)) consists of two cGANs (G_d for digits and G_l for letters). The shape embeddings are fed into two cGANs to generate all the possible letter and digit images of each cell. We formally define how we reconstruct the input images. For each cell, denote the shape embedding as $z_{d,0} \cdots z_{d,9}$ for digits, $z_{l,A} \cdots z_{l,J}$ for letters and denote the final probability distribution as $P_{d,0} \cdots P_{d,9}$ for digits, $P_{l,A} \cdots P_{l,J}$ for letters. We reconstruct the input image as: $\max(\sum_{i=0}^9 P_{d,i} G_d(z_{d,i}), \sum_{i=A}^J P_{l,i} G_l(z_{l,i}))$. We use L_1 loss as our reconstruction loss (L_{recon}).

We explain our training and test settings here. For training, we separately train two cGANs with part of the MNIST/EMNIST dataset, i.e. the digit and letter images used to pre-train the cGAN have no intersection with the visual Sudokus' images. The other parts

are jointly trained through optimizing the loss function $L = L_{\text{Sudoku}} + \lambda_3 L_{\text{recon}}$. To tackle the different difficulty of the perception and reasoning part, we start with easy Visual Mixed Sudoku instances, i.e., with 75 (out of 81) hints. And the following curriculum tasks are designed as 66, 61, 56, 51, 46, 42, 36, 31 and 25 hints. For each difficulty level, we generate 10,000 Mixed-Visual-Sudoku instances. From 25 hints to 24 hints, we observe a phase transition property of the combinatorial search problem so we can only train our model using instances with 25 hints. But surprisingly, the model can still generalize to harder cases (less than 25 hints). In our experiments, we always mix two Visual Sudokus with the same number of hints. For the *restart* scheme, the size of the model pool M is 20. The restart gap is 100. The metric ϕ is to check the validity of the Sudokus' solution. The loss function we optimized for the *restart* scheme (test) is the same as for training, i.e. $L_{\text{Sudoku}} + \lambda_3 L_{\text{recon}}$. Now we have 4 weight scalars for the Sudoku loss equation, denoted as $\lambda_{1,d}$, $\lambda_{2,d}$, $\lambda_{1,l}$ and $\lambda_{2,l}$, where d and l refer to digit and letter. We set them as $\lambda_{1,d} = \lambda_{1,l} = 1.0$, $\lambda_{2,d} = 0.01$, $\lambda_{2,l} = 0.02$. We up-weight the λ_2 for letters Sudoku in that recognizing letters is usually harder than digits. λ_3 is set as 0.005. These parameters are the same for training and testing. The optimizer we selected is Adam and the learning rate is $3e - 4$ in the training and $1e - 4$ in the *restarting* (test) phase.

CLR-DRNets' Sudoku accuracy is significantly higher than DRNets' and also better than ResNet+SAT (see Fig. 4). De-mixing is very challenging for standard deep learning methods. DRNets can only solve some easy instances (e.g., 66 hints). When the number of hints decreases, the difficulty of the reasoning task increases comparatively to the perception task, making it more challenging (or infeasible) for DRNets to learn the task.

4.3 Ablation Studies

The results above show that CLR-DRNets significantly outperform the baselines, due to (1) the curriculum learning (see Alg. 1) and (2) the *restart* scheme (see Alg. 2). We conducted ablation studies to analyze the contribution of the two factors and the results are showed in the table 2. In both tasks, curriculum learning contributes the most to the improvement. *Restarts* also play an important role, especially for challenging instances (17 hint Visual Sudoku and 20 hints Visual Mixed Sudoku). We postulate that the 25 hint case is not very hard, therefore a single model suffices.

■ **Table 2** The test set Sudoku accuracy performance on different tasks, we report the proportion to the CLR-DRNets' results. r refers to the *restart* scheme and c refers to the curriculum learning.

Task	CLR-DRNets	w/o c	w/o r	w/o r+c
Visual Sudoku (17-hints)	1.0	0.237	0.450	0.007
Visual Mixed Sudoku (20-hints)	1.0	0.005	0.955	0.004
Visual Mixed Sudoku (25-hints)	1.0	0.009	0.472	0

4.4 Standard 17-hints Sudoku

We also evaluate CLR-DRNets on learning to solve standard Sudokus (i.e., hint values known), supervised only by the Sudoku rules (no labeled data). We unsupervised train it on standard Sudoku task by simply optimizing the loss function, L_{Sudoku} . The model architecture is exactly the same as the reasoning module of Visual Sudoku, i.e. a modified LSTM. We train our model on 100,000 Sudoku with a uniform distribution of 18 to 25 hints.

And test on 1,000 Sudoku with 17 hints. We compared CLR-DRNets against RRN [20], which is a totally supervised method also leveraging the Sudoku rules to design the model architecture.

■ **Table 3** Sudoku accuracy for solving 17-hint standard Sudokus.

	CLR-DRNets	RRN
Sudoku Accuracy	0.912	0.64

From Table 3, we can see that CLR-DRNets outperform RRN largely. Here RRN means that we train RRN model on our training set (18–25 hints Sudoku). The reason RRN does not perform as well as their paper is that we never let RRN see the 17 hints Sudoku in the training phase, so RRN cannot generalize as CLR-DRNets does to the 17-hints case. The optimizer is Adam with learning rate $1e - 3$. We use a learnable embedding for each digit 0 to 9 with dimension 10. The model pool M for *restart* scheme is 100 and the restart gap g is 10. And the learning rate during the *restart* scheme is also $1e - 3$.

5 Conclusions

We introduce CLR-DRNets, a curriculum-learning-with-restarts framework for DRNets, along with an enhanced reasoning module. We demonstrate the CLR-DRNets’ effectiveness on challenging single-player visual combinatorial games, achieving state-of-the-art performance with weak supervision from prior knowledge (domain rules).

References

- 1 Brandon Amos and J Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. *arXiv preprint*, 2017. [arXiv:1703.00443](https://arxiv.org/abs/1703.00443).
- 2 Armin Biere and Andreas Fröhlich. Evaluating CDCL restart schemes. In Daniel Le Berre and Matti Järvisalo, editors, *Proceedings of Pragmatics of SAT 2015, Austin, Texas, USA, September 23, 2015 / Pragmatics of SAT 2018, Oxford, UK, July 7, 2018*, volume 59 of *EPiC Series in Computing*, pages 1–17. EasyChair, 2018.
- 3 Céline Brouard, Simon de Givry, and Thomas Schiex. Pushing data into cp models using graphical model learning and solving. In *International Conference on Principles and Practice of Constraint Programming*, pages 811–827. Springer, 2020.
- 4 Di Chen, Yiwei Bai, Wenting Zhao, Sebastian Ament, John Gregoire, and Carla Gomes. Deep reasoning networks for unsupervised pattern de-mixing with constraint reasoning. In *International Conference on Machine Learning*, pages 1500–1509. PMLR, 2020.
- 5 Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. Emnist: Extending mnist to handwritten letters. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2921–2926. IEEE, 2017.
- 6 Travis Dick, Eric Wong, and Christoph Dann. How many random restarts are enough. Technical report, Technical report, 2014.
- 7 Dieqiao Feng, Carla P. Gomes, and Bart Selman. Solving hard AI planning instances using curriculum-driven deep reinforcement learning. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 2198–2205. ijcai.org, 2020. [doi:10.24963/ijcai.2020/304](https://doi.org/10.24963/ijcai.2020/304).
- 8 Matteo Gagliolo and Jürgen Schmidhuber. Learning restart strategies. In *IJCAI*, pages 792–797, 2007.

- 9 Artur Garcez, Tarek R Besold, L d Raedt, Peter Fo ldiak, Pascal Hitzler, Thomas Icard, Kai-Uwe Ku hnberger, Luis C Lamb, Risto Miikkulainen, and Daniel L Silver. Neural-symbolic learning and reasoning: contributions and challenges, 2015.
- 10 Carla P Gomes, Bart Selman, Henry Kautz, et al. Boosting combinatorial search through randomization. *AAAI/IAAI*, 98:431–437, 1998.
- 11 Akhilesh Gotmare, Nitish Shirish Keskar, Caiming Xiong, and Richard Socher. A closer look at deep learning heuristics: Learning rate restarts, warmup and distillation. *arXiv preprint*, 2018. [arXiv:1810.13243](#).
- 12 Guy Hacohen and Daphna Weinshall. On the power of curriculum learning in training deep networks. *arXiv preprint*, 2019. [arXiv:1904.03626](#).
- 13 Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- 14 Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- 15 Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- 16 Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. In *Advances in Neural Information Processing Systems*, pages 3749–3759, 2018.
- 17 Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint*, 2014. [arXiv:1411.1784](#).
- 18 Maxime Mulamba, Jayanta Mandi, Rocsildes Canoy, and Tias Guns. Hybrid classification and reasoning for image-based constraint solving. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 364–380. Springer, 2020.
- 19 Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *arXiv preprint*, 2020. [arXiv:2003.04960](#).
- 20 Rasmus Palm, Ulrich Paquet, and Ole Winther. Recurrent relational networks. In *Advances in Neural Information Processing Systems*, pages 3368–3378, 2018.
- 21 Zhipeng Ren, Daoyi Dong, Huaxiong Li, and Chunlin Chen. Self-paced prioritized curriculum learning with coverage penalty in deep reinforcement learning. *IEEE transactions on neural networks and learning systems*, 29(6):2216–2226, 2018.
- 22 Po-Wei Wang, Priya Donti, Bryan Wilder, and Zico Kolter. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *International Conference on Machine Learning*, pages 6545–6554, 2019.
- 23 Daphna Weinshall, Gad Cohen, and Dan Amir. Curriculum learning by transfer learning: Theory and experiments with deep networks. *arXiv preprint*, 2018. [arXiv:1802.03796](#).
- 24 Fan Yang, Zhilin Yang, and William W Cohen. Differentiable learning of logical rules for knowledge base reasoning. In *Advances in Neural Information Processing Systems*, pages 2319–2328, 2017.

A Appendix

DRNets’ continuous relaxation for discrete constraints

Here we introduce more formally how DRNets apply continuous relaxation to the discrete constraints. The basic idea is to employ entropy to model the discrete constraints. For example, in Sudoku, we require that each row, col, and block, is filled with different digits (denoted as AllDiff constraints). Then for each cell, we use a probability distribution over all the possible digits to represent the estimation of one cell. We add all the cells’ probability

within one row, col or block, consider the best case, the summation vector should be an all-one vector, which also means it has the highest entropy. Thus, to encourage the model's output to satisfy the AllDiff constraints, we maximize the entropy defined above. More formally, use $e_i, i = 1 \dots n$ to represent the discrete variable and $p_i, i = 1 \dots n$ for the corresponding probability distribution. Then to represent $e_i \neq e_j \forall i \neq j$, we can maximize the function, $H(\sum_{i=1}^n p_i)$,

where H refers to the entropy. We also want to force the probability distribution to converge to one point since each distribution actually refers to one single point (denoted as Cardinality constraint). We can minimize the relaxed loss function to achieve this, i.e., $\sum_{i=1}^n H(p_i)$

The last discrete constraint we usually use is to select k items from n candidates. This is a little bit tricky, but we can use a hinge style loss to do that. If we want to select exactly k items from $e_i, i = 1 \dots n$, we can optimize this relaxed loss function, $\max(H(\sum_{i=1}^n p_i) - \log(k), 0)$. The idea here is simple, if the entropy probability distribution summation is $\log(k)$, supported by the Cardinality constraint, we actually selected k items from n candidates.

Once we have these powerful relaxations, we can convert a constrained optimization problem to an unconstrained optimization problem. Consider we want to optimize the objective function L under the constraint ϕ . Then we can firstly continuously relax the constraint ϕ to ψ , and optimize the loss function L_{relax} ($L_{\text{relax}} = L + \lambda\psi$), which is approximately equal to solving the original constraint optimization problem.

An Interval Constraint Programming Approach for Quasi Capture Tube Validation

Abderahmane Bedouhene ✉🏠

LIGM, Ecole des Ponts ParisTech, Université Gustave Eiffel, CNRS, Marne-la-Vallée, France

Bertrand Neveu ✉

LIGM, Ecole des Ponts ParisTech, Université Gustave Eiffel, CNRS, Marne-la-Vallée, France

Gilles Trombettoni ✉

LIRMM, Université de Montpellier, CNRS, France

Luc Jaulin ✉

Lab-STICC, ENSTA-Bretagne, Brest, France

Stéphane Le Menec ✉

MBDA, Le Plessis Robinson, France

Abstract

Proving that the state of a controlled nonlinear system always stays inside a time moving bubble (or capture tube) amounts to proving the inconsistency of a set of nonlinear inequalities in the time-state space. In practice however, even with a good intuition, it is difficult for a human to find such a capture tube except for simple examples. In 2014, Jaulin et al. established properties that support a new interval approach for validating a quasi capture tube, i.e. a candidate tube (with a simple form) from which the mobile system can escape, but into which it enters again before a given time. A quasi capture tube is easy to find in practice for a controlled system. Merging the trajectories originated from the candidate tube yields the smallest capture tube enclosing it.

This paper proposes an interval constraint programming solver dedicated to the quasi capture tube validation. The problem is viewed as a differential CSP where the functional variables correspond to the state variables of the system and the constraints define system trajectories that escape from the candidate tube “for ever”. The solver performs a branch and contract procedure for computing the trajectories that escape from the candidate tube. If no solution is found, the quasi capture tube is validated and, as a side effect, a corrected smallest capture tube enclosing the quasi one is computed. The approach is experimentally validated on several examples having 2 to 5 degrees of freedom.

2012 ACM Subject Classification Applied computing → Operations research; Mathematics of computing → Ordinary differential equations; Mathematics of computing → Differential algebraic equations; Mathematics of computing → Interval arithmetic; Theory of computation → Constraint and logic programming

Keywords and phrases Constraint satisfaction problem, Interval analysis, Dynamical systems, Contractor

Digital Object Identifier 10.4230/LIPIcs.CP.2021.18

Funding This work was supported by the French Agence Nationale de la Recherche (ANR) [grant number ANR-16-CE33-0024].

Acknowledgements We also thank our colleagues, Alexandre Goldsztejn and Alessandro Colotti, for the exchange of ideas and their kind help on the experiments.



© Abderahmane Bedouhene, Bertrand Neveu, Gilles Trombettoni, Luc Jaulin, and Stéphane Le Menec; licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 18; pp. 18:1–18:17



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Many mobile robots such as wheeled robots, boats, or planes are described by differential equations. For this type of robots, it is difficult to prove some properties such as the avoidance of collisions with some moving obstacles. This is even more difficult when the initial condition is not known exactly or when some uncertainties occur.

A Graal would be to compute a *capture tube* (or equivalently a *positive invariant* tube [24]), i.e. a time moving “bubble” (a set-valued function associating to each time t a subset of \mathbb{R}^n) from which a feasible trajectory cannot escape. The definitions and properties of capture tubes have been studied by several authors [2, 4], but the algorithms for their computation are almost absent except in the linear case [33, 25, 11]. In the nonlinear case, approaches based on interval analysis [19, 31] or Lipschitz assumptions [32] have also been investigated, but the performances are poor if no propagation techniques are used. When time is discrete, efficient algorithms are given in [35], but they cannot be extended to robotics systems described by differential equations.

Instead, a satisfactory alternative is to present a candidate tube to a tool that could validate whether it is a capture tube or not. This validation problem can generally be transformed into proving the inconsistency of a constraint system by combining guaranteed integration and Lyapunov theory [26, 36]. Unfortunately, when the system dynamics is complex, even with a good intuition, it is difficult for a human to present a significant capture tube because of its irregular form.

Jaulin et al. proposed in [13] an original approach based on interval analysis. The idea is to validate a *quasi* capture tube, also called *periodic invariant set* [17], i.e. a candidate tube (with a simple form) from which the mobile system can escape, but into which it can enter again before a given time. Merging these trajectories with the candidate tube computes the smallest capture tube enclosing the quasi capture one. Jaulin et al. established properties that support this new approach, but the algorithms were not described and were validated only on a simple pendulum example with two degrees of freedom. Their approach worked in two steps, where the first one focused on the crosscut constraints (see Section 3) while the second step managed the other constraints.

The contribution presented in this paper is built upon those properties. Compared to Jaulin et al. approach, the solver follows a pure CSP approach expressing the quasi capture tube validation problem, where the domains are tubes defined recently in the Tubex-Codac library [28, 30]. After the background in Section 2, we formally define in Section 3 the quasi capture tube validation problem and its expression as a CSP. We then propose a Branch and Contract solver dedicated to this problem in Section 4 and show in Section 5 how it scales up on several problems from 2 to 5 state dimensions.

2 Background

We first provide some background about intervals, inclusion functions and contraction. We then briefly present how intervals can be used to handle dynamical systems.

2.1 Intervals

Contrary to numerical analysis methods that work with single values, interval methods can manage sets of values enclosed in intervals. Interval methods are known to be particularly useful for handling nonlinear constraint systems.

► **Definition 1** (Interval, box, box size/diameter). An interval $[x_i] = [\underline{x}_i, \overline{x}_i]$ defines the set of reals x_i such that $\underline{x}_i \leq x_i \leq \overline{x}_i$. \mathbb{IR} denotes the set of all intervals. A box $[\mathbf{x}]$ denotes a Cartesian product of intervals $[\mathbf{x}] = [x_1] \times \dots \times [x_n]$. The size, width or diameter of a box $[\mathbf{x}]$ is given by $\text{Diam}([\mathbf{x}]) \equiv \max_i(\text{Diam}([x_i]))$ where $\text{Diam}([x_i]) \equiv \overline{x}_i - \underline{x}_i$. The midpoint $\text{mid}([x_i])$ of $[x_i]$ is $\frac{\underline{x}_i + \overline{x}_i}{2}$.

Interval arithmetic [22] has been defined to extend to \mathbb{IR} the usual mathematical operators over \mathbb{R} . For instance, the interval sum is defined by $[x_1] + [x_2] = [\underline{x}_1 + \underline{x}_2, \overline{x}_1 + \overline{x}_2]$. When a function \mathbf{f} is a composition of elementary functions, an *inclusion function* $[\mathbf{f}]$ of \mathbf{f} must be defined to ensure a conservative image computation. There are several inclusion functions. The *natural* inclusion function of a real function f corresponds to the mapping of f to intervals using interval arithmetic. For instance, the natural inclusion function $[f]_N$ of $f(x) = x(x+1)$ in the domain $[x] = [0, 1]$ computes $[f]_N([0, 1]) = [0, 1] \cdot [1, 2] = [0, 2]$. Another inclusion function is based on an interval Taylor form [12].

Interval arithmetics can be used for solving the *numerical CSP* (NCSP), *i.e.* finding solutions to an NCSP network $P = (\mathbf{x}, [\mathbf{x}], \mathbf{c})$, where \mathbf{x} is an n -set of variables taking their real values in the domain $[\mathbf{x}]$ and \mathbf{c} is an m -set of numerical constraints using operators like $+$, $-$, \times , a^b , \exp , \log , \sin , *etc.* NCSP solvers, like Gloptlab [10] or IBEX [6] to name a few, follow a Branch and Contract method to solve an NCSP. The branching operation subdivides the search space by recursively bisecting variable intervals into two subintervals and exploring both sub-boxes independently. The combinatorial nature of this tree search is not always observed thanks to the *contraction* (filtering) operations applied at each node of the search tree. Informally, a contraction applied to an NCSP instance can reduce the variables domains without losing any solution.

A contractor used in this paper is the well-known **HC4-revise** [3, 21], also called *forward-backward*. This contractor handles a single numerical constraint and obtains a (generally non optimal [7]) contracted box including all the solutions of that constraint.

To contract a box w.r.t. an NCSP instance, the HC4 algorithm performs a (generalized) AC3-like propagation loop applying iteratively the HC4-Revise procedure on each constraint individually until a quasi fixpoint is obtained in terms of contraction.

CID-consistency [34] is a stronger consistency enforced on an NCSP. The CID algorithm calls its **VarCID** procedure on all the NCSP variables for enforcing the CID-consistency. **VarCID** splits a variable interval in k subintervals, and runs a contractor, such as HC4, on the corresponding sub-boxes. The smallest box including the k sub-boxes contracted is finally returned. The 3BCID contractor used in this paper uses a variant of the **VarCID** procedure.

2.2 Dynamical CSP and tubes

Intervals can also be used to handle dynamical systems that handle functional variables, also called *trajectories*.

A trajectory, denoted $\mathbf{x}(\cdot) = (x_1(\cdot), \dots, x_n(\cdot))$, is a function from $[t_0, t_f] \subset \mathbb{R}$ to \mathbb{R}^n . The input (argument) of $\mathbf{x}(\cdot)$ is named *time* in this article (and denoted \cdot or t) while the output (image) is called *state*.

Interval methods can compute trajectories as solutions of a *differential CSP* instance.

► **Definition 2** (Differential CSP). A *differential CSP network* is defined by $(\mathbf{x}(\cdot), [\mathbf{x}](\cdot), \mathbf{c})$, where $\mathbf{x}(\cdot)$ is a trajectory variable of domain $[\mathbf{x}](\cdot)$ and \mathbf{c} denotes the set of differential constraints between variables $\mathbf{x}(\cdot)$.

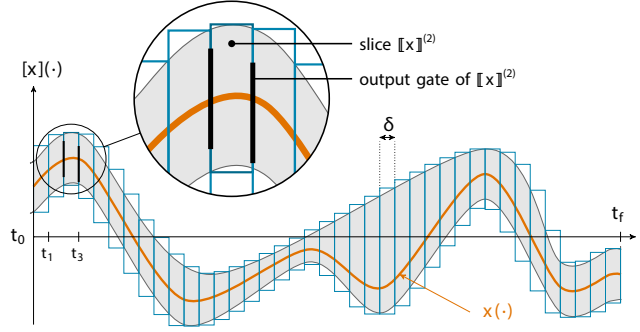
Solving a differential CSP instance consists in finding the set of trajectories in $[\mathbf{x}](\cdot)$ satisfying \mathbf{c} .

Domains of a differential CSP network are *tubes*, set-valued functions associating to each time t a subset of \mathbb{R}^n , on which we apply contraction and bisection operations.

► **Definition 3** (Tube [15]). A tube $[\mathbf{x}](\cdot) : [t_0, t_f] \rightarrow \mathcal{P}(\mathbb{R}^n)$ is an interval of two trajectories $[\underline{\mathbf{x}}(\cdot), \bar{\mathbf{x}}(\cdot)]$ such that $\forall t \in [t_0, t_f], \underline{\mathbf{x}}(t) \leq \bar{\mathbf{x}}(t)$. We also consider empty tubes that depict an absence of solutions.

A trajectory $\mathbf{x}(\cdot)$ belongs to the tube $[\mathbf{x}](\cdot)$ if $\forall t \in [t_0, t_f], \mathbf{x}(t) \in [\mathbf{x}](t)$.

Fig. 1 illustrates a one-dimensional tube $([t_0, t_f] \rightarrow \mathcal{P}(\mathbb{R}))$ enclosing a trajectory $x(\cdot)$.



■ **Figure 1** A one-dimensional tube $[\mathbf{x}](\cdot)$. Courtesy of S. Rohou). In grey enclosing a random trajectory $x(\cdot)$ depicted in plain line (orange). $[\mathbf{x}](\cdot)$ is an interval of two functions $[\underline{\mathbf{x}}(\cdot), \bar{\mathbf{x}}(\cdot)]$. The tube is numerically represented by a set of δ -width slices illustrated by blue boxes.

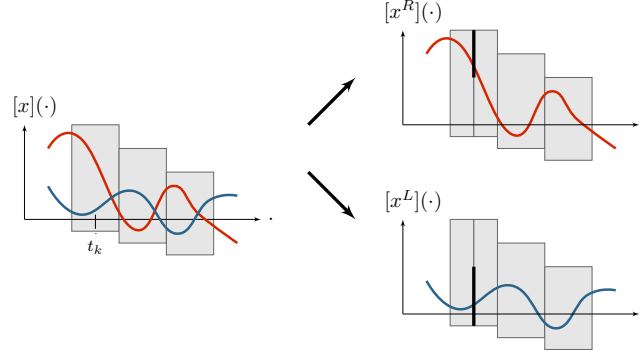
A tube is represented numerically by a set of boxes corresponding to temporal slices. More precisely, an n -dimensional tube $[\mathbf{x}](\cdot)$ with a sampling time $\delta > 0$ is implemented as a box-valued function which is constant for all t inside intervals $[k\delta, k\delta + \delta]$, $k \in \mathbb{N}$. The box $[k\delta, k\delta + \delta] \times [\mathbf{x}](t_k)$, with $t_k \in [k\delta, k\delta + \delta]$, is called the k^{th} slice of the tube $[\mathbf{x}](\cdot)$ and is denoted by $[\mathbf{x}]^{(k)}$. This implementation takes rigorously into account floating-point precision when building a tube: computations involving $[\mathbf{x}](\cdot)$ will be based on its slices, thus giving a reliable outer approximation of the solution set. The slices may be of same width as depicted in Fig. 1, but the tube can also be implemented with a customized temporal *slicing*. Finally, we endow the definition of a slice $[\mathbf{x}]^{(k)}$ with the *slice (box) envelope* (blue painted in Fig. 1) and two input/output *gates* $[\mathbf{x}](t_k)$ and $[\mathbf{x}](t_{k+1})$ (black painted) that are intervals of \mathbb{R}^n through which trajectories are entering/leaving the slice.

Once a tube is defined, it can be handled in the same way as an interval. We can for instance use arithmetic operations as well as function evaluations. If f is an elementary function such as sin, cos or exp, we define $f([\mathbf{x}](\cdot))$ as the smallest tube containing all feasible values: $f([\mathbf{x}](\cdot)) = [\{f(x(\cdot)) \mid x(\cdot) \in [\mathbf{x}](\cdot)\}]$.

The Branch & Contract algorithm presented in this paper makes choice points on tubes [28], defined as follows and illustrated by Fig. 2.

► **Definition 4** (Tube bisection). Let $[\mathbf{x}](\cdot)$ be a tube of a trajectory $\mathbf{x}(\cdot)$ defined over $[t_0, t_f]$. Let t_k be an instant in $[t_0, t_f]$, i a dimension in $\{1..n\}$, and $[x_i]$ the interval value of $[x_i](\cdot)$ at t_k . Let $mid(x_i)$ be $\frac{\underline{x_i} + \bar{x_i}}{2}$. The tube bisection (t_k, i) of $[\mathbf{x}](\cdot)$ produces two tubes $[\mathbf{x}^L](\cdot)$ and $[\mathbf{x}^R](\cdot)$ equal to $[\mathbf{x}](\cdot)$ except at time t_k , where $[\mathbf{x}_i^L] = [\underline{x_i}, mid(x_i)]$ and $[\mathbf{x}_i^R] = [mid(x_i), \bar{x_i}]$.

In practice, a bisection (t_k, i) is applied only to a gate of the tube. For the particular problem handled in this paper, t_k will always be t_0 .



■ **Figure 2** Illustration of a tube bisection at time t_k (courtesy of S. Rohou). A gate is created at t_k and the two sub-tubes $[x^L](\cdot)$ and $[x^R](\cdot)$ differ only by their new created sub-gate (in bold). Two (among an infinity) possible trajectories of the initial tube are separated by the bisection, one belonging to $[x^L](\cdot)$, the other belonging to $[x^R](\cdot)$.

There exist several types of differential constraints. The problem presented in Section 3 contains only well-known ordinary differential equations (ODEs).

► **Definition 5** (Ordinary differential equation – ODE). Consider $\mathbf{x}(\cdot) : [t_0, t_f] \rightarrow \mathbb{R}^n$, its derivative $\dot{\mathbf{x}}(\cdot) : [t_0, t_f] \rightarrow \mathbb{R}^n$, and an evolution function $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^n$, possibly non-linear. An ODE is defined by: $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t)$

This means that for all times t in the temporal domain $[t_0, t_f]$ the derivative of the function \mathbf{x} depends only on the state \mathbf{x} at time t and on the time t . An ODE can be used to define a well-known IVP differential system or an extension.

► **Definition 6** (IVP, interval IVP). The initial value problem (IVP) is defined by an ODE $\dot{\mathbf{x}}(\cdot) = \mathbf{f}(\mathbf{x}(\cdot))$ and an initial condition $\mathbf{x}(t_0) = \mathbf{x}_0$, where \mathbf{x}_0 is a constant in \mathbb{R}^n . In an interval IVP, the initial condition is bounded by a box, i.e. $\mathbf{x}(t_0) \in [\mathbf{x}_0]$.

The IVP is studied for hundreds years and can be solved by numerous numerical methods, e.g. the Euler method [5]. The interval IVP can be solved by interval analysis tools, such as VNODE [23], CAPD [14], COSY [27] and DynIbex [8]. These solvers are also called *Guaranteed Integration (GI)* solvers. GI solvers use different algorithms to rigorously integrate the initial information over time. In particular, the CAPD tool used in our solver combines a high-order interval Taylor form to integrate the state from an instant to a next one, and a step limiting the wrapping effect implied by interval calculation: it encloses the solution at gates by an envelope sharper than a box, such as rotated boxes [20].

3 Quasi Tube Capture Validation as a CSP

In automatic control, validation of *stability* properties of dynamical systems is an important and difficult problem [16]. A tube $\mathbb{G}(t)$ is *positive invariant* (or a *capture tube*) for a dynamic system $\mathbf{x}(\cdot)$ if all the possible trajectories of $\mathbf{x}(\cdot)$ remain in $\mathbb{G}(t)$ for ever, i.e. for every time in the temporal domain defined.

► **Definition 7** (Capture tube¹). Let S_f be a dynamic system defined by an ODE $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t)$. Let $\mathbb{G}(t)$ be a tube defined by an inequality $\{\mathbf{x}(t) \mid g(\mathbf{x}(t), t) \leq 0\}$, where $g : \mathbb{R}^n \times \mathbb{R} \rightarrow \mathbb{R}$ is a differentiable function w.r.t. \mathbf{x} and t .

Then:

$\mathbb{G}(t)$ is said to be a capture tube for S_f if: $\mathbf{x}(t_i) \in \mathbb{G}(t_i), \tau > 0 \implies \mathbf{x}(t_i + \tau) \in \mathbb{G}(t_i + \tau)$

Conditions can be checked to validate whether a given tube is a capture tube or not.

► **Theorem 1** (Cross-out conditions [13]). Let S_f be a dynamic system defined by $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t)$, and a tube $\mathbb{G}(t) = \{\mathbf{x}(t) \mid g(\mathbf{x}(t), t) \leq 0\}$. Consider the constraint system:

$$\begin{cases} (i) & \frac{\partial g(\mathbf{x}, t)}{\partial \mathbf{x}} \cdot \mathbf{f}(\mathbf{x}, t) + \frac{\partial g(\mathbf{x}, t)}{\partial t} \geq 0 \\ (ii) & g(\mathbf{x}, t) = 0 \end{cases} \quad (1)$$

If (1) is inconsistent (i.e., $\forall \mathbf{x}, \forall t \geq 0$, (1) has no solution), then $\mathbb{G}(t)$ is a capture tube.

The constraint system (1) describes the subset of S_f trajectories that escape from $\mathbb{G}(t)$. If this subset is empty, it means that $\mathbb{G}(t)$ is a capture tube.

In [13], Jaulin et al. highlighted that it is not easy for the user to define “by hand” a relevant capture tube of irregular form and propose rather to ask for a so-called *quasi* capture tube of simple form. Informally, some trajectories can escape from a quasi capture tube, but can enter into it again later, i.e. before a given horizon t_f . Such a trajectory satisfies the following constraints:

- $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t)$ ($\mathbf{x}(t)$ is a trajectory of S)
- $\exists t_0 \in [t_0], \mathbf{x}(t_0)$ satisfies (1) ($\mathbf{x}(t)$ exits from $\mathbb{G}(t)$ at $t_0 \in [t_0]$)
- $\exists t_{in} \in]t_0, t_f]$ s.t. $\mathbf{x}(t_{in}) \in \mathbb{G}(t_{in})$ ($\mathbf{x}(t)$ goes back inside $\mathbb{G}(t)$ at t_{in})

Instead of using these constraints directly, the idea of this paper is to propose a CSP expressing the “negation” of the quasi capture problem, and to detail a Branch & Contract method to solve it.

► **Definition 8** (CSP defining the quasi capture validation problem). Let S_f be a dynamic system defined by $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t)$, and a candidate tube $\mathbb{G}(t) = \{\mathbf{x}(t) \mid g(\mathbf{x}(t), t) \leq 0\}$.

The constraint network $N = (\mathbf{x}(\cdot), [\mathbf{x}(\cdot)], \mathbf{c})$ defines the quasi capture validation problem, where $\mathbf{x}(\cdot)$ describes the system living in the domain/tube $[\mathbf{x}(\cdot)]$, and \mathbf{c} includes the three following (vectorial) constraints:

$$\begin{cases} \dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t) & (\text{differential constraint}) \\ \exists t_0, \mathbf{x}(t_0) \text{ satisfies (1)} & (\text{cross out constraint}) \\ \forall t \in]t_0, t_f] \ g(\mathbf{x}(t), t) > 0 & (\text{escape constraint}) \end{cases}$$

The constraints model the fact that the system can escape from $\mathbb{G}(t)$ “for ever”, i.e. cannot go back in $\mathbb{G}(t)$ before t_f . If N is inconsistent, then it proves that $\mathbb{G}(t)$ is a quasi capture tube.

Furthermore, consider the trajectories that satisfy the cross out constraint but violate the escape constraint. It is straightforward to check that if the CSP has no solution, adding these trajectories to the candidate (quasi capture) tube builds a capture tube [13].

¹ For the sake of clarity, and because our application problems fall in this case, we restrict ourselves to the case where $\mathbb{G}(t)$ is defined by only one inequality. The corresponding cross out constraint system is slightly more complicated otherwise [13], but the solver presented in the next section also works on it.

4 Branch and Contract Algorithm

In this section, we describe a branch and contract algorithm for solving the differential CSP defined above. More precisely, Algorithm 1 computes a set *OutList* of tubes including all the system trajectories that escape from the candidate tube $\mathbb{G}(t)$ “for ever”, i.e. at a time greater than t_0 and remaining outside $\mathbb{G}(t)$ until t_f .

4.1 Main algorithm

The initial domain *initTube* is $[t_0, t_f] \times [x]$, where $[x]$ is a big or infinite box initializing the state variables. The other input parameters are the candidate capture tube $\mathbb{G}(t)$, a precision parameter on the time (*timestep*) and vectorial parameters ϵ_{start} and ϵ_{min} , that specify the diameters of all variables at the initial gate. They are detailed further.

Algorithm 1 Branch and contract.

```

1 Input ( $\mathbb{G}(t)$ , initTube,  $t_0$ ,  $t_f$ , timestep,  $\epsilon_{start}$ ,  $\epsilon_{min}$ )
2 Output (OutList : list of solution tubes ; UndeterminedList : list of “small” tubes
   still undetermined)
3 tubes  $\leftarrow$  {initTube}
4 while (tubes  $\neq \emptyset$ ) do
5   tube  $\leftarrow$  Pop(tubes)
6   (ContractionResult, tube)  $\leftarrow$  Contraction(tube, S,  $\mathbb{G}(t)$ ,  $t_0$ ,  $t_f$ , timestep,  $\epsilon_{start}$ )
7   if (ContractionResult = out) then
8     | OutList  $\leftarrow$  OutList  $\cup$  {tube}
9   else if (ContractionResult = undetermined) then
10    | if Diam(tube( $t_0$ ))  $\leq \epsilon_{min}$  then
11      | UndeterminedList  $\leftarrow$  UndeterminedList  $\cup$  {tube}
12    | else
13      | (tubeleft, tuberight)  $\leftarrow$  Bisect(tube, bisectionStrategy)
14      | tubes  $\leftarrow$  {tubeleft}  $\cup$  {tuberight}  $\cup$  tubes
15    | end
16  else
17    | /* ContractionResult = in: Nothing to do : tube is discarded because its
18      | trajectories all enter inside  $\mathbb{G}(t)$  at an instant in  $[t_0, t_f]$  */
19  end
20 end

```

Algorithm 1 follows a tree search that combinatorially subdivides the initial domain *initTube* into smaller tubes, in depth-first order. At each node of the search tree handling a *tube*, a contraction is achieved using the three types of constraints detailed above. The function **Contraction** (Line 6) returns a contracted tube and a status *ContractionResult* associated to it. *tube* can become empty (and *ContractionResult* = in) if **Contraction** could prove that the tube is entirely inside $\mathbb{G}(t)$ at an instant between t_0 and t_f (see Lines 16–18). A second case occurs when *tube* has been detected outside $\mathbb{G}(t)$ after a time and until t_f (Line 7). It is not useful to subdivide *tube* further because all the trajectories inside *tube* are solutions. Therefore *tube* is stored in *OutList*. The last case corresponds to an internal node of the search tree and occurs when the contraction cannot decide one of the cases “in” or “out” above (Line 9). If *tube* is sufficiently large (Line 12), the branching operation bisects

$tube$ in two sub-tubes $tube_{left}$ and $tube_{right}$ and pushed them in front of $tubes$ (depth first order). The tube bisection is performed at the first gate (at t_0) because one has the most information at this time (cross out conditions hold). Note it is sufficient to perform all the bisections at the same time because with an ODE an “instanciation” at one time allows one to deduce the trajectory perfectly.

No more bisection is achieved if the $tube$ size has reached a given precision ϵ_{min} , and $tube$ is stored in a list of “undetermined” tubes (Line 11). Algorithm 1 stops when $tubes$ is empty. If $OutList$ and $UndeterminedList$ are empty, then $\mathbb{G}(t)$ is a quasi invariant tube for the system S .

We detail in Algorithm 2 the different contractors applied to the current $tube$. $tube$ is first contracted by the cross out constraints (Line 3). **CrossoutContraction** contracts $tube$ at time t_0 according to the cross out constraints. It calls the state-of-the-art contractors HC4 [3] and 3BCID [18, 34] on the cross out constraint subsystem (see Section 5 describing the experiments).

With the call to **ODEEvalContraction** (Line 6), we then proceed with the contraction of the differential (ODE) constraint and the escape constraint. Note that this contraction procedure is run only under a given level of the search tree, where, for each dimension, the tube diameter at t_0 is lower than the user parameter ϵ_{start} . Indeed, this differential contraction during the time window $[t_0, t_f]$ is costly and needs a relatively small input box (initial condition) to efficiently contract $tube$, with the help of guaranteed integration.

■ **Algorithm 2** Function **Contraction** called by Algorithm 1.

```

1 Function Contraction( $S, \mathbb{G}(t), tube, t_0, t_f, timestep, \epsilon_{start}$ )
2    $tube \leftarrow \text{CrossOutContraction}(tube, S, \mathbb{G}(t))$ 
3   if ( $tube = \emptyset$ ) then
4     |  $ContractionResult \leftarrow \text{in}$ 
5   else if ( $Diam(tube(t_0)) < \epsilon_{start}$ ) then
6     |  $ContractionResult \leftarrow \text{ODEEvalContraction}(S, tube, \mathbb{G}(t), t_0, t_f, timestep)$ 
7   else
8     |  $ContractionResult \leftarrow \text{undetermined}$ 
9   end
10  return ( $ContractionResult, tube$ )
11 end

```

4.2 Differential contraction

White box differential contractors, e.g. the **ctcDeriv** and **ctcEval** contractors available in the TUBEX/CODAC free library [29], could be used to contract $tube$ w.r.t. the ODE and escape constraints.

Instead, for performance reasons, we preferred to exploit a state-of-the-art guaranteed integration (GI) tool, like VNODE-LP [23] or CAPD [14], to benefit from its optimized internal representations. The corresponding method is described in Algorithm 3.

The **ODEEvalContraction** function contracts $tube$ by integrating the ODE from t_0 to t_f using the CAPD GI solver. The function **GI_Simulation** (Line 5) calls the GI solver with the interval initial value $tube(t_i)$, the $tube$ gate at time t_i . The GI generally needs to construct several gates before reaching t_f , and **GI_Simulation** allows one to incrementally build the next *slice* between t_i and a computed time t_{i+1} . By doing this integration, the

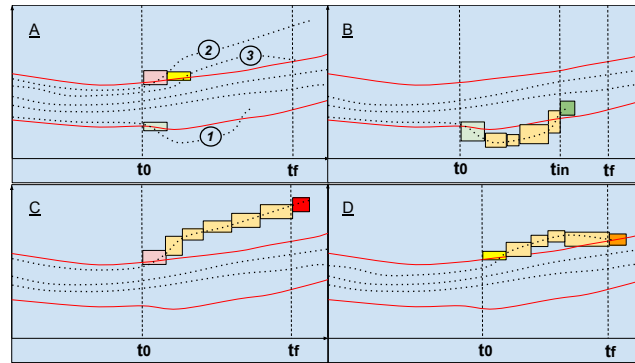
■ **Algorithm 3** Function `ODEEvalContraction` called by Algorithm 2.

```

1 Function ODEEvalContraction( $S, tube, \mathbb{G}(t), t_0, t_f, timestep$ )
2    $t_i \leftarrow t_0$ 
3    $t_{out} \leftarrow \infty$ 
4   repeat
5      $(slice, t_{i+1}) \leftarrow \text{GI\_Simulation}(S, tube(t_i), t_i, t_f)$ 
6      $(ContractionResult, t_{out}) \leftarrow \text{GI\_Eval}(slice, \mathbb{G}(t), timestep, t_i, t_{i+1}, t_{out})$ 
7      $tube[t_i, t_{i+1}] \leftarrow tube[t_i, t_{i+1}] \cap slice$ 
8      $t_i \leftarrow t_{i+1}$ 
9   until  $(t_i = t_f)$  or  $(ContractionResult = \text{in})$ 
10  if  $ContractionResult = \text{in}$  or  $t_{out} \neq \infty$  then
11    return  $ContractionResult$ 
12  else
13    return undetermined
14  end
15 end

```

GI solver builds an associated high-order Taylor polynomial that can be evaluated rapidly at any gate or subslice inside $[t_i, t_{i+1}]$. This is the task achieved by `GI_Eval`. Without detailing, `GI_Eval` splits $[t_i, t_{i+1}]$ into contiguous subslices of (time) size *timestep* and tests whether *tube* during the studied subslice satisfies the escape (from $\mathbb{G}(t)$) constraint or not. In the latter case, the integration is interrupted (Algorithm 3 stops) and *ContractionResult* is set to `in`. The whole *tube* is rejected. If a subslice satisfies the escape constraint, t_{out} is used to memorize the first instant where it occurs. If $t_{out} = \infty$, then t_{out} is set to t_i . If a subsequent subslice evaluation does not return `out`, then t_{out} is set back to ∞ . Indeed, recall that a solution tube must satisfy the escape constraint in all times from t_{out} to t_f . When t_f is reached, only two cases are still possible. Either *tube* has escaped from $\mathbb{G}(t)$ at t_{out} until t_f (a solution), or *tube* has intersected $\mathbb{G}(t)$ at some instants, including t_f . In that case, we cannot conclude and the result of the contraction will be `undetermined`. Figure (3) summarizes the different cases described above.



■ **Figure 3** Different tubes built by the solver. Three particular trajectories (1), (2) and (3) are highlighted in the figure. **A:** First slice satisfying the cross out constraints corresponding to the three trajectories leaving the tube $\mathbb{G}(t)$. **B:** A tube enclosing (1) is integrated and is getting inside $\mathbb{G}(t)$. **C:** A tube enclosing (2) is escaping from $\mathbb{G}(t)$. **D:** An undetermined tube enclosing (3): the algorithm cannot conclude.

Another possible case not described in the pseudo-code is when `GI_Simulation` fails to compute a part of the simulation. This result is equivalent to the `undetermined` result since the algorithm is not able to conclude if the *tube* goes inside $\mathbb{G}(t)$ or not. The choice of ϵ_{start} has a significant impact on the frequency of this “pathological” case (see experiments).

4.3 Discussion

Algorithm 1 provides two main answers. The favorable case is when the solver returns no solution: `OutList` and `Underdeterminedlist` are empty. The algorithm is correct and guarantees that $\mathbb{G}(t)$ is a quasi capture tube. Furthermore, as a side effect, by merging with $\mathbb{G}(t)$ all the `in` tubes rejected by the algorithm, we can build the smallest (i.e., inclusion-wise minimal) capture tube including the quasi capture tube. The second case occurs when the solver computes a non empty `OutList` or `Underdeterminedlist`. This corresponds generally to the computation of a non quasi capture tube but, theoretically, it is possible that a trajectory could enter inside $\mathbb{G}(t)$ after t_f , before t_{out} (`OutList` \neq `emptyset`), or during the undetermined temporal slices. In this sense, the solver is not complete while these numerical issues occur rarely provided t_f is large enough (according to the command of the system), and the precision size ϵ_{min} is sufficiently small.

5 Experiments

The current section presents some results provided by an implementation of Algorithm 1 that significantly improves a first code called `Bubbibex` and written to validate the pendulum problem [1]. This new `Bubbibex` is implemented in C++. It uses the `IBEX` library [6] with the `HC4` [3] and `3BCID` [18, 34] contractors for propagating the cross out conditions constraints. It also uses the `CAPD/DynSys` library for the differential contractor based on guaranteed integration [14] and the `Tubex/CODAC` library for tube structure [29].

Experiments have been carried out using an Intel(R) Xeon(R) CPU E3-1225 V2 at 3.20GHz. In each experiment, see Table [1], we highlight the results obtained by the solver when we tried different values for one so-called observed parameter (ϵ_{start} or bubble radius or etc.).

These responses include the running time of each experiment (CPU-Time) in second, and the number of computed tubes corresponding to the leaves of the search tree: “In” for tubes getting inside $\mathbb{G}(t)$, “Und” for undetermined tubes and “Out” for tubes staying out of $\mathbb{G}(t)$ at t_f . These numbers are reported in the tables presenting the results of each experiment.

The simulation time of each experiment is at most $t_f = 100$ with $timestep = 0.01$. The bisection strategy used is the `Maximum Diam Ratio`, selecting the variable $[x_i]$ with the greatest ratio $Diam([x_i])/\epsilon_i$.

► **Remark 2.** Rewriting a non autonomous ODE as an autonomous ODE adds the temporal variable “t” to the state variables, increasing the dimension of the problem by 1. As a result, the dimension of the vectorial parameters ϵ_{start} and ϵ_{min} might increase if the domain of the temporal variable “t” defined by $[t_0]$ is not a degenerate interval (i.e. $[t_0 < \overline{t_0}]$).

5.1 Pendulum

$$P : \begin{cases} \dot{x} = y \\ \dot{y} = -\sin(x) - \rho \cdot y \end{cases} \quad (2)$$

■ **Table 1** Characteristics of the different experiments.

Problem	Type	State variables	Bubble
Pendulum	Non Linear	2	Static
2D Linear system	Linear	2	Dynamic
Tracking	Linear	2 and 3	Static and Dynamic
Pursuit game	Non Linear	3 and 5	Dynamic

Let P be a dynamical system describing the motion of a pendulum, where x is the angular position, y is the angular velocity and $\rho = 0.15$ the constant friction coefficient of the pendulum. We want to find a quasi capture tube for the system P .

■ **Table 2** Parameters of the pendulum experiment.

First gate	Bubble	r_0	Observed parameter
$x, y \in [-10, 10]$	$x^2 + y^2 - r_0^2 \leq 0$	1	ϵ_{start}

When $\epsilon_{start} = \{1, 1\}$ (Line 1 of Table 3), the differential contractor is not able to successfully contract the tube. This is due to a large initial condition that prevents the guaranteed integration from computing a solution and leads the solver to bisect the initial gate of the tube before reaching the right precision. Having a good intuition on the parameter ϵ_{start} (Line 2 of Table 3) can improve the efficiency of the method. The CSP has no solution, the studied bubble is a quasi capture tube.

■ **Table 3** Results for pendulum system.

ϵ_{start}	ϵ_{min}	In	Und	Out	CPU
$\{1, 1\}$	$\{0.1, 0.1\}$	6	0	0	72.2
$\{0.5, 0.5\}$	$\{0.1, 0.1\}$	6	0	0	0.00734

5.2 2D linear system

$$R: \begin{cases} \dot{x} = u_1 \\ \dot{y} = u_2 \end{cases} \quad (3)$$

Let R be a robot described by the linear dynamical system (3) such that (x, y) is the position and $u_1 = -x + t$, $u_2 = -y$ the controllers.

We want the robot to stay inside a dynamic bubble.

For bubbles with radius $r_0 \geq 1.2$, the solver is able to verify that they are capture tubes (the cross-out constraint contracts to an empty domain).

Table 5 depicts the results obtained with bubbles having a constant radius $r_0 = 1.1$, $r_0 = 1$ or $r_0 = 0.9$ or a time dependent radius $r_0 = \frac{1}{\sqrt{5}}(1 + t)$. For instance, for $[t_0] = 0$, we can prove that, for $r_0 = 0.9$, the bubble is not a quasi capture tube, but we are not able to conclude for $r_0 = 1$, even for a small ϵ_{min} . It is therefore not necessary for $r_0 = 1$ and for $r_0 = 0.9$ to perform the experiment for $[t_0] = [0, 100]$ since the bubble cannot be proved to be a quasi capture tube. On the other hand, the bubble with a radius $r_0 = 1.1$, and the bubble with an increasing radius $r_0 = \frac{1}{\sqrt{5}}(1 + t)$ are quasi capture tubes for all t_0 in $[0, 100]$.

18:12 Interval CP for Quasi Capture Tube Validation

■ **Table 4** Parameters of the 2D linear system experiment.

First gate	Bubble	Observed parameter
$x, y \in [-100, 100]$	$(x - t)^2 + (y)^2 - r_0^2 \leq 0$	r_0

■ **Table 5** Results for $r_0 = 1.1$, $r_0 = 1$, $r_0 = 0.9$ and $r_0 = \frac{1}{\sqrt{5}}(1 + t)$.

r_0	$[t_0]$	ϵ_{start}	ϵ_{min}	In	Und	Out	CPU
1.1	[0, 100]	{1,1,0.1}	{0.1,0.1,0.01}	2048	0	0	2.6
1	0	{1,1}	{0.1,0.1}	0	2	0	0.08
1	0	{1,1}	{1e-8,1e-8}	0	10	0	0.91
0.9	0	{1,1}	{0.1,0.1}	0	0	2	0.05
$\frac{(1+t)}{\sqrt{5}}$	[0, 100]	{1,1,0.1}	{0.1,0.1,0.01}	7	0	0	0.04

5.3 Linear tracking system

Consider the following linear dynamical system:

$$\dot{\mathbf{x}}(t) = A(\mathbf{x}(t) - \mathbf{T}(t)) \quad (4)$$

with $\mathbf{x}(t)$ the tracking system and $\mathbf{T}(t)$ the target.

We want to study the stability of the system (4) by finding a quasi capture tube. We will study two cases for the system (4), one with a static bubble centered at the origin, and the other one with a dynamic bubble centered at the target.

2D and 3D tracking systems

Consider the 2D linear system:

$$n = 2: \quad A = \begin{bmatrix} 1 & 3 \\ -3 & -2 \end{bmatrix}, \quad T(t) = \begin{bmatrix} \cos(t) \\ \sin(2t) \end{bmatrix} \quad (5)$$

and the 3D linear system:

$$n = 3, \quad A = \begin{bmatrix} 1 & 3 & 0 \\ -3 & -2 & -1 \\ 0 & 1 & -3 \end{bmatrix}, \quad T(t) = \begin{bmatrix} \cos(t) \\ \cos(t) \sin(2t) \\ -\sin(t) \sin(2t) \end{bmatrix} \quad (6)$$

■ **Table 6** Parameters of the linear tracking system experiment.

First gate	Bubble	r_0	Observed parameter
$x_1, x_2 \in [-10, 10]$	$x_1^2 + x_2^2 - r_0^2 \leq 0$	2	Dim/Bubble
$x_1, x_2 \in [-10, 10]$	$(x_1 - T_1(t))^2 + (x_2 - T_2(t))^2 - r_0^2 \leq 0$	2	Dim/Bubble
$x_1, x_2, x_3 \in [-10, 10]$	$x_1^2 + x_2^2 + x_3^2 - r_0^2 \leq 0$	2	Dim/Bubble
$x_1, x_2, x_3 \in [-10, 10]$	$(x_1 - T_1(t))^2 + (x_2 - T_2(t))^2 + (x_3 - T_3(t))^2 - r_0^2 \leq 0$	2	Dim/Bubble

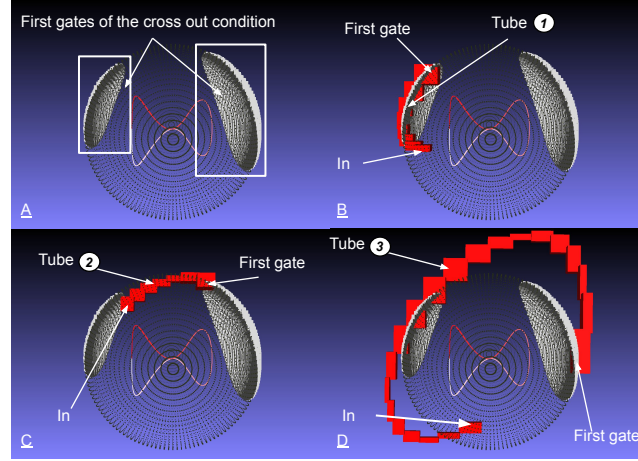
Both targets, in the 2D and 3D linear systems, have a periodic pattern movement and their period is 2π . We can then restrict the study of the stability of both systems to $t_0 \in [0, 2\pi]$ by setting the time domain of the initial gate to $[t_0] = [0, 2\pi]$.

From Table 7 we can conclude that both bubbles are quasi capture tubes for the system (4).

Fig. 4 illustrates the 3D tracking system.

■ **Table 7** Results for both systems (2D and 3D) and both bubbles (static and dynamic).

Dim	Bubble	ϵ_{start}	ϵ_{min}	In	Und	Out	CPU
2D	Static	{1,1,0.05}	{0.1,0.1,0.01}	370	0	0	1.20
2D	Dynamic	{1,1,0.05}	{0.1,0.1,0.01}	1021	0	0	1.65
3D	Static	{1,1,1,0.05}	{0.1,0.1,0.1,0.01}	3290	0	0	7.10
3D	Dynamic	{1,1,1,0.05}	{0.1,0.1,0.1,0.01}	4040	0	0	11.94



■ **Figure 4** Sample of tubes of the 3D linear tracking system leaving the static bubble. The figure illustrates the bubble and the tubes in the state dimensions. **A**: First gates satisfying the cross out constraints appear in white on the spherical bubble of radius $r_0 = 2$. The periodic target, with an “ ∞ ” trajectory, appears in the center of the bubble. Its color is going from red at $t = 0$ to white at $t = 2\pi$. **B** and **C**: Tubes (in red) getting almost immediately inside the sphere. **D**: Tube going far away from the sphere and finally landing after one first unsuccessful landing trial.

5.4 Pursuit evasion game

A “pursuit evasion” game is a situation where a pursuer (P) wants to catch an evader (E) trying to escape from him. In the following experiment, we will present two problems based the “pursuit evasion” game, one in the plane, and the other one in the 3D-space. The evader (E) will be at the center of a dynamic bubble, and we want the pursuer to stay inside the bubble in order to catch the evader. In other words, we want the bubble to be a capture tube, or at least, a quasi capture tube.

Pursuit game on the plane

Consider the following pursuer P and evader E :

$$P : \begin{cases} \dot{x} = u_1 \cos(\theta) \\ \dot{y} = u_1 \sin(\theta) \\ \dot{\theta} = u_2 \end{cases} \quad E : \begin{cases} x_d = v \cdot t \\ y_d = \sin(\rho t) \end{cases} \quad (7)$$

where x and y are the position and θ the heading of P .

The velocity of the pursuer and its heading are respectively controlled by $u_1 = \|n\|$ and $u_2 = -K \sin(\theta - \theta_d)$ such that $\theta_d = \text{atan2}(n)$ and n is defined as follows:

$$n = \begin{bmatrix} n_x \\ n_y \end{bmatrix} = \frac{1}{dt} \begin{bmatrix} x_d - x \\ y_d - y \end{bmatrix} + \begin{bmatrix} \dot{x}_d \\ \dot{y}_d \end{bmatrix}$$

18:14 Interval CP for Quasi Capture Tube Validation

■ **Table 8** Parameters of the pursuit game on the plane experiment.

First gate	Bubble	r_0	Observed parameter
$x, y \in [-10, 10], \theta \in [0, 2\pi]$	$(x - x_d)^2 + (y - y_d)^2 - r_0^2 = 0$	1	ϵ_h

We add the following constraint on the heading of the pursuer:

$$h(x, y, \theta, t) = (\cos(\theta) - \cos(\theta_d))^2 + (\sin(\theta) - \sin(\theta_d))^2 - \epsilon_h \leq 0$$

Constants: $K = 1, v = 7, \rho = 1, dt = 1$

Pursuit Evasion game in the 3D-space

Let the pursuer P and the evader E :

$$P : \begin{cases} \dot{x} = u_1 \cdot \cos(\theta) \cdot \cos(\psi) \\ \dot{y} = u_1 \cdot \cos(\theta) \cdot \sin(\psi) \\ \dot{z} = u_1 \cdot \sin(\theta) \\ \dot{\psi} = u_2 \\ \dot{\theta} = u_3 \end{cases} \quad E : \begin{cases} x_d = v \cdot w \cdot t \\ y_d = v \cdot w \cdot \sin(w \cdot t) \\ z_d = -v \cdot w \cdot \cos(w \cdot t) \end{cases} \quad (8)$$

where x, y and z are the position, ψ is the circular rotation speed and θ is the vertical rotation speed of P. The controls $u_1 = \|n\|$, $u_2 = K(\psi - \psi_d)$ and $u_3 = K(\theta - \theta_d)$. Without going into details, θ_d and ψ_d are defined with analytical expressions.

$$n = \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} = \frac{1}{dt} \begin{bmatrix} x_d - x \\ y_d - y \\ z_d - z \end{bmatrix} + \begin{bmatrix} \dot{x}_d \\ \dot{y}_d \\ \dot{z}_d \end{bmatrix}$$

We have added the following constraints on the circular and vertical rotations of the pursuer:

$$h_1(\psi, t) = (\cos(\psi) - \cos(\psi_d))^2 + (\sin(\psi) - \sin(\psi_d))^2 - \epsilon_h \leq 0$$

$$h_2(\theta, t) = (\cos(\theta) - \cos(\theta_d))^2 + (\sin(\theta) - \sin(\theta_d))^2 - \epsilon_h \leq 0$$

Constants: $v = 2, w = 1, K = 10, dt = 1$.

■ **Table 9** Parameters of the pursuit game in the 3D-space experiment.

First gate	Tube candidate	r_0	Observed parameter
$x, y, z \in [-10, 10], \theta, \psi \in [0, 2\pi]$	$(x - x_d)^2 + (y - y_d)^2 + (z - z_d)^2 - r_0^2 = 0$	1	ϵ_h

Pursuit evasion game results

Here again, both evaders follow a periodic pattern of period 2π , so the study is restricted to a time domain $t_0 \in [0, 2\pi]$.

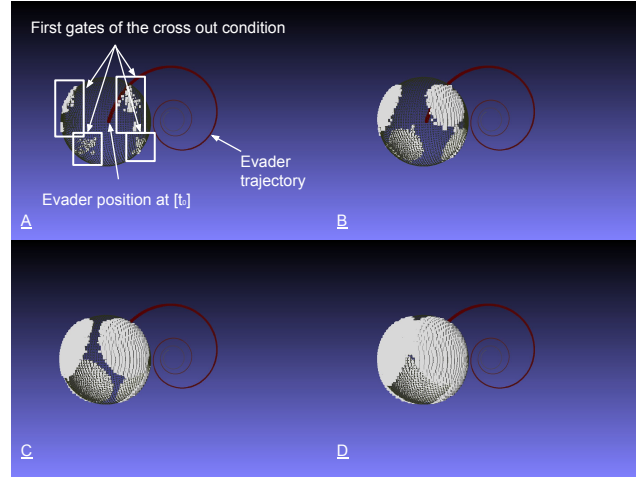
When the problem scales up (in the number of the state variables, number of nonlinearities, system stiffness, etc.), the solver faces some difficulties. We can see in Tables 10 and 11 how varies the number of tubes computed by the solver for validating a quasi capture tube (the reader can compare with the previous experiments). The number of tubes required can be drastically lowered by using small values for parameter ϵ_h that restrict the initial heading (resp. circular and vertical rotations) of the pursuer (see Fig. 5).

■ **Table 10** Results of the pursuit game on the plane show that, with a small parameter ϵ_h , we can validate the quasi capture tube on the whole period of the evader.

$[t_0]$	ϵ_h	ϵ_{start}	ϵ_{min}	In	Und	Out	CPU
0	0.02	{0.1, 0.1, 0.1}	{0.01, 0.01, 0.005}	129	0	0	1.74
$[0, 2\pi]$	0.02	{0.1, 0.1, 0.1, 0.05}	{0.01, 0.01, 0.005, 0.005}	16672	0	0	585
0	0.2	{0.1, 0.1, 0.1}	{0.01, 0.01, 0.005}	437	0	0	8.01
$[0, 2\pi]$	0.2	{0.1, 0.1, 0.1, 0.05}	{0.01, 0.01, 0.005, 0.005}	105735	0	0	6561

■ **Table 11** Results of the pursuit game in 3D-space: even for small parameter value ϵ_h , studying one tenth of the period for $[t_0]$ requires a huge CPU time. On the other hand, the quasi capture tube is then validated.

$[t_0]$	ϵ_h	ϵ_{start}	ϵ_{min}	In	Und	Out	CPU
0	0.1	{0.1, 0.1, 0.1, 0.05, 0.05}	{0.01, 0.01, 0.01, 0.005, 0.005}	21414	0	0	590
0	0.08	{0.1, 0.1, 0.1, 0.05, 0.05}	{0.01, 0.01, 0.01, 0.005, 0.005}	8128	0	0	236
0	0.0625	{0.1, 0.1, 0.1, 0.05, 0.05}	{0.01, 0.01, 0.01, 0.005, 0.005}	1852	0	0	62.4
0	0.05	{0.1, 0.1, 0.1, 0.05, 0.05}	{0.01, 0.01, 0.01, 0.005, 0.005}	176	0	0	11.1
$[0, \frac{\pi}{5}]$	0.05	{0.1, 0.1, 0.1, 0.05, 0.05, 0.05}	{0.01, 0.01, 0.01, 0.005, 0.005, 0.005}	241103	0	0	109



■ **Figure 5** Pursuit evasion game in 3D-space: Illustration of the bubble and the evader trajectory (in red) in the state dimensions. First gates satisfying the cross out constraints at $[t_0] = 0$ appear in white on the spherical bubble of radius $r_0 = 1$, centered on the position of the evader at $[t_0] = 0$. We can notice how the number of gates varies for different values for ϵ_h . **A**: $\epsilon_h = 0.05$. **B**: $\epsilon_h = 0.0625$. **C**: $\epsilon_h = 0.08$. **D**: $\epsilon_h = 0.1$

6 Conclusion

We have proposed a Branch and Contract solver dedicated to the quasi capture tube validation, a problem for which the algorithms are almost absent. The solver is sufficiently generic to handle different problems. The performance of the solver is based on filtering/contraction algorithms and on the use of the guaranteed integration solver CAPD for the integration of differential equations. We have validated the solver in different application examples scaling from 2 to 5 state dimensions. To simplify the problem, the solver can accept additional constraints on the command parameters. We have tried to propagate domain reductions backward (from the escape constraint deductions to t_0) with no success. Nevertheless, there is still improvement space for future work. We could improve the shape of the capture

tube candidate using Lyapunov approaches or parametric barrier functions [9]. We could also improve our algorithm for computing in an auto-adaptive manner the ϵ_{start} parameter deciding the tube size under which it is relevant to run the guaranteed integration solver. Finally, we could improve our software using a multi threading approach.

References

- 1 A. Akkouché, J.-B. Bénéfice, Q. Bréfort, B. Desrochers F. Carbonera, T. Issautier, M. Laranjeira-Moreira, V. Le Doze, D. Monnet, and A. Oubelhaj. BUBBIBEX with IBEX. Engineering internship report, ENSTA Bretagne, 2014. URL: https://www.ensta-bretagne.fr/jaulin/archirob.html#bm_2013_14.
- 2 J.-P. Aubin. Viability Kernels and Capture Basins of Sets Under Differential Inclusions. *SIAM Journal on Control and Optimization*, 40(3):853–881, 2001. doi:10.1137/S036301290036968X.
- 3 F. Benhamou, F. Goualard, L. Granvilliers, and J.F. Puget. Revising Hull and Box Consistency. In D. De Schreye, editor, *Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999*, pages 230–244. MIT Press, 1999.
- 4 F. Blanchini and S. Miani. *Set Theoretic Methods in Control*. Birkhauser, 2008.
- 5 J.C. Butcher. *Numerical Methods for Ordinary Differential Equations*. Wiley, 2004. URL: <https://books.google.fr/books?id=okzplwEX8aEC>.
- 6 G. Chabert. IBEX – an Interval-Based EXplorer, 2020. URL: <http://www.ibex-lib.org/>.
- 7 H. Collavizza, F. Delobel, and M. Rueher. Comparing Partial Consistencies. *Reliable Computing*, 5(3):213–228, 1999.
- 8 Julien Alexandre dit Sandretto and Alexandre Chapoutot. Validated Explicit and Implicit Runge–Kutta Methods. *Reliable Computing*, 22(1):79–103, July 2016.
- 9 A. Djabbalah, A. Chapoutot, M. Kieffer, and O. Bouissou. Construction of Parametric Barrier Functions for Dynamical Systems using Interval Analysis. *Automatica*, 78:287–296, 2017. doi:10.1016/j.automatica.2016.12.013.
- 10 F. Domes. GLOPTLAB: a Configurable Framework for the Rigorous Global Solution of Quadratic Constraint Satisfaction Problems. *Optimization Methods & Software*, 24:727–747, October 2009. doi:10.1080/10556780902917701.
- 11 A. Girard, C. Le Guernic, and O. Maler. Efficient Computation of Reachable Sets of Linear Time-invariant Systems with Inputs. *Hybrid Systems: Computation and Control*, 3927:257–271, 2006.
- 12 E. R. Hansen. *Global Optimization using Interval Analysis*. Marcel Dekker, New York, NY, 1992.
- 13 L. Jaulin, D. Lopez, V. Le Doze, S. Le Menec, J. Ninin, G. Chabert, M. S. Ibenseddik, and A. Stancu. Computing Capture Tubes. In Marco Nehmeier, Jürgen Wolff von Gudenberg, and Warwick Tucker, editors, *Scientific Computing, Computer Arithmetic, and Validated Numerics*, pages 209–224. Cham, 2016. Springer International Publishing.
- 14 T. Kapela, M. Mrozek, D. Wilczak, and P. Zgliczynski. CAPD: : Dynsys: a flexible C++ toolbox for rigorous numerical analysis of dynamical systems. *CoRR*, abs/2010.07097, 2020. arXiv:2010.07097.
- 15 F. Le Bars, J. Sliwka, L. Jaulin, and O. Reynet. Set-membership State Estimation with Fleeting Data. *Automatica*, 48(2):381–387, 2012. doi:10.1016/j.automatica.2011.11.004.
- 16 S. Le Menec. Linear Differential Game with Two Pursuers and One Evader. *Advances in Dynamic Games*, 11:209–226, 2011.
- 17 Y. I. Lee and B. Kouvaritakis. Constrained Robust Model Predictive Control Based on Periodic Invariance. *Automatica*, 42:2175–2181, 2006.
- 18 O. Lhomme. Consistency Techniques for Numeric CSPs. In R. Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*, pages 232–238. Morgan Kaufmann, 1993. URL: <http://ijcai.org/Proceedings/93-1/Papers/033.pdf>.

- 19 M. Lhommeau, L. Jaulin, and L. Hardouin. Inner and Outer Approximation of Capture Basins using Interval Analysis. *ICINCO 2007*, 2007.
- 20 R. Lohner. Enclosing the Solutions of Ordinary Initial and Boundary Value Problems. In E. Kaucher, U. Kulisch, and Ch. Ullrich, editors, *Computer Arithmetic: Scientific Computation and Programming Languages*, pages 255–286. BG Teubner, Stuttgart, Germany, 1987.
- 21 F. Messine. *Méthodes d’Optimisation Globale basées sur l’Analyse d’Intervalle pour la Résolution des Problèmes avec Contraintes*. PhD thesis, LIMA-IRIT-ENSEEIH-ENST, Toulouse, 1997.
- 22 R. E. Moore. *Interval Analysis*, volume 4. Prentice-Hall Englewood Cliffs, 1966.
- 23 N. S. Nedialkov, K. R. Jackson, and J. D. Pryce. An Effective High-Order Interval Method for Validating Existence and Uniqueness of the Solution of an IVP for an ODE. *Reliable Computing*, 7(6):449–465, 2001. doi:10.1023/A:1014798618404.
- 24 S. Oлару, J.A. De Dona, M.M. Seron, and F. Stoican. Positive Invariant Sets for Fault Tolerant Multisensor Control Schemes. *International Journal of Control*, 83(12):2622–2640, 2010.
- 25 S. V. Rakovic, E. C. Kerrigan, K. I. Kouramas, and D. Q. Mayne. Invariant approximations of the minimal robust positively invariant set. *IEEE Trans. Autom. Control*, 50(3):406–410, 2005.
- 26 S. Ratschan and Z. She. Providing a Basin of Attraction to a Target Region of Polynomial Systems by Computation of Lyapunov-like Functions. *SIAM J. Control and Optimization*, 48(7):4377–4394, 2010.
- 27 N. Revol, K. Makino, and M. Berz. Taylor Models and Floating-point Arithmetic: proof that arithmetic operations are validated in COSY. *Journal of Logic and Algebraic Programming*, 64:135–154, 2005.
- 28 S. Rohou, A. Bedouhene, G. Chabert, A. Goldsztejn, L. Jaulin, B. Neveu, V. Reyes, and G. Trombettoni. Towards a Generic Interval Solver for Differential-Algebraic CSP. In *Proc. CP, Constraint Programming, Springer, LNCS 12333*, pages 864–879. Springer, 2020.
- 29 S. Rohou et al. The Tubex Library – Constraint-programming for robotics, 2021. URL: <http://simon-rohou.fr/research/tubex-lib/>.
- 30 Simon Rohou, Luc Jaulin, Lyudmila Mihaylova, Fabrice Le Bars, and Sandor M. Veres. Guaranteed Computation of Robot Trajectories. *Robotics and Autonomous Systems*, 93:76–84, 2017. doi:10.1016/j.robot.2017.03.020.
- 31 S. Romig, L. Jaulin, and A. Rauh. Using Interval Analysis to Compute the Invariant Set of a Nonlinear Closed-Loop Control System. *Algorithms*, 12(262), 2019.
- 32 P. Saint-Pierre. Hybrid Kernels and Capture Basins for Impulse Constrained Systems. In C.J. Tomlin and M.R. Greenstreet, editors, *Hybrid Systems: Computation and Control*, volume 2289, pages 378–392. Springer-Verlag, 2002.
- 33 F. Tahir and M. Jaimoukha. Low-Complexity Polytopic Invariant Sets for Linear Systems Subject to Norm-Bounded Uncertainty. *IEEE Trans. Autom. Control*, 60:1416–1421, 2015.
- 34 G. Trombettoni and G. Chabert. Constructive Interval Disjunction. In *Proc. CP, Constraint Programming, LNCS 4741*, pages 635–650. Springer, 2007.
- 35 J. Wan, J. Vehi, and N. Luo. A Numerical Approach to Design Control Invariant Sets for Constrained Nonlinear Discrete-time Systems with Guaranteed Optimality. *Journal of Global Optimization*, 44:395–407, 2009.
- 36 J. A. Yorke. Invariance for Ordinary Differential Equations. *Mathematical System Theory*, 1(4):353–372, 1967.

Exhaustive Generation of Benzenoid Structures

Sharing Common Patterns

Yannick Carissan ✉ 

Aix Marseille Univ, CNRS, Centrale Marseille, ISM2, Marseille, France

Denis Hagebaum-Reignier ✉ 

Aix Marseille Univ, CNRS, Centrale Marseille, ISM2, Marseille, France

Nicolas Prcovic ✉

Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

Cyril Terrioux ✉ 

Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

Adrien Varet ✉

Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

Abstract

Benzenoids are a subfamily of hydrocarbons (molecules that are only made of hydrogen and carbon atoms) whose carbon atoms form hexagons. These molecules are widely studied both experimentally and theoretically and can have various physicochemical properties (mechanical resistance, electronic conductivity, ...) from which a lot of concrete applications are derived. These properties can rely on the existence or absence of fragments of the molecule corresponding to a given pattern (some patterns impose the nature of certain bonds, which has an impact on the whole electronic structure). The exhaustive generation of families of benzenoids sharing the absence or presence of given patterns is an important problem in chemistry, particularly in theoretical chemistry, where various methods can be used to better understand the link between their shapes and their electronic properties.

In this paper, we show how constraint programming can help chemists to answer different questions around this problem. To do so, we propose different models including one based on a variant of the subgraph isomorphism problem and we generate the desired structures using Choco solver.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence

Keywords and phrases Constraint satisfaction problem, modeling, pattern, application, theoretical chemistry

Digital Object Identifier 10.4230/LIPIcs.CP.2021.19

Supplementary Material *Software (Source Code)*: <https://github.com/AdrienVaret/BenzenoidApplicationReleases/releases/tag/latest-version>

archived at `swb:1:dir:8b64aa73cb6b96ad557006c28255fc8642cef0a7`

Funding This work has been funded by the Agence Nationale de la Recherche project ANR-16-CE40-0028.

1 Introduction

Polycyclic aromatic hydrocarbons (PAHs) are hydrocarbons whose carbons are forming cycles of different sizes. *Benzenoids* are a subfamily of PAHs for which all the cycles are of size 6. *Benzene*, represented in Figure 1(a), is the smallest one. It is made of 6 carbon atoms and 6 hydrogen atoms. Its carbon atoms form a hexagon (also called *benzenic cycle* or *benzenic ring*) and each of them is linked to a hydrogen atom. Benzenoids can also be seen as the molecules obtained by aggregating benzenic rings. For example, Figure 1(b) shows anthracene, which contains three benzenic rings. Atoms establish bonds between themselves



© Yannick Carissan, Denis Hagebaum-Reignier, Nicolas Prcovic, Cyril Terrioux, and Adrien Varet; licensed under Creative Commons License CC-BY 4.0

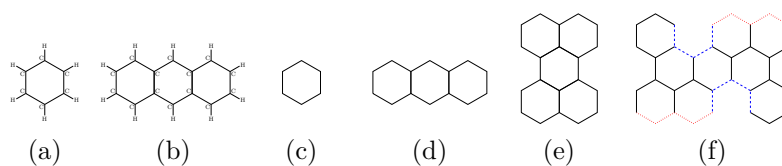
27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 19; pp. 19:1–19:18

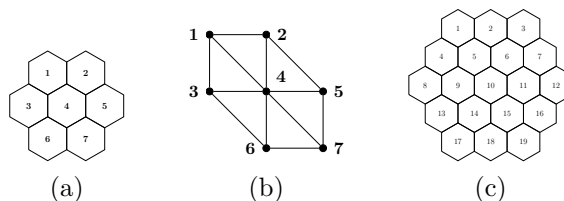
Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Examples of benzenoids: benzene (a) and anthracene (b) with their graphical representation (c) and (d), perylene (e) and a benzenoid containing two instances of the pattern *deep bay* (in blue dashed) and two of instances of the pattern *zigzag bay* (in dotted red) (f).



■ **Figure 2** Coronene (a), its hexagon graph (b) and the coronenoid of size 3 (c).

which can be single or double depending on the number of electrons involved in the bond. In a benzenoid, each carbon atom is linked either to two carbon atoms and one hydrogen atom, or to three carbon atoms. In the following, hydrogen atoms play no role (and their presence can be inferred if necessary). Also, they can be omitted in the representation. Thus, a benzenoid can be represented as an undirected graph $B = (V, E)$ in which each vertex of V corresponds to a carbon atom and each edge of E reflects the existence of a bond between the two corresponding carbon atoms. Note that the nature of bonds (simple or double) has no importance for our purpose. This graph is connected, bipartite and planar. Figures 1(c) and (d) show the graphs corresponding to benzene and anthracene. Moreover, since benzenoids can be defined as a combination of fused benzenic rings, we consider, for each benzenoid B , a second graph called the *hexagon graph*. This graph $B_h = (V_h, E_h)$ is an undirected graph in which each vertex corresponds to a hexagon (i.e. a benzenic ring) of B and such that two vertices are connected by an edge if the corresponding hexagons share an edge in the graph B . Figures 2(a)-(b) show the graph corresponding to coronene and its hexagon graph.

Benzenoids and more generally PAHs are well-studied in various fields (molecular nanoelectronics, organic synthesis, interstellar chemistry, ...) because of their energetic stability, molecular structures or optical spectra. They have a wide variety of physicochemical properties depending on their size and structure. For example, they can combine a strong mechanical resistance with high electronic conductivity. These properties can rely on the existence or absence of fragments of the molecule corresponding to a given pattern. Some patterns impose the nature of certain bonds, which impacts the whole electronic structure. For instance, perylene (Figure 1(e)) can be seen as two overlapping triangles of three fused rings with the consequence that the central bonds are essentially simple.

The controlled synthesis of PAH with tuned edges is very recent. This is a hot topic as shown by the number of recent publications on the synthesis of such compounds in high impact factor chemistry journals (e.g. [29, 19, 1, 6, 8, 14, 16, 31, 35]). This has motivated many theoretical studies to better understand the impact of the edge topology on their electronic properties. This chemistry leads to enhanced optoelectronic molecular properties [11, 20, 24, 25, 28, 36] or magnetic properties [23, 38], which are further improved and combined in mixed edge molecules [17, 26, 27, 37]. As recent examples, it was shown that the addition of two extra K-regions (or armchair edges, see Figure 5(a)) to hexabenzocoronene leads to an enhanced optical activity of the molecules with potential applications as organic

laser materials [11]. Furthermore, PAHs with armchair edges are semiconductors with high bands gaps whereas zigzag edges (see Figure 5(g)) lead to improved conductivity but are fairly unstable. Thus chemists intend to design molecules with zigzag patterns at the edges and stabilize the molecular structure with neighboring cove regions, which also lead to higher dispersibility in solution and improved optoelectronic properties [26]. Niu et al. [26] provide several benzenoids that chemists can synthesize and which contain such patterns. For instance, Figure 1(f) describes one of them which contains two instances of the pattern *cove* (depicted in blue dashed and called *deep bay* in [34]) and two instances of the pattern *zigzag* (depicted in dotted red and called *zigzag bay* in [34]). It is thus important to be able to exhaustively generate families of benzenoids sharing common given patterns on their edges for a given number of fused rings.

In the literature, bespoke approaches have been proposed to generate benzenoid structures satisfying or not some particular properties (e.g. [3]). They turn to be very efficient in practice but are difficult to adapt to the needs of chemists. Moreover, they only consider properties on the whole molecule. In [5], we have described a new approach based on constraint programming (CP) which is more flexible while being competitive. More precisely, we model the problem as an instance of the *Constraint Satisfaction Problem (CSP)*. Remember that a CSP instance can be defined as a triplet (X, D, C) where $X = \{x_1, \dots, x_n\}$ is the set of *variables*, $D = \{D_{x_1}, \dots, D_{x_n}\}$ is the set of domains, the domain D_{x_i} being related to the variable x_i , and $C = \{c_1, \dots, c_e\}$ represents the set of the *constraints* which define the interactions between the variables and describe the allowed combinations of values. For sake of simplicity and expressivity, our model exploits *graph variables* (notably to represent the hexagon graph). Graph variables have as domain a set of graphs defined by a lower bound (a subgraph called *GLB*) and an upper bound (a super-graph called *GUB*). In [5], we used Choco solver [13] since this library supports *graph variables* and its graph-related constraints (e.g. the *connected* constraint [12]) and also the usual global constraints which make the modeling easier. However, this model only handles global properties. Hence, in this paper, we describe how to integrate the notion of pattern. Several models being possible, we study them on the basis of the different questions that arise about patterns before comparing them experimentally.

This paper is organized as follows. In Section 2, we recall how to generate benzenoid structures, in particular with the help of CP. Afterward, in Section 3, we formalize the problem we are interested in and address different questions that may be of interest for chemists in Sections 4 to 6. Finally, we assess experimentally some models in Section 7, before concluding in Section 8.

2 Generating Benzenoid Structures

Generating benzenoid structures which have certain structural properties (e.g. having a given number of hexagons or having a particular structure from a graph viewpoint) is an interesting and important problem in theoretical chemistry [9, 21, 22, 30]. This problem is a preliminary step to the study of their chemical properties. It can be formally defined as follows: Given a set of structural properties \mathcal{P} , generate all the benzenoid structures satisfying the properties of \mathcal{P} . These properties may concern the number of carbon or hexagon atoms or particular shapes (tree, rectangle, presence of “holes”, ...).

In the literature, bespoke methods have been proposed (e.g. [3]). If they are often efficient in practice, they have the disadvantage of being difficult to adapt to the needs of chemists. In [5], we proposed a CP model of this problem and showed how easy it is to meet the wishes expressed by the chemists by simply adding variables and constraints. Moreover, beyond its flexibility, this approach is relatively efficient thanks to Choco solver.

We now recall the CP model allowing to generate all structures with n hexagons. It relies on the property that any benzenoid of n hexagons can be embedded in a coronenoid of size at most $k(n) = \lfloor \frac{n}{2} + 1 \rfloor$. A coronenoid of size k is a benzene molecule to which $k - 1$ crowns of hexagons have been successively added. Coronene (see Figure 2(a)) is the coronenoid of size 2. Figure 2(c) shows the coronenoid of size 3. We can remark that the number of hexagons in the i th crown grows with i . Thereafter, we denote $B_h^{c(k(n))}$ (where $c(k(n))$ stands for coronenoid of size $k(n)$) the hexagon graph of the coronenoid of size $k(n)$, n_c its number of hexagons and m_c its number of edges. The hexagons and edges of $B_h^{c(k(n))}$ are arbitrarily numbered starting from 1. Figure 2 presents a possible numbering. First, in this model (denoted \mathcal{M}), we consider a graph variable x_G to represent the hexagon graph of the desired structure. Its domain is the set of all subgraphs between the empty graph and $B_h^{c(k(n))}$. The use of a graph variable makes it much easier to express the connectedness of the generated structures (as described below). We also exploit a set of n_c Boolean variables $\{x_1, \dots, x_{n_c}\}$. The variable x_i is set to 1 if the i -th hexagon of $B_h^{c(k(n))}$ is used in x_G , 0 otherwise. Similarly, we also consider a set of m_c Boolean variables $y_{i,j}$. The variable $y_{i,j}$ is set to 1 if the edge $\{i, j\}$ of $B_h^{c(k(n))}$ is used in x_G , 0 otherwise.

Then, the following properties are expressed thanks to constraints:

- *Link between x_G and x_i (resp. $y_{i,j}$):* a **channeling** constraint imposes that $x_i = 1 \iff x_G \text{ contains the vertex } i$ (resp. $y_{i,j} = 1 \iff x_G \text{ contains the edge } \{i, j\}$).
- *x_G is an induced subgraph of $B_h^{c(k(n))}$:* Any value of x_G is not necessarily a valid hexagon graph. To guarantee its validity, it must correspond to a subgraph of $B_h^{c(k(n))}$ induced by the vertices belonging to x_G . Thus, for each edge $\{i, j\}$ of $B_h^{c(k(n))}$, one adds a constraint $x_i = 1 \wedge x_j = 1 \Rightarrow y_{i,j} = 1$. In other words, the edge $\{i, j\}$ exists in x_G if and only if the vertices i and j appear in x_G .
- *The structure has n hexagons:* $\sum_{i \in \{1, \dots, n_c\}} x_i = n$.
- *The hexagon graph is connected:* It is achieved by applying the **connected** graph constraint on x_G [12].
- *Six hexagons forming a cycle generate a hexagon (and not a hole):* For each hexagon u , let $N(u)$ denotes the set of the neighbors of u in the hexagon graph. Then, for each vertex u having 6 neighbors, the property is ensured by adding a constraint between x_u and the variables corresponding to its neighbors which imposes: $\sum_{v \in N(u)} x_v = 6 \Rightarrow x_u = 1$.

Finally, several constraints are added in order to avoid redundancies. First, x_G must have at least one vertex on the top (resp. left) edge of $B_h^{c(k(n))}$ in order to discard the symmetries by translation. This can be achieved by posting a constraint that specifies that the sum of the Boolean variables x_i associated with the top (resp. left) edge of $B_h^{c(k(n))}$ is strictly positive. Then, one must ensure that the graph described by x_G is the only representative of its symmetry class. There are up to twelve symmetric solutions: six 60 degree rotational symmetries combined with a possible axial symmetry. These symmetries are broken by the constraint **lex-lead** [10]. For each of the twelve symmetries, one needs to add n_c Boolean variables (one per variable x_i) and a total of $3 \cdot n_c$ ternary clauses.

This model can easily be implemented with Choco solver. It can also be specialized to take into account the needs of chemists by adding variables and/or constraints. For example, generating structures with a tree shape (called *catacondensed* benzenoids) simply requires the addition of the **tree** graph constraint on x_G to the general model. Other properties have been modeled in order to generate structures having a rectangular shape, possessing a hole or being symmetrical [5].

3 Considering Patterns

The model \mathcal{M} , presented in [5] and recalled in Section 2, allows to express the benzenoid structure generation problem in all its generality. If several specializations of this model have been proposed in [5], all of them correspond to structural properties concerning the whole molecule. These properties could thus be qualified as *global*. However, in some cases, it may be useful to reason in terms of local properties that may or may not be satisfied by some parts (called fragments) of the generated structures. In particular, among these local properties, it is important to be able to deal with the local properties related to the edge of the benzenoid structure.

The local properties we consider in this article can be defined by “drawing” a shape whose basic bricks are hexagons. These hexagons can be of three different natures:

- (i) The *positive* hexagons whose presence is required in the property,
- (ii) The *negative* hexagons whose absence is required in the property,
- (iii) The *neutral* hexagons whose presence or absence has no influence on the property.

If the use of positive hexagons is obvious, one can ask the question of the interest of negative or neutral hexagons. Negative (respectively neutral) hexagons are useful, for example, to indicate that there is nothing between two positive hexagons or to model the edge of the benzenoid (resp. to guarantee a certain gap between two positive hexagons). In order to represent the desired shapes, we now introduce the notion of extended hexagon graph:

► **Definition 1** (extended hexagon graph). *An extended hexagon graph is a hexagon graph whose vertices and edges are labeled by the symbols $+$ (for positive), $-$ (for negative) and \circ (for neutral) such that:*

- (i) *Each vertex is labeled with the nature of the hexagon it represents.*
- (ii) *An edge is labeled $-$ if at least one of its vertices is labeled $-$. Otherwise, it is labeled \circ if at least one of its vertices is labeled \circ . Otherwise, it is labeled $+$.*

As for the hexagons (or vertices), the labels associated with the edges qualify the status that the interaction between two hexagons must have in the local property that we wish to define. Formally a local property can be defined by a *pattern*:

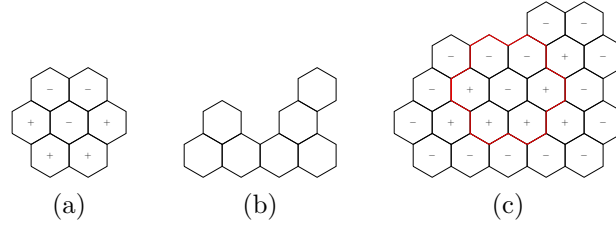
► **Definition 2** (pattern). *A pattern P is defined by giving a triplet (P_+, P_-, P_\circ) and an extended hexagon graph P_h such that:*

- (i) P_+ , P_- and P_\circ denote the set of positive, negative and neutral hexagons respectively,
- (ii) these three sets are pairwise disjoint and
- (iii) P_h is a connected graph on the set of hexagons defined by $P_+ \cup P_- \cup P_\circ$.

Its order k_P is the maximal length (expressed in terms of the number of edges) of the shortest paths of P_h separating a negative or neutral hexagon from a positive hexagon.

In other words, a pattern is defined by a collection of positive, negative and neutral hexagons whose arrangement is described by an extended hexagon graph. As an example, Figure 3(a) shows the pattern *deep bay* [34] of order 1 composed of four positive hexagons and three negative ones. The bonds (i.e. the edges of hexagons) which are at the interface between the positive hexagons and the negative ones allow us to handle the local property of the edge of the benzenoid depicted in blue in Figure 1(f). We finally define the notion of pattern inclusion:

► **Definition 3.** *Given a non-negative integer k , let B_h^k be the extended hexagon graph representing the benzenoid B surrounded by k layers of negative hexagons (i.e., the extended graph of B augmented by all negative hexagons located within distance k of a hexagon of B).*



■ **Figure 3** The pattern *deep bay* (a), a benzenoid satisfying this pattern (b) and the “extended” benzenoid related to its extended hexagon graph B_h^1 with the pattern in red (c).

A fragment F^k of order k of a benzenoid B is a subset of hexagons of B_h^k whose extended hexagon graph is connected. It satisfies the pattern P if $k = k_P$ and if there exists a bijection that maps a positive or neutral hexagon of P to each positive hexagon of F^k and maps a negative or neutral hexagon of P to each negative hexagon of F^k . A benzenoid B contains (or includes) the pattern P if it has a fragment of order k_P satisfying P .

Considering B_h or B_h^k does not change the nature of the benzenoid B . B_h^1 simply materializes the vacuum around it, which is necessary for some properties. For example, the benzenoid in Figure 3(b) (like the one in Figure 1(f)) satisfies the pattern *deep bay* of Figure 3(a). For this, we must take into account the absence of hexagon at the edge of the benzenoid to identify a suitable fragment, which is achieved thanks to its extended hexagon graph B_h^k (see Figure 3(c)).

In this paper, we aim to generate benzenoid structures satisfying local properties expressed thanks to the patterns introduced above. These local properties can take different forms. The simplest one is to include a given pattern. Then, one can also be interested in generalizing the approach by including several different patterns or a given number of times the same pattern. On the contrary, one may also wish to exclude a given pattern. The following sections deal with these different issues. In all cases, the idea is to generate benzenoid structures starting from the general model \mathcal{M} . By so doing, it follows that it is quite possible to consider both global and local properties.

4 Generating Structures Including a Pattern

Let P be a pattern involving n_P hexagons which can be positive, negative or neutral. We arbitrarily number each hexagon of the pattern P from 1 to n_P . The sets P_+ , P_- and P_0 are then defined accordingly. In this section, we wish to model the problem of generating all benzenoid structures having n hexagons and including the pattern P . We first consider all the variables and constraints of the general model \mathcal{M} to which we will add variables and constraints to express the fact that the pattern must be present in the generated structures. At this level, we have several possibilities depending on the point of view we consider. In the following, we explore three tracks. The first one consists in identifying all the possible locations of a fragment satisfying the pattern P . The second one considers the existence of a fragment by reasoning on the neighborhood of each hexagon. Finally, the third one exploits the proximity of our problem with the subgraph isomorphism problem.

4.1 First Model

We start with the model \mathcal{M} and thus with a coronenoid of size $k(n)$. In this first model (denoted \mathcal{M}_{i_1}), we first identify all the possible fragments of the pattern P in this coronenoid. Their number being in $O(|c(k(n))|) = O(n^2)$, this computation can be efficiently performed

using rotations, axial symmetries and translations. For each of these fragments F_i , we define the sets F_{i+} , $F_{i\circ}$ and F_{i-} of its positive, neutral and negative hexagons. We associate to each fragment F_i a Boolean variable e_i such that the fragment F_i is present in the structure under construction if e_i is true. This is guaranteed via the constraint $e_i = 1 \Rightarrow \bigwedge_{j \in F_{i-}} x_j = 0 \wedge \bigwedge_{j \in F_{i+}} x_j = 1$. Note that for patterns whose order is strictly positive, it is not necessary to consider a larger coronenoid. Indeed, the fragment can be placed at the edge of the coronenoid with negative or neutral hexagons being outside this coronenoid and, therefore, being considered as absent. In this case, these hexagons will not be represented in F_{i-} , nor in $F_{i\circ}$, but placed in a set F_{i*} . Finally, we set the sum constraint $\sum_j e_j = 1$ to guarantee the existence of at least one fragment satisfying the pattern P .

4.2 Second Model

In this second model (denoted \mathcal{M}_{i_2}), we express the existence of a fragment corresponding to the pattern P by reasoning on the neighborhood of each hexagon. To do so, starting from the model \mathcal{M} , we add a variable f_i per hexagon of the coronenoid of size $k(n)$. Each variable f_i has domain $\{0, 1, \dots, n_P\}$. The variable f_i takes a positive value j if the hexagon i of the coronenoid of size $k(n)$ participates in the searched fragment as a hexagon occupying the position j in P , 0 otherwise. Thus, the variable f_i specifies whether the hexagon i is involved in the fragment and if so to which hexagon of the pattern P it corresponds. Then, since the generation of the benzenoid structures and the search for a fragment are done simultaneously, we need to ensure their concordance. In particular, we must guarantee that the positive (resp. negative) hexagons are indeed present (resp. absent) in the generated structure. As a reminder, this structure is represented by the graph variable x_G and by the Boolean variables x_i . Also, for each hexagon i of the coronenoid of size $k(n)$, we set the following constraints:

- $x_i = 1 \Rightarrow f_i \in P_+ \cup P_\circ \cup \{0\}$ (if the hexagon i is present in x_G , it is involved in the fragment as a positive or neutral hexagons or it does not participate in the fragment),
- $x_i = 0 \Rightarrow f_i \in P_- \cup P_\circ \cup \{0\}$ (if the hexagon i is absent, it is involved in the fragment as a negative or neutral hexagons or it does not participate in the fragment),
- $f_i \in P_+ \Rightarrow x_i = 1$ (if the hexagon i participates in the fragment as a positive hexagon, it is necessarily present),
- $f_i \in P_- \Rightarrow x_i = 0$ (if the hexagon i participates in the fragment as a negative hexagon, it is necessarily absent).

Next, we need to define the bijection that establishes that the constructed fragment satisfies the pattern P . In other words, we need to guarantee that exactly n_P hexagons of the structure must correspond to n_P hexagons of the pattern P . Also, for each hexagon j of the pattern, we add the global constraint¹ $\text{Count}(\{f_1, \dots, f_{n_c}\}, \{j\}) = 1$ if $j \in P_+$ (≤ 1 otherwise). The value 0 is obtained in the case where a negative or neutral hexagon is outside the coronenoid of size $k(n)$. In other words, a part of the pattern overflows from this coronenoid, but only for negative or neutral hexagons (which would then be absent). By doing so, we avoid introducing additional variables (and associated constraints) to represent the k_P layers of absent hexagons used in the formal definition of fragment (see Definition 3).

The last step consists in defining the pattern itself. To do this, we consider the neighborhood links between each hexagon of the pattern. A hexagon can have up to six neighboring hexagons. For a given hexagon h , we consider its potential neighbors v_1 to v_6 in a clockwise

¹ As a reminder, the global constraint $\text{Count}(Y, V) \odot k$ is satisfied if the number of variables of Y assigned with a value in V satisfies the condition with respect to the operator \odot and the value k .

■ **Table 1** The compact table constraint describing the neighborhood for the pattern *deep bay*.

f_i	f_{v_1}	f_{v_2}	f_{v_3}	f_{v_4}	f_{v_5}	f_{v_6}	f_i	f_{v_1}	f_{v_2}	f_{v_3}	f_{v_4}	f_{v_5}	f_{v_6}
0	*	*	*	*	*	*	3	1	4	6	0	0	0
1	0	2	4	3	0	0	4	2	5	7	6	3	1
1	2	4	3	0	0	0	5	0	0	0	7	4	2
1	4	3	0	0	0	2	6	4	7	0	0	0	3
1	3	0	0	0	2	4	7	5	0	0	0	6	4
1	0	0	0	2	4	3							
1	0	0	2	4	3	0							
2	0	0	5	4	1	0							

direction, starting with the neighbor at the top right. From there, we list the different configurations taken by the neighbors depending on which the hexagon h participates in the fragment or not. More precisely, each configuration is a tuple composed of one integer per neighbor. This integer is a non-zero value j if the neighbor participates in the fragment as the hexagon j of the pattern P , 0 otherwise. For each position of the hexagon h in the pattern P , we consider six possible configurations in order to take into account the 60° rotations of the pattern. This is necessary to generate all the structures because the model \mathcal{M} imposes the existence of hexagon(s) on the top and left edges of the considered coronenoid. Note that from a given configuration, applying a 60° rotation is equivalent to performing a circular permutation at the tuple level. For example, in Table 1, we list all the possible neighborhood configurations when the hexagon is in position 1 in the pattern *deep bay*, the numbering of the hexagons being that of Figure 2. For the other positions, we give only one configuration by lack of space. These configurations will be used to define the relation associated with compact table constraints [32]. We consider one table constraint per hexagon h of the coronenoid of size $k(n)$ whose scope involves the variable f_h and each variable f_i associated with a neighbor of h in $B_h^{c(k(n))}$. For hexagons at the edge of the coronenoid, we keep only the rows of the table whose neighbors participating in the fragment correspond to hexagons (whatever their nature) inside the coronenoid or to negative or neutral hexagons outside the coronenoid. Then, we make a projection of these lines on the present neighbors and the variable f_h .

4.3 Third Model

A fragment of order k of a benzenoid B corresponds to a connected subgraph of B_h^k . Thus, determining whether there exists a fragment satisfying a pattern P in a benzenoid B is, in some way, the same as determining whether there exists a subgraph in $B_h^{k_P}$ isomorphic to P_h . However, this is not exactly the usual subgraph isomorphism problem, but one of its variants taking into account the labeling of vertices and edges. This does not change the complexity of the decision problem which remains NP-complete. Fortunately, we do not need to tackle this problem because, in our approach, we will, by construction, directly produce structures satisfying the pattern.

We now present our model \mathcal{M}_{i_3} . Starting again from the general model \mathcal{M} , we add one variable s_i per hexagon of the pattern P (whatever its nature). Each variable s_i has domain $\{1, \dots, n'_c\}$ with n'_c the number of hexagons of the coronenoid of size $k(n) + k_P$. We exploit a coronenoid of size $k(n) + k_P$, instead of $k(n)$, because we need to surround the coronenoid of size $k(n)$ with k_P crowns of absent hexagons. Note that this has no impact on the graph

variable x_G or on the variables x_i because we are adding hexagons that are known not to be present in the structure under consideration. The variable s_i has value j if the i -th hexagon of the pattern P is the j -th hexagon of the coronenoid of size $k(n) + k_P$. By convention, values of j between 1 and n_c correspond to hexagons present in the coronenoid of size $k(n)$. We then add the following constraints to express the notion of isomorphism:

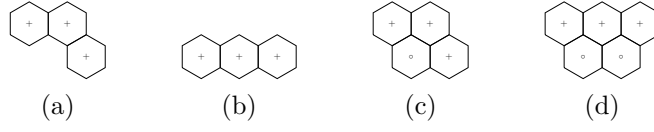
- *Injectivity*: The hexagons participating in the fragment must be pairwise different. This is imposed thanks to the global constraint `alldifferent`($\{s_1, \dots, s_{n_P}\}$). This also ensures that n_P hexagons of x_G participate in the fragment.
- *Edge preservation*: We must guarantee that two neighboring vertices of P_h correspond to two neighboring vertices in the hexagon graph of the coronenoid of size $k(n)$. Also, for each edge $\{i, i'\}$ of P_h (whatever its nature), we set a table constraint on s_i and $s_{i'}$ whose relation contains all pairs (j, j') such that $\{j, j'\}$ is an edge of the hexagon graph of the coronenoid of size $k(n) + k_P$.

This part of the model is inspired by the model of the subgraph isomorphism problem presented in [18]. However, it should be noted that, in our case, the graph in which the subgraph is searched is not known in advance, as it is the graph we wish to construct. Also, in our model, we circumvent this difficulty by considering the hexagon graph of the coronenoid of size $k(n) + k_P$.

Concerning the labeling, by definition, the labeling of the edges follows from that of the vertices. The labeling of the vertices is directly taken into account by definition of the variables s_i . It only remains to express the adequacy between the labeling of the vertices and the existence of the hexagons thanks to the following constraints:

- $\forall i \in P_+, \forall j \in \{1, \dots, n_c\}, s_i = j \Rightarrow x_j = 1$ (if the positive hexagon i of P corresponds to the hexagon j in x_G , j must be present),
- $\forall i \in P_+, \forall j \in \{1, \dots, n_c\}, x_j = 0 \Rightarrow s_i \neq j$ (if the hexagon j of x_G is absent, it cannot correspond to a positive hexagon i of P),
- $\forall i \in P_-, \forall j \in \{1, \dots, n_c\}, s_i = j \Rightarrow x_j = 0$ (if the negative hexagon i of P corresponds to the hexagon j in x_G , j must be absent), and
- $\forall i \in P_-, \forall j \in \{1, \dots, n_c\}, x_j = 1 \Rightarrow s_i \neq j$ (if the hexagon j of x_G is present, it cannot correspond to a negative hexagon i of P).

We now turn to the limitations of reasoning in terms of subgraph isomorphism from the perspective of chemistry. Figures 4(a)-(b) describe two patterns based on three positive hexagons and whose hexagon graphs are isomorphic. It turns out that the two corresponding molecules do not have the same chemical properties. However, if we ask Choco to produce the structures corresponding to each of these two patterns based on the model \mathcal{M}_{i_3} , we will obtain the same solutions. Also, to overcome this problem, we add a preprocessing step before the generation of the instance to solve. This step consists in adding neutral hexagons so that every edge of the hexagon graph of the pattern appears in at least one triangle (i.e. a clique of size 3). A triangle in the hexagon graph represents three hexagons which are pairwise adjacent. It thus characterizes a unique configuration (within one axial symmetry or 60° rotation). This preprocessing can be implemented by going through the hexagons of the initial pattern from top to bottom and from left to right. For lack of space, we do not detail this algorithm. Figures 4(c)-(d) present the patterns thus completed related to the patterns of Figures 4(a)-(b). Note that it is not always necessary to add neutral hexagons. For example, the pattern deep bay remains unchanged because each edge of its hexagon graph is already involved in at least one triangle.



■ **Figure 4** The limits of reasoning in terms of subgraph isomorphism with two different patterns (a) and (b) having isomorphic hexagon graphs. The patterns (a) and (b) after preprocessing (c)–(d).

5 Generating Structures Including Several Patterns

In this section, we are interested in generating structures containing several patterns simultaneously. Let $E_P = \{P^1, \dots, P^\ell\}$ be the set of these patterns. The existence of several patterns raises the question of how they can interact with each other. We list here three cases that make sense from a chemical point of view:

- (1) Patterns can share hexagons (regardless of their nature),
- (2) Patterns can share only absent hexagons (i.e. it is allowed to share the vacuum),
- (3) The patterns are pairwise disjoint.

A first naive approach to solve this multi-pattern problem is to solve a collection of single pattern problems. This would require enumerating all the single patterns that could be constructed on the basis of the patterns in E_P . But, given the combinatorics, this approach seems to be out of the question. Therefore we propose below to adapt the models we present in the previous section.

5.1 First Model

As usual, we start with the general model \mathcal{M} . Then, we add, for each pattern P^j of E_P , a set of variables e_i^j equivalent to the variables e_i for a single pattern P in the model \mathcal{M}_{i_1} as well as the associated sum constraint. Of course, this assumes to have computed in advance all the possible fragments of each pattern of E_P . This defines the model $\mathcal{M}_{m_1}^1$.

In order to obtain pairwise disjoint patterns (model $\mathcal{M}_{m_1}^3$), one must add to the model $\mathcal{M}_{m_1}^1$ the mutual exclusion clauses $e_i^j = 0 \vee e_{i'}^{j'} = 0$ for each pair of overlapping fragments $\{F_i^j, F_{i'}^{j'}\}$ (i.e. fragments such that $(F_{i+}^j \cup F_{i-}^j \cup F_{i\circ}^j \cup F_{i*}^j) \cap (F_{i'+}^{j'} \cup F_{i'-}^{j'} \cup F_{i'\circ}^{j'} \cup F_{i'*}^{j'}) \neq \emptyset$).

To share only vacuum (model $\mathcal{M}_{m_1}^2$), we add, to the model $\mathcal{M}_{m_1}^1$, constraints of the form $e_i^j = 0 \vee e_{i'}^{j'} = 0$ as soon as F_i^j and $F_{i'}^{j'}$ can share a present hexagon (i.e. if $(F_{i+}^j \cap F_{i'+}^{j'}) \cup (F_{i+}^j \cap F_{i'\circ}^{j'}) \cup (F_{i\circ}^j \cap F_{i'+}^{j'}) \neq \emptyset$). Otherwise, if they share neutral hexagons, these hexagons must be absent from the structure, which is ensured by posting the constraint $(e_i^j = 1 \wedge e_{i'}^{j'} = 1) \Rightarrow x_h = 0$ for each hexagon $h \in F_{i\circ}^j \cap F_{i'\circ}^{j'}$.

5.2 Second Model

We start with the general model \mathcal{M} . Then, we add, for each pattern P^j of E_P , a set of variables f_i^j equivalent to the variables f_i for a single pattern P in the model \mathcal{M}_{i_2} as well as all the associated constraints. However, for each table constraint defining the pattern P^j , we introduce a Boolean variable t^j into its scope. This variable is set to 1 if the associated configuration is obtained after applying an axial symmetry on P^j , 0 otherwise. Thus, the table will list all valid configurations obtained from the pattern P^j or its image by an axial symmetry. Taking into account axial symmetries in the case of the inclusion of several patterns is required in order to list all the possibilities of combining the patterns with each other. Several axes of symmetries are possible. However, it is sufficient to consider only one,

as the others can be obtained by combining with 60° rotations. The use of the variable t^j within each table constraint defining P^j guarantees that globally, one exploits either the pattern P^j if t^j is set to 0, or its image by axial symmetry otherwise. This avoids considering erroneous fragments of which one part would correspond to P^j and another to its image by symmetry. Note that, in the case of a single pattern, the use of this variable t^j would only add equivalent solutions to those already produced. The model we have just described corresponds to case (1). We denote it $\mathcal{M}_{m_2}^1$.

Then, to deal with the case (2) allowing sharing only absent hexagons, we take the model $\mathcal{M}_{m_2}^1$ and add mutual exclusion constraints for the present hexagons. This amounts to posting the following constraint for each hexagon h of the coronenoid of size $k(n)$: $x_h = 1 \Rightarrow \text{Count}(\{f_h^1, \dots, f_h^\ell\}, \{1 \dots, n_{E_P}\}) \leq 1$ with $n_{E_P} = \max_{P^j \in E_P} n_{P^j}$. In other words, if the hexagon h is present, it can participate in at most one fragment. We denote $\mathcal{M}_{m_2}^2$ this model.

Finally, in order to consider pairwise disjoint patterns (case (3)), we need to consider hexagons that might be shared outside the coronenoid of size $k(n)$. To do this, we define the order k_{E_P} of the set E_P as the maximum order of a pattern P^j of E_P . Then, we consider the model $\mathcal{M}_{m_2}^1$ but in a coronenoid of size $k(n) + k_{E_P}$. In other words, we add to $\mathcal{M}_{m_2}^1$ a variable f_i^j per hexagon located outside the coronenoid of size $k(n)$ and per pattern P^j . Since all hexagons are represented explicitly, the table constraints are defined taking into account these new variables and the *Count* constraints of $\mathcal{M}_{m_2}^1$ for negative or neutral hexagons j' of the P^j pattern are now of the form $\text{Count}(\{f_1^j, \dots, f_{n_c}^j\}, \{j'\}) = 1$. Finally, we add a mutual exclusion constraint $\text{Count}(\{f_h^1, \dots, f_h^\ell\}, \{1 \dots, n_{E_P}\}) = 1$ for each hexagon h of the coronenoid of size $k(n) + k_{E_P}$. We denote $\mathcal{M}_{m_2}^3$ this model.

5.3 Third Model

The principle is the same as for the two previous models. For each pattern P^j of E_P , we add to the model \mathcal{M} a set of variables s_i^j equivalent to the variables s_i used for a single pattern P in the model \mathcal{M}_{i_3} as well as all the associated constraints. By doing so, we obtain the model $\mathcal{M}_{m_3}^1$ corresponding to case (1). Since the model \mathcal{M}_{i_3} depends on the order of the considered pattern, the generated structures will have to be embedded in a coronenoid of size $k(n) + k_{E_P}$. Of course, as in \mathcal{M}_{i_3} , each pattern must be preprocessed beforehand in order to remove any ambiguity.

Then, we can extend this model to the model $\mathcal{M}_{m_3}^3$ in order to take into account pairwise disjoint patterns, by adding the mutual exclusion constraint $\text{alldifferent}(\{s_1^1, \dots, s_{n_{P^1}}^1\} \cup \dots \cup \{s_1^\ell, \dots, s_{n_{P^\ell}}^\ell\})$.

Finally, the model $\mathcal{M}_{m_3}^2$ corresponding to case (2) is obtained from the model $\mathcal{M}_{m_3}^1$ model, by adding the following constraints:

- $\text{alldifferent}(\{s_i^j | j \in \{1, \dots, \ell\}, i \in P_+^j\})$ which expresses that the positive hexagons in x_G are pairwise disjoint,
- $\forall j, j' \in \{1, \dots, \ell\}, j < j', \forall i \in P_o^j, \forall i' \in P_o^{j'}, s_i^j = s_{i'}^{j'} \Rightarrow s_i^j > n_c \vee \text{Element}(\{x_z | z \in \{1, \dots, n_c\}\}, s_i^j) = 0)^2$ which expresses the fact that if two neutral hexagons designate the same hexagon of x_G , then the corresponding vertex does not appear in x_G .
- $\forall j, j' \in \{1, \dots, \ell\}, j \neq j', \forall i \in P_o^j, \forall i' \in P_+^{j'}, s_i^j \neq s_{i'}^{j'}$ which prohibits having the same hexagon of x_G for a neutral hexagon and a positive hexagon of two different patterns.

² As a reminder, the constraint $\text{Element}(Y, j) \odot k$ is satisfied if the value of the j -th variable of Y satisfies the condition with respect to the operator \odot and the value k .

6 Others Problems About Patterns

We now turn to some related problems around patterns. First, we deal with the exclusion of a pattern before showing how to constraint the number of occurrences of a given pattern.

6.1 Generating Structures Excluding a Pattern

We now aim to generate all the structures having n hexagons and not containing a given pattern P . The reasoning followed for the models \mathcal{M}_{i_2} and \mathcal{M}_{i_3} seems to be unsuitable because we would have to guarantee that there exists no suitable f_i numbering or isomorphic subgraph. Therefore, we follow here the same reasoning as for the model \mathcal{M}_{i_1} . More precisely, we start from the model \mathcal{M} and add to it a variable e_i per possible fragment in a coronenoid of size $k(n)$. Each variable e_i is true if the constraint $\bigwedge_{j \in F_{i-}} x_j = 0 \wedge \bigwedge_{j \in F_{i+}} x_j = 1$ is satisfied (i.e. the fragment is present in the structure). Finally, we set a sum constraint $\sum_j e_j = 0$. This model is denoted $\mathcal{M}_{e_1}^1$. An equivalent formulation consists in representing directly each fragment F_i as a nogood $\bigvee_{j \in F_{i-}} x_j = 1 \vee \bigvee_{j \in F_{i+}} x_j = 0$, leading to a model denoted $\mathcal{M}_{e_1}^2$.

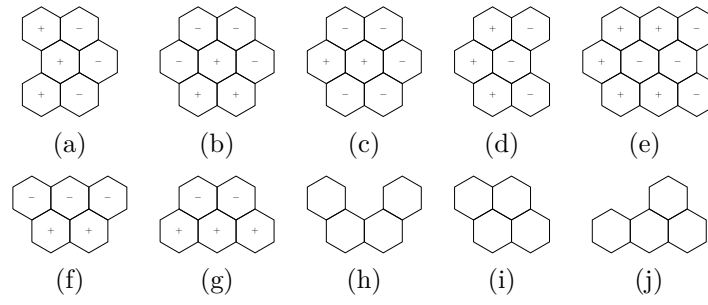
6.2 Generating Structures by Constraining the Number of Occurrences

Some constraints on the number of occurrences of a pattern P are easy to model. For example, to generate benzenoid structures with at least k pairwise disjoint occurrences of the pattern P , one can use any model among the models $\mathcal{M}_{m_1}^3$, $\mathcal{M}_{m_2}^3$ and $\mathcal{M}_{m_3}^3$ and a set E_P consisting of k times the pattern P . Others are a bit trickier. In order to make easier the expression of such constraints, we define a variable n_e which represents the number of occurrences of the pattern P contained in the generated structure and on which we will place the appropriate constraints according to the needs of the chemists. The variable n_e has the domain $\{0, \dots, k_{max}\}$ with k_{max} the maximum number of occurrences that the structure can contain. By default, if no information is given as input on k_{max} , we take $k_{max} = \left\lfloor \frac{n}{|P|} \right\rfloor$.

Once again, the approach followed in the model \mathcal{M}_{i_1} seems to be the most appropriate. Thus, starting from the model \mathcal{M}_{i_1} , we integrate the variable n_e . In addition to the variables e_i introduced for each possible fragment, we add a Boolean variable e'_i per fragment. The variable e'_i is true if the fragment F_i is present in the pattern. This is ensured by adding, for each fragment F_i the constraint $\bigwedge_{j \in F_{i-}} x_j = 0 \wedge \bigwedge_{j \in F_{i+}} x_j = 1 \Rightarrow e'_i = 1$. Then, as some fragments may share some hexagons, we add mutual exclusion constraints, with, for each hexagon h , the constraint $\sum_{i|h \in F_i} e'_i \geq 1 \Rightarrow \sum_{i|h \in F_i} e_i = 1$ (given that $F_i = F_{i+} \cup F_{i-} \cup F_{i\circ} \cup F_{i*}$). Thus, this guarantees that if a hexagon participates simultaneously in several fragments, only one of these fragments is considered as present. Finally, the constraint $\sum_j e_j = n_e$ allows us to compute the number of occurrences on which we can then easily put any arithmetic constraint. It is also possible to use this variable to find the structures maximizing the number of occurrences of the pattern.

7 Experiments

In this section, we assess empirically the different proposed models. To this end, we consider the eight patterns described in Figures 5(a)-(g) and Figure 3(a) from [34] and vary the number n of hexagons present in the structures from the number of positive hexagons in



■ **Figure 5** The patterns used as benchmarks in addition to the pattern *deep bay*: *armchair edge* (a), *C₃H₃ protusion* (b), *C₄H₄ protusion* (c), *shallow armchair bay* (d), *ultra deep bay* (e), *zigzag bay* (f), shortened to *zigzag* in [26], and *zigzag edge* (g). Two of the four benzenoid structures of four hexagons containing an instance of the pattern *armchair edge* (h) and (i). One of the three benzenoid structures of four hexagons containing no instance of the pattern *armchair edge* (j).

■ **Table 2** The number of instances which are successfully processed (#I) and the related cumulative runtime in hours (Time) for each possible variable and value heuristics and for model \mathcal{M}_{i_1} , \mathcal{M}_{i_2} and \mathcal{M}_{i_3} .

	\mathcal{M}_{i_1}				\mathcal{M}_{i_2}				\mathcal{M}_{i_3}			
	<i>inc</i>		<i>desc</i>		<i>inc</i>		<i>desc</i>		<i>inc</i>		<i>desc</i>	
	#I	Time	#I	Time	#I	Time	#I	Time	#I	Time	#I	Time
<i>dom</i>	55	1.44	55	1.42	55	1.51	55	1.49	55	1.54	55	1.52
<i>dom/wdeg</i>	47	23.58	48	17.71	47	22.07	48	16.90	48	20.93	48	17.40
<i>dom/wdeg^{ca.cd}</i>	50	12.56	55	4.85	50	15.47	50	15.35	49	16.47	50	14.76
<i>CHS</i>	43	27.06	41	28.97	47	23.04	48	20.69	48	20.42	48	17.43

the pattern to 9. This allows us to produce 55 instances (resp. 135) of the problem of generating structures containing/excluding one pattern (resp. containing two patterns). Our implementation is based on Choco (v. 4.10.7). We consider four state-of-the-art variable ordering heuristics namely *dom/wdeg* [2], *dom/wdeg^{ca.cd}* [33], *CHS* [15] and *dom*. This latter chooses as next variable the first variable in the lexicographical ordering having the smallest domain. Regarding the value ordering heuristic, we use the heuristics *inc* and *desc* which choose respectively the smallest and the largest value first. The experiments are carried out on DELL PowerEdge R440 servers with an Intel Xeon 4112 2.6 GHz processor and 32 GB of memory. The runtime for processing an instance is limited to 2 hours. We do not compare our approach with an existing method because, to our knowledge, no such method has been proposed yet, probably due to the fact that this line of research has emerged recently.

First, from Table 2, we can observe that, whatever the model (among \mathcal{M}_{i_1} , \mathcal{M}_{i_2} and \mathcal{M}_{i_3}) or the variable heuristic, the best results are generally obtained with the value heuristic *desc*. This can be explained by the fact that each model mainly involves Boolean variables. For instance, assigning 1 to a variable x_i amounts to create a hexagon and so allow us to exploit more quickly most of the constraints of the general model \mathcal{M} . Now, regarding the variable heuristic, the most sophisticated heuristics are not those leading to the best results. Indeed, whatever the model, the heuristic *dom* turns to be the more relevant for our problem. Moreover, as shown in Figure 6(a), *dom* performs better than *dom/wdeg^{ca.cd}* (which is the best variable heuristic after *dom*) on all the considered instances. This may seem surprising, but, if we look closely at the definition of *dom*, we can note that it corresponds to start the search with the hexagons located on the top edge of the coronenoid. At the same time, the more sophisticated heuristics may be penalized by the uniformity of the problem. Finally, for

■ **Table 3** The number of instances which are successfully processed (#I) and the related cumulative runtime in hours (Time) for each possible variable and value heuristics and for model $\mathcal{M}_{m_1}^3$, $\mathcal{M}_{m_2}^3$ and $\mathcal{M}_{m_3}^3$.

	$\mathcal{M}_{m_1}^3$				$\mathcal{M}_{m_2}^3$				$\mathcal{M}_{m_3}^3$			
	<i>inc</i>		<i>desc</i>		<i>inc</i>		<i>desc</i>		<i>inc</i>		<i>desc</i>	
	#I	Time	#I	Time	#I	Time	#I	Time	#I	Time	#I	Time
<i>dom</i>	135	6.82	135	6.85	129	44.37	135	17.76	135	7.68	135	7.78
<i>dom/wdeg</i>	100	100.16	113	74.69	105	92.48	108	77.77	106	82.99	113	63.06
<i>dom/wdeg^{ca.cd}</i>	121	51.45	134	25.64	93	101.77	105	81.44	96	102.07	105	79.44
<i>CHS</i>	85	116.19	83	120.62	86	106.24	70	136.93	108	75.84	110	66.61

given value and variable heuristics, we can note that the models often obtain close results. If we focus our attention on *dom* and *desc* (see Figures 6(b)-(d)), \mathcal{M}_{i_1} turns out to perform slightly better than \mathcal{M}_{i_2} , which itself is better than \mathcal{M}_{i_3} . Maybe, this could be explained by the fact that all the models are based on the general model \mathcal{M} to which some variables and constraints are added. Indeed, at the end, the models have similar numbers of constraints while the model \mathcal{M}_{i_1} requires a few more variables than the other models.

If we are now interested in the generation of structures containing two given patterns, we can note that the observed trends in Table 3 are quite similar to those obtained for a single pattern. Again, the value heuristic *desc* leads to the best results. Regarding the variable heuristic, *dom* is once more the most relevant and robust one. A slight difference from the single pattern case is that the efficiency of the other variables heuristic seems to depend on the model we consider. Beyond, we can observe that the differences between the models, whatever the variable heuristic, are more pronounced. Globally, the model $\mathcal{M}_{m_1}^3$ turns out to be the best one followed by the model $\mathcal{M}_{m_3}^3$ while the model $\mathcal{M}_{m_2}^3$ turns out to perform worst. This is clearly visible on Figures 6(e)-(g) when considering the heuristics *dom* and *desc*. This result seems to be correlated with the number of constraints which is twice as large for the model $\mathcal{M}_{m_2}^3$ than for $\mathcal{M}_{m_1}^3$ or $\mathcal{M}_{m_3}^3$.

Regarding the exclusion of a given pattern, the trends we observe for value and variable heuristics are similar to previous comparisons. By lack of place, we do not provide more details. If we compare the two models $\mathcal{M}_{e_1}^1$ and $\mathcal{M}_{e_1}^2$ (see Figure 6(h)), it appears that the latter is the most efficient. Using the heuristics *dom* and *desc*, both achieve the same exploration of the search space, but the model $\mathcal{M}_{e_1}^2$ does not consider additional variables w.r.t. the general model \mathcal{M} . Moreover, it only requires some clauses as additional constraints.

Finally, in Figure 7(b), we compare the average number of solutions (some of them are depicted in Figures 5(i)-(j) and 7(a)) with the number of benzenoid structures depending on the considered problem and the number n of hexagons. This figure first allows us to observe the growth of the number of structures with n . Then, we can also notice that we have to compute a large number of benzenoid structures. For instance, for the inclusion of a single pattern, we have to consider about 70% of all the benzenoid structures.

8 Conclusion and Perspectives

We have presented an approach based on CP to generate exhaustively benzenoid structures satisfying certain constraints around patterns. For this purpose, several models have been considered and compared. The model based on the identification of all the possible locations of a fragment turns out to be the most robust. It leads to an efficient solving while being able to deal with several questions about patterns (inclusion or exclusion, number of occurrences, ...).

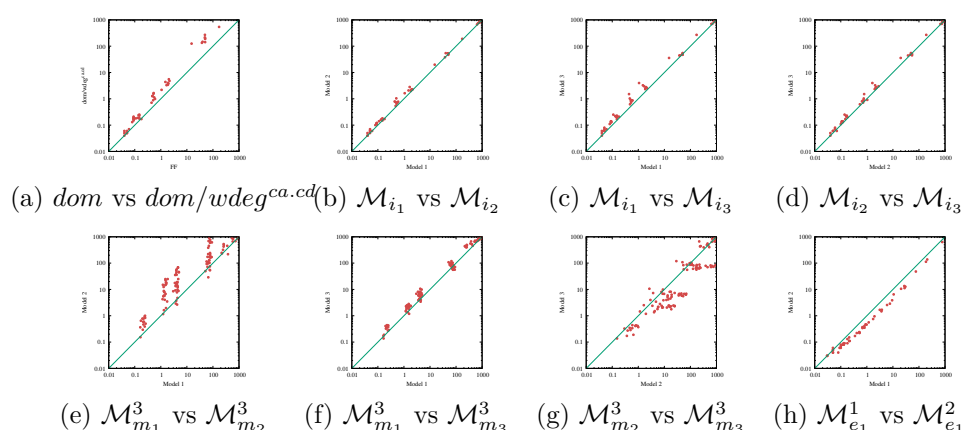


Figure 6 Comparison of the variable heuristics *dom* and *dom/wdeg^{ca.cd}* based on the runtime (in seconds) for \mathcal{M}_{i_1} and the value heuristic *desc*. Pairwise comparison of models based on the runtime (in seconds) when using the variable heuristic *dom* and the value heuristic *desc* (b)-(h).

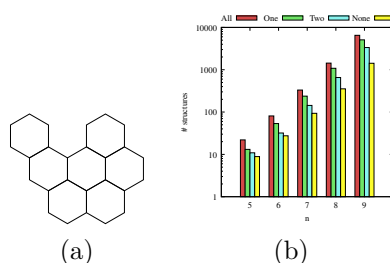


Figure 7 One of the twenty-five benzenoid structures of seven hexagons containing the patterns *armchair edge* and *deep bay* (a). Number of benzenoid structures (all) and average number of benzenoid structures containing a given pattern (one), two given patterns (two) or excluding a given pattern (none) (b).

In a way, we have proposed a modeling brick per issue which can be combined each other or with global properties (e.g. those defined in [5]) depending on the needs of chemists. To our knowledge, this work provides chemists with the first tool for generating benzenoid structures satisfying certain conditions on their edge topology. It would be useful in validating their theoretical models or identifying the most promising benzenoid structures before trying to synthesize them, what are currently hot topics in chemistry.

As a consequence, a first perspective of this work will be to study its repercussions from the viewpoint of chemistry (e.g. by generating all the structures containing some given patterns and applying to them some theoretical chemistry tools [7, 4]). Concerning the modeling, other forms of interaction between two patterns can be of interest to chemists (e.g. by sharing a positive hexagon only if the two patterns do not use the same bonds). Finally, several avenues can be explored to improve the practical efficiency of the approach.

References

- 1 M. R. Ajayakumar, Ji Ma, Andrea Lucotti, Karl Sebastian Schellhammer, Gianluca Serra, Evgenia Dmitrieva, Marco Rosenkranz, Hartmut Komber, Junzhi Liu, Frank Ortmann, Matteo Tommasini, and Xinliang Feng. Persistent peri-Heptacene: Synthesis and In Situ Characterization. *Angew. Chem. Int. Ed.*, 2021. doi:10.1002/anie.202102757.
- 2 Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting Systematic Search by Weighting Constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, pages 146–150, 2004.

- 3 Gunnar Brinkmann, Gilles Caporossi, and Pierre Hansen. A Constructive Enumeration of Fusenes and Benzenoids. *Journal of Algorithms*, 45(2), 2002. doi:10.1016/S0196-6774(02)00215-8.
- 4 Yannick Carissan, Chisom-Adaobi Dim, Denis Hagebaum-Reignier, Nicolas Prcovic, Cyril Terrioux, and Adrien Varet. Computing the Local Aromaticity of Benzenoids Thanks to Constraint Programming. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 673–689, 2020. doi:10.1007/978-3-030-58475-7_39.
- 5 Yannick Carissan, Denis Hagebaum-Reignier, Nicolas Prcovic, Cyril Terrioux, and Adrien Varet. Using Constraint Programming to Generate Benzenoid Structures in Theoretical Chemistry. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 690–706, 2020. doi:10.1007/978-3-030-58475-7_40.
- 6 Ying Chen, Chaojun Lin, Zhixing Luo, Zhibo Yin, Haonan Shi, Yanpeng Zhu, and Jiaobing Wang. Double π -Extended Undecabenz[7]helicene. *Angew. Chem. Int. Ed.*, 60(14):7796–7801, 2021. doi:10.1002/anie.202014621.
- 7 Zhongfang Chen, Chaitanya S. Wannere, Clémence Corminboeuf, Ralph Puchta, and Paul von Ragué Schleyer. Nucleus-Independent Chemical Shifts (NICS) as an Aromaticity Criterion. *Chem Rev*, 105:3842–3888, 2005. doi:10.1021/cr030088.
- 8 Kwan Yin Cheung, Kosuke Watanabe, Yasutomo Segawa, and Kenichiro Itami. Synthesis of a zigzag carbon nanobelt. *Nat. Chem.*, 13(3):255–259, 2021. doi:10.1038/s41557-020-00627-5.
- 9 J. Cyvin, J. Brunvoll, and B. N. Cyvin. Search for Concealed Non-Kekuléan Benzenoids and Coronoids. *J. Chem. Inf. Comput. Sci.*, 29(4):237, 1989. doi:10.1021/ci00064a002.
- 10 Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved Static Symmetry Breaking for SAT. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 104–122, 2016. doi:10.1007/978-3-319-40970-2_8.
- 11 Tim Dumsclaff, Yanwei Gu, Giuseppe M. Paternò, Zijie Qiu, Ali Maghsoumi, Matteo Tomasini, Xinliang Feng, Francesco Scotognella, Akimitsu Narita, and Klaus Müllen. Hexa-peri-benzocoronene with two extra K-regions in an ortho-configuration. *Chem. Sci.*, 11(47):12816–12821, 2020. doi:10.1039/D0SC04649C.
- 12 Jean-Guillaume Fages. *Exploitation de structures de graphe en programmation par contraintes*. PhD thesis, École des mines de Nantes, France, 2014.
- 13 Jean-Guillaume Fages, Xavier Lorca, and Charles Prud’homme. Choco solver user guide documentation. <https://choco-solver.readthedocs.io/en/latest/>.
- 14 Kei Fujise, Eiji Tsurumaki, Kan Wakamatsu, and Shinji Toyota. Construction of Helical Structures with Multiple Fused Anthracenes: Structures and Properties of Long Expanded Helicenes. *Chemistry – A European Journal*, 27(14):4548–4552, March 2021. doi:10.1002/chem.202004720.
- 15 Djamal Habet and Cyril Terrioux. Conflict History based Search for Constraint Satisfaction Problem. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC)*, pages 1117–1122, 2019. doi:10.1145/3297280.3297389.
- 16 Sindhu Kancherla and Kåre B. Jørgensen. Synthesis of Phenacene–Helicene Hybrids by Directed Remote Metalation. *J. Org. Chem.*, 85(17):11140–11153, 2020. doi:10.1021/acs.joc.0c01097.
- 17 Ashok Keerthi, Carlos Sánchez-Sánchez, Okan Deniz, Pascal Ruffieux, Dieter Schollmeyer, Xinliang Feng, Akimitsu Narita, Roman Fasel, and Klaus Müllen. On-surface Synthesis of a Chiral Graphene Nanoribbon with Mixed Edge Structure. *Chem. – Asian J.*, 15(22):3807–3811, 2020. doi:10.1002/asia.202001008.
- 18 Christophe Lecoutre and Olivier Roussel, editors. *Proceedings of the 2018 XCSP3 Competition*, 2018. arXiv:1901.01830.
- 19 Junzhi Liu and Xinliang Feng. Synthetic tailoring of graphene nanostructures with Zigzag-Edged topologies: Progress and perspectives. *Angewandte Chemie International Edition*, 59:2–18, 2020. doi:10.1002/anie.202008838.

- 20 Max M. Martin, Frank Hampel, and Norbert Jux. A Hexabenzocoronene-Based Helical Nanographene. *Chem. – Eur. J.*, 26(45):10210–10212, 2020. doi:10.1002/chem.202001471.
- 21 Shantanu Mishra, Doreen Beyer, Kristjan Eimre, Shawulienu Kezilebieke, Reinhard Berger, Oliver Gröning, Carlo A. Pignedoli, Klaus Müllen, Peter Liljeroth, Pascal Ruffieux, Xinliang Feng, and Roman Fasel. Topological frustration induces unconventional magnetism in a nanographene. *Nature Nanotechnology*, 15(1):22–28, 2020. doi:10.1038/s41565-019-0577-9.
- 22 Shantanu Mishra, Doreen Beyer, Kristjan Eimre, Junzhi Liu, Reinhard Berger, Oliver Gröning, Carlo A. Pignedoli, Klaus Müllen, Roman Fasel, Xinliang Feng, and Pascal Ruffieux. Synthesis and Characterization of π -Extended Triangulene. *Journal of the American Chemical Society*, 141(27):10621–10625, 2019. doi:10.1021/jacs.9b05319.
- 23 Shantanu Mishra, Xuelin Yao, Qiang Chen, Kristjan Eimre, Oliver Gröning, Ricardo Ortiz, Marco Di Giovannantonio, Juan Carlos Sancho-García, Joaquín Fernández-Rossier, Carlo A. Pignedoli, Klaus Müllen, Pascal Ruffieux, Akimitsu Narita, and Roman Fasel. Large magnetic exchange coupling in rhombus-shaped nanographenes with zigzag periphery. *Nat. Chem.*, pages 1–6, 2021. doi:10.1038/s41557-021-00678-2.
- 24 Tadashi Mori. Chiroptical Properties of Symmetric Double, Triple, and Multiple Helicenes. *Chem. Rev.*, 121(4):2373–2412, 2021. doi:10.1021/acs.chemrev.0c01017.
- 25 Marvin Nathusius, Barbara Ejlli, Frank Rominger, Jan Freudenberger, Uwe H. F. Bunz, and Klaus Müllen. Chrysene-Based Blue Emitters. *Chemistry – A European Journal*, 26(66):15089–15093, 2020. doi:10.1002/chem.202001808.
- 26 Wenhui Niu, Ji Ma, Paniz Soltani, Wenhao Zheng, Fupin Liu, Alexey A. Popov, Jan J. Weigand, Hartmut Komber, Emanuele Poliani, Cinzia Casiraghi, Jörn Droste, Michael Ryan Hansen, Silvio Osella, David Beljonne, Mischa Bonn, Hai I. Wang, Xinliang Feng, Junzhi Liu, and Yiyong Mai. A Curved Graphene Nanoribbon with Multi-Edge Structure and High Intrinsic Charge Carrier Mobility. *J. Am. Chem. Soc.*, 142(43):18293–18298, 2020. doi:10.1021/jacs.0c07013.
- 27 Michele Pizzochero and Efthimios Kaxiras. Imprinting Tunable π -Magnetism in Graphene Nanoribbons via Edge Extensions. *J. Phys. Chem. Lett.*, 12(4):1214–1219, February 2021. doi:10.1021/acs.jpcllett.0c03677.
- 28 Zijie Qiu, Cheng-Wei Ju, Lucas Frédéric, Yunbin Hu, Dieter Schollmeyer, Grégory Pieters, Klaus Müllen, and Akimitsu Narita. Amplification of Dissymmetry Factors in π -Extended [7]- and [9]Helicenes. *J. Am. Chem. Soc.*, 143(12):4661–4667, March 2021. doi:10.1021/jacs.0c13197.
- 29 Zijie Qiu, Akimitsu Narita, and Klaus Müllen. Carbon nanostructures by macromolecular design from branched polyphenylenes to nanographenes and graphene nanoribbons. *Faraday Discussions*, 2020. Publisher: The Royal Society of Chemistry. doi:10.1039/D0FD00023J.
- 30 Georges Trinquier and Jean-Paul Malrieu. Predicting the Open-Shell Character of Polycyclic Hydrocarbons in Terms of Clar Sextets. *The Journal of Physical Chemistry A*, 122(4):1088–1103, 2018. doi:10.1021/acs.jpca.7b11095.
- 31 Mizuho Uryu, Taito Hiraga, Yoshito Koga, Yutaro Saito, Kei Murakami, and Kenichiro Itami. Synthesis of Polybenzoacenes: Annulative Dimerization of Phenylene Triflate by Twofold C-H Activation. *Angew. Chem.*, 132(16):6613–6616, 2020. doi:10.1002/ange.202001211.
- 32 Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Extending Compact-Table to Negative and Short Tables. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pages 3951–3957, 2017. URL: <https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14359/14122>.
- 33 Hugues Watez, Christophe Lecoutre, Anastasia Paparrizou, and Sébastien Tabary. Refining Constraint Weighting. In *Proceedings of the 31st IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 71–77, 2019. doi:10.1109/ICTAI.2019.00019.
- 34 Natalie Wohner, Pui K. Lam, and Klaus Sattler. Systematic energetics study of graphene nanoflakes: From armchair and zigzag to rough edges with pronounced protrusions and overcrowded bays. *Carbon*, 82:523–537, 2015. doi:10.1016/j.carbon.2014.11.004.

- 35 Zeming Xia, Sai Ho Pun, Han Chen, and Qian Miao. Synthesis of Zigzag Carbon Nanobelts through Scholl Reactions. *Angew. Chem. Int. Ed.*, 60(18):10311–10318, 2021. doi:10.1002/anie.202100343.
- 36 Xuan Yang, Frank Rominger, and Michael Mastalerz. Benzo-Fused Perylene Oligomers with up to 13 Linearly Annulated Rings. *Angew. Chem. Int. Ed.*, 60(14):7941–7946, 2021. doi:10.1002/anie.202017062.
- 37 Xuelin Yao, Wenhao Zheng, Silvio Osella, Zijie Qiu, Shuai Fu, Dieter Schollmeyer, Beate Müller, David Beljonne, Mischa Bonn, Hai I. Wang, Klaus Müllen, and Akimitsu Narita. Synthesis of Nonplanar Graphene Nanoribbon with Fjord Edges. *J. Am. Chem. Soc.*, 143(15):5654–5658, 2021. doi:10.1021/jacs.1c01882.
- 38 Cheng Zeng, Bohan Wang, Huanhuan Zhang, Mingxiao Sun, Liangbin Huang, Yanwei Gu, Zijie Qiu, Klaus Müllen, Cheng Gu, and Yuguang Ma. Electrochemical Synthesis, Deposition, and Doping of Polycyclic Aromatic Hydrocarbon Films. *J. Am. Chem. Soc.*, 143(7):2682–2687, February 2021. doi:10.1021/jacs.0c13298.

Combining VSIDS and CHB Using Restarts in SAT

Mohamed Sami Cherif 

Aix-Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

Djamal Habet 

Aix-Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

Cyril Terrioux 

Aix-Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

Abstract

Conflict Driven Clause Learning (CDCL) solvers are known to be efficient on structured instances and manage to solve ones with a large number of variables and clauses. An important component in such solvers is the branching heuristic which picks the next variable to branch on. In this paper, we evaluate different strategies which combine two state-of-the-art heuristics, namely the Variable State Independent Decaying Sum (VSIDS) and the Conflict History-Based (CHB) branching heuristic. These strategies take advantage of the restart mechanism, which helps to deal with the heavy-tailed phenomena in SAT, to switch between these heuristics thus ensuring a better and more diverse exploration of the search space. Our experimental evaluation shows that combining VSIDS and CHB using restarts achieves competitive results and even significantly outperforms both heuristics for some chosen strategies.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence

Keywords and phrases Satisfiability, Branching Heuristic, Restarts

Digital Object Identifier 10.4230/LIPIcs.CP.2021.20

Funding Supported by the French National Research Agency, project ANR-16-CE40-0028.

1 Introduction

Given a CNF Boolean formula ϕ , solving the Satisfiability (SAT) problem consists in determining whether there exists an assignment of the variables which satisfies ϕ . SAT is at the heart of many applications in different fields and is used to model a large variety of crafted and real-world problems [33, 17, 24]. It is the first decision problem proven to be NP-complete [16]. Nevertheless, modern solvers based on Conflict Driven Clause Learning (CDCL) [34] manage to solve instances involving a huge number of variables and clauses. An important component in such solvers is the branching heuristic which picks the next variable to branch on. The Variable State Independent Decaying Sum (VSIDS) [35] has been the dominant heuristic since its introduction two decades ago. Recently, Liang and al. devised a new heuristic for SAT, called Conflict History-Based (CHB) branching heuristic [29], and showed that it is competitive with VSIDS. In the last years, VSIDS and CHB have dominated the heuristics landscape as practically all the CDCL solvers presented in recent SAT competitions and races incorporate a variant of one of them.

In recent years, combining VSIDS and CHB has shown promising results. For instance, the MapleCOMSPS solver, which won several medals in the 2016 and 2017 SAT competitions, switches from VSIDS to CHB after a set amount of time, or alternates between both heuristics by allocating the same duration of restarts to each one [31, 28]. Yet, we still lack a thorough analysis on such strategies in the state of art as well as a comparison with new promising methods based on machine learning in the context of SAT solving. Indeed, recent research has also shown the relevance of machine learning in designing efficient search heuristics for SAT [29, 30, 25] as well as for other decision problems [42, 41, 36, 13]. One of the main



© Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 20; pp. 20:1–20:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

challenges is defining a heuristic which can have high performance on any considered instance. It is well known that a heuristic can perform very well on a family of instances while failing drastically on another. To this end, several reinforcement learning techniques can be used, specifically under the Multi-Armed Bandit (MAB) framework, to pick an adequate heuristic among CHB and VSIDS for each instance. These strategies also take advantage of the restart mechanism in modern CDCL algorithms to evaluate each heuristic and choose the best one accordingly. The evaluation is usually achieved by a reward function, which has to estimate the efficiency of a heuristic by relying on information acquired during the runs between restarts. In this paper, we want to compare these different strategies and, in particular, we want to know whether incorporating strategies which switch between VSIDS and CHB can achieve a better result than both heuristics and bring further gains to practical SAT solving.

The paper is organized as follows. An overview of CDCL algorithms is given in Section 2. The heuristics VSIDS and CHB as well as the Multi-Armed Bandit Problem are recalled in Section 3. Different strategies to combine VSIDS and CHB through restarts are described in Section 4 and experimentally evaluated and compared in Section 5. Finally, we conclude and discuss future work in Section 6.

2 Preliminaries

Let X be the set of propositional variables. A literal l is a variable $x \in X$ or its negation \bar{x} . A clause is a disjunction of literals. A formula in Conjunctive Normal Form (CNF) is a conjunction of clauses. An assignment $I : X \rightarrow \{true, false\}$ maps each variable to a Boolean value and can be represented as a set of literals. A literal l is satisfied by an assignment I if $l \in I$, else it is falsified by I . A clause is satisfied by an assignment I if at least one of its literals is satisfied by I , otherwise it is falsified by I . A CNF formula is satisfiable if there exists an assignment I which satisfies all its clauses, else it is unsatisfiable. Solving the Satisfiability (SAT) problem consists in determining whether a given CNF formula is satisfiable.

Although SAT is NP-complete [16], Conflict Driven Clause Learning [34] (CDCL) solvers are surprisingly efficient and manage to solve instances involving a huge number of variables and clauses. Such solvers are based on backtracking algorithms which rely on powerful branching heuristics as well as several solving techniques, namely Boolean Constraint Propagation (BCP), clause learning and restarts among others. In each step, BCP is applied to simplify the formula by propagating literals in unit clauses, i.e. clauses with one literal. If BCP is no longer possible, a branching heuristic picks a variable based on information acquired throughout the search. More importantly, when a conflict is detected, i.e. a clause is falsified by the current assignment, the steps of the algorithm are retraced and clauses involved in the conflict are resolved until the First Unit Implication Point (FUIP) in the implication graph [34]. The clause produced by this process is learnt, i.e. added to the formula. This enables to avoid revisiting an explored subspace of the search tree. Restarts are also an important component in CDCL solvers, initially introduced to deal with the heavy-tailed phenomena in SAT [19]. At the beginning of each restart, the solver parameters and its data structures are reinitialized in order to start the search somewhere else in the search space without discarding learnt clauses. There are two main restart strategies namely geometric restarts [40] and Luby restarts [32]. Most modern CDCL solvers use Luby restarts as it was shown that this policy outperforms geometric restarts [20].

3 Related Work

3.1 Branching Heuristics for SAT

The branching heuristic is one of the most important components in modern CDCL solvers and has a direct impact on their efficiency. It can be considered as a function that ranks variables using a scoring function, updated throughout the search. In this section, we describe two of the main state-of-the-art branching heuristics that we will consider in this work.

3.1.1 VSIDS

The Variable State Independent Decaying Sum (VSIDS) [35] has been the most used heuristic since its introduction around two decades ago. This heuristic maintains a floating point score for each variable, called activity and initially set to 0. When a conflict occurs, the activity of some variables is bumped, i.e. increased by 1. Furthermore, the variable activities are decayed periodically, usually after each conflict. More precisely, variable activities are multiplied by a decaying factor in $]0, 1[$. There are several variants of VSIDS. For instance, MiniSat [18] bumps the activities of variables appearing in the learnt clause while Chaff [35] does it for all the variables involved in the conflict, i.e. the resolved variables including those in the learnt clause.

3.1.2 CHB

The Conflict History-Based (CHB) branching heuristic was recently introduced in [29]. This heuristic based on the Exponential Recency Weighted Average (ERWA) [38] favors the variables involved in recent conflicts as in VSIDS. CHB maintains a score (or activity) $Q(x)$ for each variable x , initially set to 0. The score $Q(x)$ is updated when a variable x is branched on, propagated, or asserted using ERWA as follows:

$$Q(x) = (1 - \alpha) \times Q(x) + \alpha \times r(x).$$

The parameter $0 < \alpha < 1$ is the step-size, initially set to 0.4 and decayed by 10^{-6} after every conflict to a minimum of 0.06. $r(x)$ is the reward value for variable x which can decrease or increase the likelihood of picking x . Higher rewards are given to variables involved in recent conflicts according to the following formula:

$$r(x) = \frac{\text{multiplier}}{\text{Conflicts} - \text{lastConflict}(x) + 1}.$$

Conflicts denotes the number of conflicts that occurred since the beginning of the search. *lastConflict*(x) is updated to the current value of *Conflicts* whenever x is present in the clauses used by conflict analysis. *multiplier* is set to 1.0 when branching, propagating or asserting the variable that triggered the score update lead to a conflict, else it is set to 0.9. The idea is to give extra rewards for variables producing a conflict.

3.2 Multi-Armed Bandit Problem

A Multi-Armed Bandit (MAB) is a reinforcement learning problem consisting of an agent and a set of candidate arms from which the agent has to choose while maximizing the expected gain. The agent relies on information in the form of rewards given to each arm and collected through a sequence of trials. An important dilemma in MAB is the tradeoff between exploitation and exploration as the agent needs to explore underused arms often

enough to have a robust feedback while also exploiting good candidates which have the best rewards. The first MAB model, stochastic MAB, was introduced in [26] then different policies have been devised for MAB [1, 4, 6, 38, 39]. In recent years, there was a surge of interest in applying reinforcement learning techniques and specifically those related to MAB in the context of SAT solving. In particular, CHB [29] and LRB [30] (a variant of CHB) are based on ERWA [38] which is used in non-stationary MAB problems to estimate the average rewards for each arm. Furthermore, a new approach, called Bandit Ensemble for parallel SAT Solving (BESS), was devised in [27] to control the cooperation topology in parallel SAT solvers, i.e. pairs of units able to exchange clauses, by relying on a MAB formalization of the cooperation choices. MAB frameworks were also extensively used in the context of Constraint Satisfaction Problem (CSP) solving to choose a branching heuristic among a set of candidate ones at each node of the search tree [42] or at each restart [41, 13]. Finally, simple bandit-driven perturbation strategies to incorporate random choices in constraint solving with restarts were also introduced and evaluated in [36]. The MAB framework we introduce in the context of SAT in Section 4.2 is closely related to those introduced in [41, 36] in the sense that we also pick an adequate heuristic at each restart. In particular, our framework is closer to the one in [36] in terms of the number of candidate heuristics and the chosen reward function and yet it is different in the sense that we consider two efficient state-of-the-art heuristics instead of perturbing one through random choices which may deteriorate the efficiency of highly competitive SAT solvers.

4 Strategies to Combine VSIDS and CHB Using Restarts

In this section, we describe different strategies which take advantage of the restart mechanism in SAT solvers to combine VSIDS and CHB. First, we describe simple strategies which are either static or random. Then, we describe reinforcement learning strategies, in the context of a MAB framework, which rely on information acquired through the search to choose the most relevant heuristic at each restart.

4.1 Static and Random Strategies

Hereafter, we describe three different strategies, one of which is random while the other two are static. These strategies are defined as follows:

- **Random Strategy (RD_R)**: This strategy randomly picks a heuristic among VSIDS and CHB at each restart with equal probabilities, i.e. each heuristic is assigned a probability of $\frac{1}{2}$. This strategy is denoted RD_R in contrast with RD_D which randomly picks a heuristic at each decision.
- **Single Switch Strategy (SS)**: This strategy switches from VSIDS to CHB after a set amount of time and was used in the 2016 version of MapleCOMSPS [31]. We maintain the threshold time in which the heuristic is switched to $\frac{t}{2}$ where t is the timeout as in [31].
- **Round Robin Strategy (RR)**: This strategy alternates between VSIDS and CHB in the form of a round robin. This is similar to the strategy used in the latest version of MapleCOMSPS [28]. However, since we want to consider strategies which are independent from the restart policy and which only focus on choosing the heuristics, we do not assign equal amounts of restart duration (in terms of number of conflicts) to each heuristic and, instead, let the duration of restarts augment naturally with respect to the restart policy of the solver.

4.2 Multi-Armed Bandit Strategies

In order to use MAB strategies, we first introduce a MAB framework for SAT. Let $A = \{a_1, \dots, a_K\}$ be the set of arms for the MAB containing different candidate heuristics. The trials are the runs, i.e. executions, of the backtracking algorithm between restarts. The proposed framework selects a heuristic a_i where $i \in \{1 \dots K\}$ at each restart of the backtracking algorithm according to two different strategies that we will describe below. To choose an arm, MAB strategies generally rely on a reward function calculated during each run to estimate the performance of the chosen arm. The reward function plays an important role in the proposed framework and has a direct impact on its efficiency. We choose a reward function that estimates the ability of a heuristic to reach conflicts quickly and efficiently. If t denotes the current run, the reward of arm $a \in A$ is calculated as follows:

$$r_t(a) = \frac{\log_2(\text{decisions}_t)}{\text{decidedVars}_t}.$$

decisions_t and decidedVars_t respectively denote the number of decisions and the number of variables fixed by branching in the run t . Consequently, the earlier conflicts are encountered in the search tree and the fewer variables are instantiated, the greater the assigned reward value will be for the corresponding heuristic. $r_t(a)$ is clearly in $[0, 1]$ since $\text{decisions}_t \leq 2^{\text{decidedVars}_t}$. This reward function is adapted from the explored sub-tree measure introduced in [36].

Next, we describe strategies for MAB which belong to a family of well know strategies, referred to as Upper Confidence Bound (UCB) [1, 5, 4]. For this family, the following parameters are maintained for each candidate arm $a \in A$:

- $n_t(a)$ is the number of times the arm a is selected during the $t - 1$ previous runs,
- $\hat{r}_t(a)$ is the empirical mean of the rewards of arm a over the $t - 1$ previous runs.

We consider two UCB strategies, UCB1 and MOSS (Minimax Optimal Strategy in the Stochastic case). These strategies select the arm $a \in A$ that respectively maximizes $UCB1(a)$ and $MOSS(a)$ defined below. The left-side terms of $UCB1(a)$ and $MOSS(a)$ are identical and aim to put emphasis on arms that received the highest rewards. Conversely, the right-side terms ensure the exploration of underused arms. The main difference between UCB1 and MOSS is that the latter also takes into account the number of arms K .

$$UCB1(a) = \hat{r}_t(a) + \sqrt{\frac{4 \ln(t)}{n_t(a)}}$$

$$MOSS(a) = \hat{r}_t(a) + \sqrt{\frac{4}{n_t(a)} \ln \left(\max \left(\frac{t}{K \cdot n_t(a)}, 1 \right) \right)}$$

Finally, a strategy for MAB is evaluated by its expected cumulative regret, i.e. the difference between the cumulative expected value of the reward if the best arm is used at each restart and its cumulative value for all the runs. The expected cumulative regret R_T is formally defined below, where $a_t \in A$ denotes the arm chosen at run t and T denotes the total number of runs. In particular, UCB1 and MOSS respectively guarantee an expected cumulative regret no worse than $O(\sqrt{K \cdot T \cdot \ln T})$ and $O(\sqrt{K \cdot T})$ [5, 4].

$$R_T = \max_{a \in A} \sum_{t=1}^T \mathbf{E}[r_t(a)] - \sum_{t=1}^T \mathbf{E}[r_t(a_t)]$$

5 Experimental Evaluation

In this section, we describe our experimental protocol and then we evaluate and compare the different strategies presented in Section 4.

5.1 Experimental Protocol

We consider the benchmarks from the Main Track of the last three SAT Competitions/Races, totalling to 1,200 instances. For our experiments, we use the state-of-the-art solver Kissat [10] which won first place in the main track of the SAT Competition 2020. Note that this solver is a condensed and improved reimplement of the reference and competitive solver CaDiCaL [9, 10] in C. Data provided by Armin Bierre and Marjin Heule¹ show that Kissat is highly competitive and outperforms all-time winners of SAT competitions/Races particularly on the 2020 and 2019 Benchmarks. Kissat alternates between stable and non-stable phases as is the case in Cadical [9], renamed to stable mode and focused mode in [10]. VSIDS is used in stable phases which mainly target satisfiable instances. During non-stable phases targeting unsatisfiable instances, the solver uses the Variable Move-To-Front (VMTF) heuristic [37, 12], in which analyzed variables are moved to the front of the decision queue. It is important to note that the only modified components of the solver are the decision component and the restart component, i.e. all the other components as well as the default parameters of the solver are left untouched. Even the changes to the restart component are as minimal as possible, i.e. we maintain the phase alternation mechanism and the restart policies set for each mode as described in [10]. Furthermore, we maintain the VSIDS variant already implemented in Kissat, called Exponential VSIDS (EVSIDS) [8, 12], which is based on Chaff's where all analyzed variables are bumped after every conflict. Therefore, in the experimental evaluation, *VSIDS* corresponds to default Kissat. Moreover, we augment the solver with the heuristic CHB as specified in [29] except that we update the scores of the variables in the last decision level after BCP. In addition, we have implemented the MAB framework specified in Section 4 with $A = \{VSIDS, CHB\}$. The rewards for UCB1 and MOSS are both initialized by launching each heuristic once during the first restarts. Finally, The experiments are performed on Dell PowerEdge M620 servers with Intel Xeon Silver E5-2609 processors under Ubuntu 18.04 with a timeout of 5,000 s for each instance.

5.2 Decisions vs Restarts

First, we would like to emphasize that taking advantage of the restart mechanism to combine VSIDS and CHB was not an arbitrary choice. Indeed, we conducted an experiment to help us choose the appropriate level, i.e. decisions or restarts, to combine VSIDS and CHB. To this end, we implemented and tested the two random strategies RD_D and RD_R which randomly chose a heuristic among VSIDS and CHB respectively in each decision and in each restart. The average results (over 10 runs with different seeds) of RD_D and RD_R on the whole benchmark are reported in Table 1 and indicate that RD_R outperforms RD_D with a gain of more than 2% in terms of solved instances and 3.5% in terms of solving time with a penalty of 10,000 s for unsolved instances. This is not surprising as the structures needed for VSIDS and CHB need to be maintained and updated simultaneously which can be quite costly. On the other hand, they are used independently in RD_R during each restart, i.e. only the chosen heuristic is used and its structures updated during the restart. Furthermore, combining both

¹ Data available on <http://fmv.jku.at/kissat/>

■ **Table 1** Comparison between VSIDS, CHB, the different strategies and the VBS (over VSIDS and CHB) in terms of the number of solved instances in Kissat. For each row, the best results without considering the VBS are written in bold.

		VSIDS	CHB	RD_D	RD_R	SS	RR	UCB1	MOSS	VBS
Competition 2018 (400 instances)	SAT	160	159	160	164	163	165	167	168	169
	UNSAT	111	109	109	110	113	110	110	110	113
	TOTAL	271	268	268	274	276	275	277	278	282
Race 2019 (400 instances)	SAT	158	149	155	158	154	162	161	162	162
	UNSAT	97	95	95	96	96	96	96	97	99
	TOTAL	255	244	250	254	250	258	257	259	261
Competition 2020 (400 instances)	SAT	131	146	146	151	147	152	154	156	157
	UNSAT	121	119	117	120	118	120	120	122	123
	TOTAL	252	265	263	271	265	272	274	278	280
TOTAL (1,200 instances)	SAT	449	454	461	473	464	479	482	486	488
	UNSAT	329	323	321	326	327	326	326	329	335
	TOTAL	778	777	782	799	791	805	808	815	823

heuristics at the decision level can cause interference and may not allow each heuristic to conduct robust learning since they are being constantly interchanged. Surprisingly, both versions are competitive with CHB and VSIDS. In particular, RD_R outperforms them and solves, on average, 21 additional instances (+ 2.7%) compared to the best heuristic. This is due to randomization and diversification which help to avoid heavy tail phenomena in SAT and which can therefore improve the performance of SAT solvers [21, 19].

5.3 Comparison of Strategies

5.3.1 Number of Solved Instances

In Table 1, we present the results in terms of solved instances for CHB and VSIDS as standalone heuristics and for the different strategies presented in Section 4. We also include the results of the Virtual Best Solver (VBS) over VSIDS and CHB. Before discussing the results, we recall that “improving SAT solvers is often a cruel world. To give an idea, improving a solver by solving at least ten more instances (on a fixed set of benchmarks of a competition) is generally showing a critical new feature. In general, the winner of a competition is decided based on a couple of additional solved benchmarks” [3].

The results clearly indicate that MOSS outperforms VSIDS and CHB as well as all the other strategies. Indeed, MOSS manages to solve 37 additional instances in total (+4.8%) compared to the best heuristic (among VSIDS and CHB). The UCB1 (resp. RR) strategy is also competitive and manages to solve 30 (resp. 27) additional instances in total which corresponds to an increase of 3.9% (resp. 3.5%) in terms of solved instances compared to the best heuristic. The strategies UCB1 and RR remain comparable with a difference of 3 instances in favor of UCB1. SS also outperforms VSIDS and CHB although to a lesser degree as it solves 13 additional instances only which is worse than RD_R . If we focus on the individual yearly benchmarks, we observe that although the overall results obtained by VSIDS and CHB are comparable, they have different behaviours on each benchmark and yet MOSS, UCB1 and RR manage to capture the behaviour of the best heuristic and even outperform it on each individual benchmark. In particular, MOSS maintains its top rank on the individual benchmarks with an average of 8 (resp. 17) additional instances for each

one compared to the best (resp. worst) heuristic. Moreover, the results achieved by MOSS are very close to the VBS. Indeed it achieves 99% (resp. 99.6%) of the performance of the VBS on the whole benchmark in terms of the number of solved instances (resp. satisfiable instances) while the best heuristic does not exceed 95% (resp. 93%).

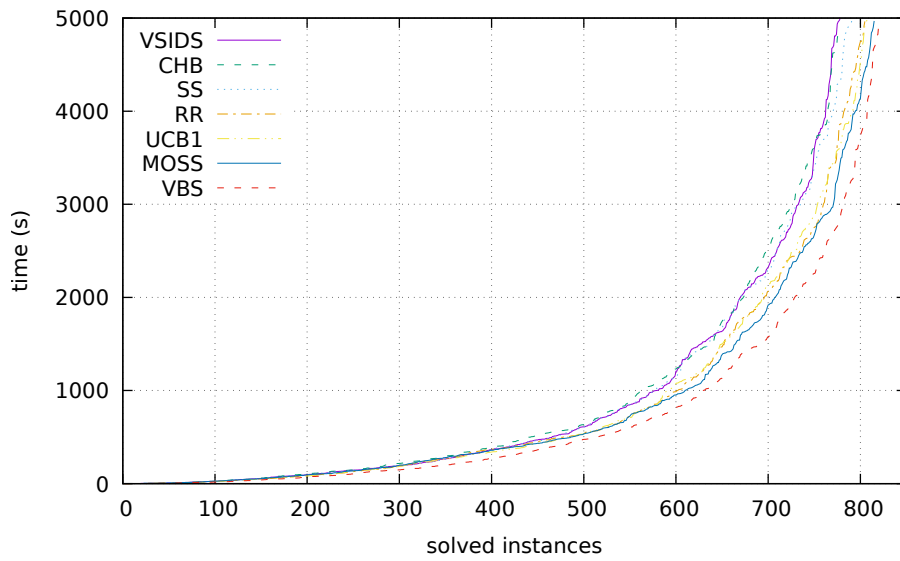
However, it is important to note that the gain is mainly in satisfiable instances whereas, for unsatisfiable instances, all the strategies (except RD_D) remain comparable to both heuristics and slightly outperform CHB but not VSIDS. Nevertheless, they remain competitive with VSIDS and particularly MOSS which solves the same number of unsatisfiable instances as VSIDS. This shows that MOSS is a robust strategy as it is able to improve the performance globally and on each individual benchmark without decreasing it for unsatisfiable instances. Note that the observed behaviour of these different strategies on unsatisfiable instances may be due to different factors. First, the results in terms of unsatisfiable instances seem very homogeneous for each year and are very close to the results obtained by the VBS as both heuristics (resp. the best heuristic) achieve more than 96% (resp. 98%) of its performance in terms of the number of unsatisfiable instances. Since our motivation is to bridge the gap between the heuristics and the VBS with these strategies, it is expected that this would be very difficult for unsatisfiable instances, for which the gap is very small already. It is also very difficult to simultaneously improve the performance on both satisfiable and unsatisfiable instances. Notice how SS which seems to work better for unsatisfiable instances especially in terms of solving time (refer to Section 5.3.2) fails on satisfiable instances compared to the three top strategies. Another possible factor for this behaviour is Kissat's restarting policy which alternates between the stable mode and focused mode [10]. The heuristics VSIDS and CHB are only used in the stable mode while the focused mode targets unsatisfiable instances. This may also help to explain the homogeneity of the results obtained by the solver for unsatisfiable instances with respect to the different heuristics and strategies.

5.3.2 Solving Time

In this section, we want to evaluate the different strategies in terms of solving time. In Figure 1, we represent the number of solved instances as a function of the CPU time for VSIDS, CHB, the static and MAB strategies and the VBS on the whole benchmark. One would think that MAB based strategies in this regard would be worse than the considered heuristics and/or other strategies as UCB1 and MOSS need to conduct continuous exploration in order to ensure the selection of the most adequate arm. This does not seem to be the case. In fact, conducting exploitation with the best arm and alternating the heuristics seems to offset this disadvantage. We observe that MOSS is the best strategy as it achieves 6.1% gain in terms of solving time on the whole benchmark compared to the best heuristic if we give a penalty 10,000 s to unsolved instances while UCB1, RR and SS respectively achieve a gain of 5.7%, 5.2% and 1.7%. This gain is substantial especially considering that we are working on the solver Kissat which won the SAT competition 2020 with a remarkable performance.

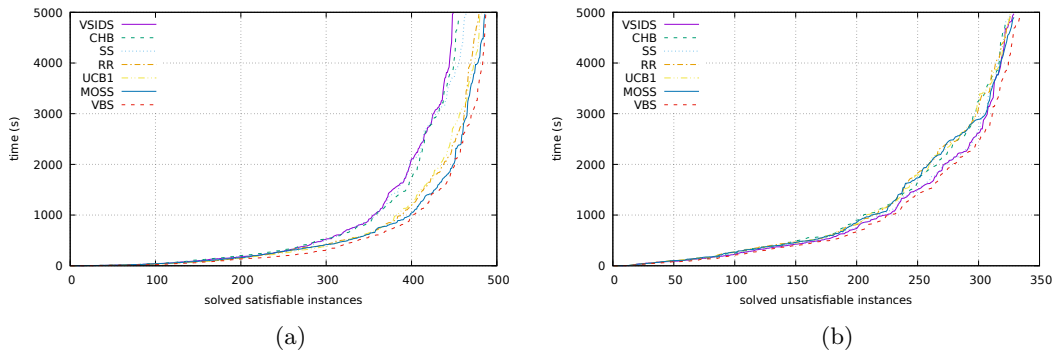
We represent in Figure 2 the number of solved satisfiable and unsatisfiable instances separately as a function of the CPU time for VSIDS, CHB, the static and MAB strategies and the VBS on the whole benchmark. Notice how the gap between MOSS and the VBS (and even UCB1 and RR) narrows if we consider the satisfiable instances only. On the other hand, these top three strategies present a small gap in terms of solving time for unsatisfiable instances compared to the best heuristic, i.e. VSIDS, while remaining comparable to CHB. In particular, MOSS shows better results with respect to VSIDS and SS for instances whose solving time exceeds 4,000 s. Surprisingly, although SS seems to be the worst strategy overall and remains globally comparable to VSIDS and CHB while achieving a slight gain

in solving time especially on instances whose solving time exceeds 4,000 s, it achieves the best results in terms of solving time for unsatisfiable instances and is comparable to VSIDS and the VBS in this regard. On the other hand, RR and UCB1 achieve substantial gain while remaining comparable to each other and with results slightly in favor of UCB1. To provide more detailed results, we represent in Figures 3, 4 and 5 the runtime comparison per instance with VSIDS, CHB and the VBS respectively for the top three best strategies, i.e. MOSS, UCB1 and RR. These figures confirm the trends that we observed above. More interesting, we can note that, for a noticeable number of instances, MOSS, UCB1 or RR lead to a more efficient solving than the VBS. In Figure 6, we represent the runtime comparison per instance between MOSS, UCB1 and RR. These figures show that MOSS performs better than UCB1 and RR. Surprisingly, MOSS's results are closer to RR than UCB1. However, we will show in Section 5.3.4 that this is consistent with the observed behaviour of MOSS.

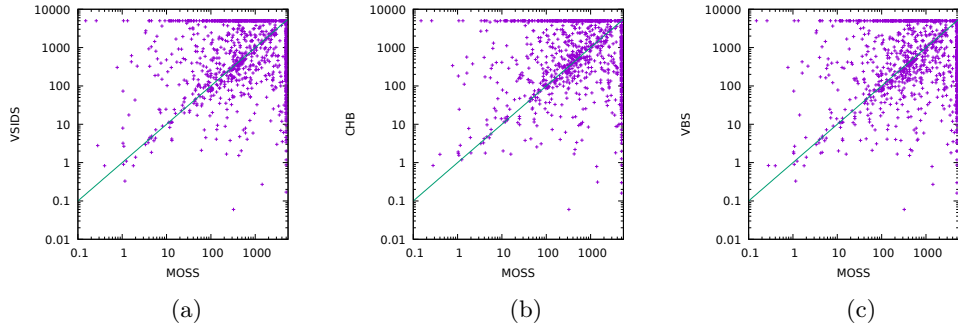


1

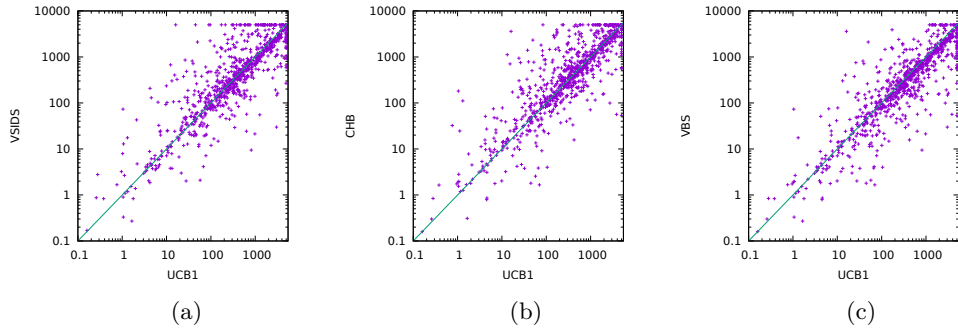
■ **Figure 1** Number of solved instances as a function of CPU time for VSIDS, CHB, static and MAB strategies and the VBS with respect to the whole benchmark.



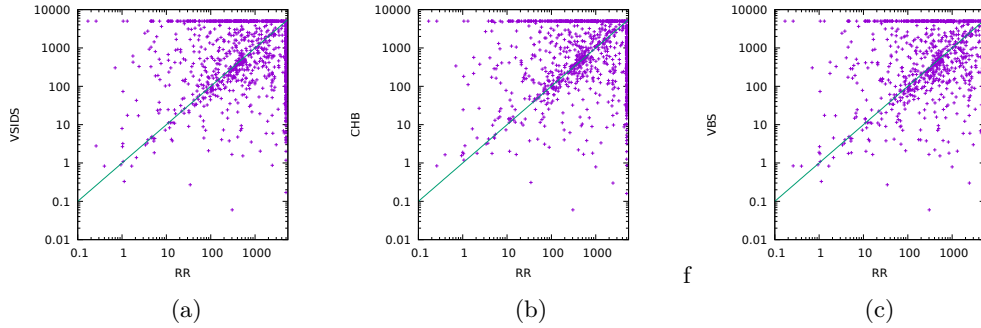
■ **Figure 2** Number of solved satisfiable (a) and unsatisfiable (b) instances as a function of CPU time for VSIDS, CHB, static and MAB strategies and the VBS w.r.t the whole benchmark.



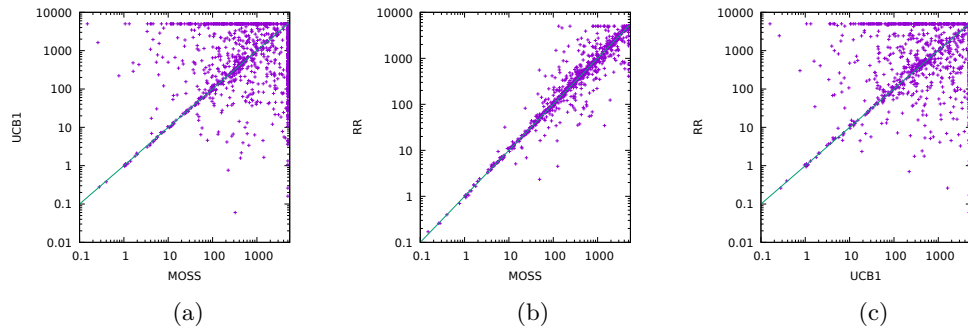
■ **Figure 3** Runtime comparison (in seconds) of MOSS w.r.t. VSIDS (a), CHB (b) and VBS (c) in logarithmic scale.



■ **Figure 4** Runtime comparison (in seconds) of UCB1 w.r.t. VSIDS (a), CHB (b) and VBS (c) in logarithmic scale.



■ **Figure 5** Runtime comparison (in seconds) of RR w.r.t. VSIDS (a), CHB (b) and VBS (c) in logarithmic scale.



■ **Figure 6** Runtime comparison (in seconds) of MOSS w.r.t. UCB1 (a) and RR (b) and of UCB1 w.r.t. RR (c) in logarithmic scale.

Table 2 Comparison between VSIDS, CHB, static and MAB strategies and the VBS (over VSIDS and CHB) in terms of the number of solved instances (#I) and cumulative solving time (for solved instances in seconds) in Kissat for instance families in the benchmark. The results of families marked with † are joint from two different yearly benchmarks. For each row, the best results without considering the VBS are written in bold, breaking ties with milliseconds if necessary.

Family		VSIDS		CHB		SS		RR		UCB1		MOSS		VBS	
#I	name	#I	time	#I	time	#I	time	#I	time	#I	time	#I	time	#I	time
14	Antibandwidth	2	1,010	7	9,804	7	21,381	8	11,469	9	15,651	9	14,370	7	9,628
20	Almost Perfect Non-Linear S-box Finder	11	15,324	7	14,386	11	18,815	10	18,974	11	19,659	11	15,969	12	16,138
38	Arithmetic Verification	13	11,010	14	12,268	8	6,657	9	12,160	13	13,349	9	11,811	14	10,801
13	Baseball-lineup	12	3,317	12	2,949	12	3,192	12	2,645	12	2,947	12	2,540	12	2,906
17	Bitcoin	8	1,972	7	479	8	2,001	8	1,907	8	2,199	8	1,901	8	1,784
14	Coloring	6	7,501	5	7,327	7	12,097	5	3,101	5	2,753	6	7,556	6	5,876
14	Core-based	13	8,438	13	10,860	13	8,737	13	7,431	14	14,165	14	14,861	13	7,239
20	Course Scheduling	14	14,439	14	15,363	13	11,205	14	11,804	15	9,323	15	10,801	14	9,654
13	Cover	4	9	4	10	4	9	4	10	4	10	4	11	4	9
20	Chromatic Number (CNP)	20	1,708	20	1,972	20	1,743	20	1,402	20	1,180	20	1,415	20	1,194
20	Divide and Unique Inverse	16	18,456	16	22,537	16	19,615	16	20,864	16	23,543	16	20,931	16	18,109
7	Discrete-Logarithm	4	3,640	4	7,623	4	3,344	4	4,718	4	4,894	4	5,883	4	3,627
14	Edge-Matching Puzzle †	3	4,476	3	6,201	2	1,430	4	5,546	4	8,674	4	6,615	4	8,823
32	Factoring †	30	17,224	27	12,497	30	16,554	27	10,297	28	20,528	28	20,106	30	12,546
15	Floating-Point Program Verification	12	972	12	874	12	1,024	12	1,050	12	1,076	12	991	12	775
19	Grand Tour Puzzle	9	1,834	9	2,037	9	1,771	9	2,042	9	2,061	9	2,153	9	1,783
20	Hard 3-SAT	18	5,486	19	8,888	18	6,424	19	3,654	18	3,643	19	4,042	19	7,238
13	Hgen	12	3,168	12	2,423	12	3,134	12	1,783	12	783	12	2,590	12	2,365
14	Influence Maximization	12	9,617	12	7,424	12	9,472	12	10,037	12	9,989	12	10,152	12	6,865
14	Kakuro Puzzle	12	15,705	11	13,942	11	10,312	12	16,365	12	16,269	12	16,216	12	14,582

■ **Table 3** Comparison between VSIDS, CHB, static and MAB strategies and the VBS (over VSIDS and CHB) in terms of the number of solved instances (#I) and cumulative solving time (for solved instances in seconds) in Kissat for some instance families in the benchmark (Table 2 continued). The results of families marked with † are joint from two different yearly benchmarks. For each row, the best results without considering the VBS are written in bold, breaking ties with milliseconds if necessary.

Family name	VSIDS		CHB		SS		RR		UCB1		MOSS		VBS	
	#I	time	#I	time	#I	time	#I	time	#I	time	#I	time	#I	time
k-Colorability	15	5 7,923	5	6,528	6	12,338	5	5,167	5	6,998	5	5,540	6	10,660
Keystream Generator Cryptanalysis	18	18 14,494	14	15,104	18	14,731	18	19,433	18	13,118	18	16,828	18	13,240
Lam-Discrete-Geometry	9	7 4,756	7	7,106	7	4,899	7	7,147	7	6,856	7	6,953	7	4,680
Logical Cryptanalysis	20	20 5,606	20	10,476	20	6,748	20	5,518	20	4,241	20	4,208	20	4,946
Polynomial Multiplication †	27	20 28,884	21	27,880	21	33,016	21	27,107	20	23,653	22	30,535	25	41,331
Population Safety	15	13 2,188	14	1,991	13	2,423	12	1,624	13	3,148	13	2,417	14	1,809
Preimage	11	6 11,865	4	7,998	7	13,070	6	16,201	5	9,255	5	5,852	8	15,435
Relativized Pigeonhole Principle (RPHP)	20	11 14,890	10	11,344	11	15,229	10	9,869	11	14,065	11	13,967	11	14,890
Reversing Elementary Cellular Automata	11	11 4,046	11	4,065	11	3,923	11	4,738	11	5,151	11	4,476	11	3,664
Scrambled	20	19 7,022	18	9,278	20	11,441	19	8,114	19	14,519	20	15,141	20	6,089
SHA-1 Pre-image Attack	20	20 14,509	20	23,429	20	14,085	19	21,975	20	25,206	20	20,255	20	12,214
Social Golfer	14	2 3,008	1	3,791	1	410	2	1,202	3	6,046	2	1,530	2	3,008
Software Bounded Model Checking	19	18 7,082	18	8,959	18	7,219	18	7,966	18	8,308	18	8,435	18	6,969
Station Repacking	12	6 15,286	12	10,656	11	29,669	12	13,752	12	8,766	12	6,855	12	10,656
Stedman Triples †	27	10 8,766	11	11,508	11	11,394	12	8,215	12	7,399	13	12,329	11	6,947
SV Competition	18	18 7,522	17	3,350	17	4,227	18	7,251	18	7,935	18	7,829	18	5,577
Timetable †	26	1 1565	10	5082	10	28,417	11	5,607	11	6,247	11	5,157	10	5,082
Tree Decomposition	20	11 12,049	10	6,870	10	7,947	11	4,700	10	3,965	11	5,674	11	7,874
Vlsat	14	3 103	7	4,457	4	3,404	7	529	7	500	7	547	7	3,934

5.3.3 Instance Families

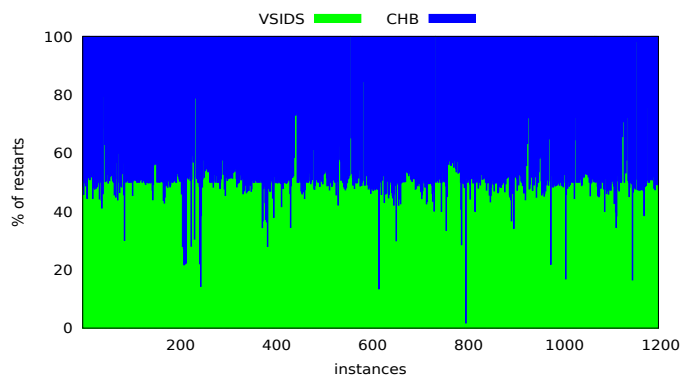
In order to provide a more thorough analysis, we describe in Tables 2 and 3 the results obtained by VSIDS, CHB, static and MAB strategies and the VBS on instance families within the benchmark [23, 22, 7]. The best strategy, i.e. MOSS, manages to rank first in 9 different families over 39 in total (23%), e.g. **Antibandwidth**, **Bitcoin** and **Stedman Triples**. Interestingly, this strategy achieves remarkable results, which are better than those of the VBS over VSIDS and CHB, for certain families such as **Logical cryptanalysis**, **RPHP** and **Station Repacking**. SS also achieves the top performance on several different families such as **Factoring**, **Scrambled** and **SHA-1 Pre-image Attack**. More precisely, SS also manages to rank on top for 9 different families which shows the interest of this strategy even though it ranks last overall compared to RR, UCB1 and MOSS. As for UCB1, it achieves top rank in 6 different families. In particular, its performance on the families **Hgen**, **CNP** and **Keystream Generator Cryptanalysis** is noteworthy since it manages to outperform the VBS. On the other hand, RR ranks top in only 4 instance families but this does not necessarily reflect its overall performance since it falls slightly behind the top ranked heuristic/strategy in other families, yet this is clearly another point in favor of UCB1 as a comparable strategy. Finally, VSIDS and CHB are ranked first in several families which shows that these heuristics remain robust as standalone heuristics.

5.3.4 MAB Behaviour

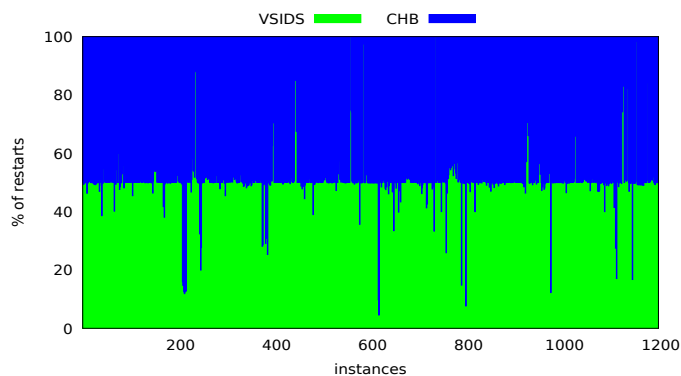
In this section, we focus on the behaviour of MAB strategies and particularly the use of arms. In Figures 7 and 8, we represent the percentage of use, i.e. percentage of restarts where each arm gets chosen respectively by UCB1 and MOSS. We observe that both strategies alternate between the heuristics but the percentages are mainly within the interval $[40\%, 60\%]$ and are often close to 50%. MOSS seems to choose in a more balanced way between VSIDS and CHB in comparison to UCB1 which introduces more variations in its choices. This behaviour is consistent with the observations made in Section 5.3.1 concerning Figure 6. The fact that the percentages are mostly within a tight interval is not surprising considering that the number of stable restarts in Kissat, during which heuristics are used, is usually very low. To give an idea, the average number of stable restarts performed by Kissat for instances solved with MOSS (resp. UCB1) is 765 (resp. 771) while the median value is much lower and amounts to 313 (resp. 338). Therefore, the obtained percentages seem adequate especially taking into account that these strategies need to achieve a good trade-off between exploration and exploitation. Notice the consecutive dents and bumps in Figures 7 and 8 which correspond to an homogeneous behaviour within the same instance family in the benchmark. It is important to note that, although the behaviour of MAB strategies may seem close to RR, this is not exactly the case. Indeed, these strategies rely on the computed reward to choose the most relevant arm during exploitation and especially when there is a large gap between the performance of the heuristics, whereas RR is a static strategy and cannot adapt its choices. This helps to explain the better results of the MAB strategies not only in terms of solved instances but also in terms of solving time and particularly in the case of MOSS. In fact, the remarkable performance of MOSS is also due to the fact that it takes into account the number of arms and has better regret than UCB1.

5.4 On MAB strategies and Branching Heuristics

In this section, we discuss the relevance of choosing Upper Confidence Bound strategies in the Multi-Armed Bandit framework and VSIDS and CHB as candidate heuristics. As mentioned in Section 3.2, many strategies were devised and theoretically studied in the context of



■ **Figure 7** Percentages of use of each arm in UCB1 w.r.t the whole benchmark. The instances are reported consecutively for each yearly benchmark (from 2018 to 2020) and are alphabetically ordered. For unsolved instances, the percentages of use at the timeout are provided.



■ **Figure 8** Percentages of use of each arm in MOSS w.r.t the whole benchmark. The instances are reported consecutively for each yearly benchmark (from 2018 to 2020) and are alphabetically ordered. For unsolved instances, the percentages of use at the timeout are provided.

MAB and can therefore be used in our framework. For instance, we can mention two other well-know strategies for MAB: ϵ -greedy [38] and EXP3 [6]. However, these strategies are not deterministic, i.e. there is a factor of uncertainty or probability. Therefore, unlike UCB strategies, they cannot always guarantee top performance and may produce different results on the same benchmark. Furthermore, UCB strategies were shown relevant and more efficient for similar MAB frameworks in the context of CSP [36, 41, 13]. This remains true in Kissat as we observed, through extensive experimentation, that ϵ -greedy and EXP3 perform poorly compared to UCB strategies and remain comparable to VSIDS and CHB.

In addition, notice that the MAB framework enables the use of several heuristics. In fact, one would argue that adding more heuristics may enable to reach more families and instances through diversification. However, recall that modern SAT solvers, and in particular Kissat, are highly competitive and rely on powerful heuristics to achieve impressive results. A bad heuristic or tuning of the parameters (e.g. the restart policy settings) can greatly deteriorate the performance of a solver. Furthermore, practically all heuristics used in modern SAT solvers are variants of VSIDS, which has been the dominant heuristic since its introduction in 2001 [35]. Only recently CHB has been introduced and shown competitive with VSIDS [29]. CHB has only one variant called LRB [30] but, through extensive experimentation, CHB turned out to be more robust with respect to different solvers and settings. The results

reported in Table 1 also show that CHB can reach new instances (the VBS achieves a gain of more than 5.8% in terms of solved instances) while remaining competitive and comparable overall with respect to VSIDS in the context of a highly competitive solver such as Kissat.

5.4.1 Kissat_MAB at the SAT Competition 2021

We submitted the solver Kissat augmented with a MAB framework relying on the UCB1 strategy to the SAT competition 2021² under the name Kissat_MAB [14]. This solver won the Main Track of the competition and managed to solve 296 instances over 400 with a gap of 8 instances compared to the second ranked solver. Kissat_MAB also placed first in the Main SAT and NoLimits tracks. Compared to default Kissat, which also participated in the competition under the name Kissat_sc2021_default with several new improvements over its last version [11], Kissat_MAB achieves better results with 9 (resp. 11) additional solved (resp. satisfiable) instances. Furthermore, Kissat_MAB remains highly competitive on unsatisfiable instances and comparable to default Kissat as it managed to solve 148, only 2 instances less than Kissat_sc2021_default. Notice that this gap can clearly be narrowed or even turned in favor of Kissat_MAB if the MOSS strategy is used as shown in our experimental evaluation. To summarize, the results of the SAT competition 2021 seem to corroborate our experimental study and to confirm the relevance of combining VSIDS and CHB using restarts in improving the performance of highly competitive SAT solvers.

6 Conclusion and Future Work

In this paper, we evaluated different strategies which take advantage of the restart mechanism to combine two state of the art heuristics, namely VSIDS and CHB. In particular, we introduced a MAB framework for SAT and chose two known Upper Confidence Bound strategies, called UCB1 and MOSS. These strategies rely on a reward function which evaluates the capacity of the heuristics to reach conflicts quickly and efficiently. Our experimental evaluation shows that VSIDS and CHB are compatible since their combination through different strategies taking advantage of the restart mechanism is able to substantially increase the performance of the competitive solver Kissat. In particular, the MOSS strategy outperforms not only VSIDS and CHB but also all the other strategies. The strategies UCB1 and RR have also shown competitive results. These three strategies achieve substantial gain in terms of solved instances, mainly satisfiable ones, and in terms of solving time. Moreover, these strategies achieve results which are very close to the VBS over VSIDS and CHB. Our solver Kissat_MAB won the Main track of the SAT competition 2021 and placed first in the Main SAT and NoLimits tracks thus showing the relevance of combining VSIDS and CHB using restarts and its ability to improve the performance of highly competitive SAT solvers.

As future work, it would be interesting to refine the reward function used in MAB strategies by relying on a combination of different criteria [15] so as to improve the MAB framework especially with respect to unsatisfiable instances. It would also be interesting to focus on one heuristic and try to refine it using a similar MAB framework, an approach which was shown relevant in the context of the Constraint Satisfaction Problem (CSP) [13]. Finally, it would be interesting to use these strategies to improve other components in modern SAT solvers such as clause deletion [2].

² Results and source code available on <https://satcompetition.github.io/2021/>.

References

- 1 Rajeev Agrawal. Sample mean based index policies by $O(\log n)$ regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27(4):1054–1078, 1995. doi:10.2307/1427934.
- 2 Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In Craig Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI'09*, page 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc. URL: <https://www.ijcai.org/Proceedings/09/Papers/074.pdf>.
- 3 Gilles Audemard and Laurent Simon. Refining Restarts Strategies for SAT and UNSAT. 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 118–126. Springer, 2012. doi:10.1007/978-3-642-33558-7_11.
- 4 Jean-Yves Audibert and Sébastien Bubeck. Minimax Policies for Adversarial and Stochastic Bandits. In *COLT 2009 - The 22nd Conference on Learning Theory, Montreal, Quebec, Canada, June 18-21, 2009*, 2009. URL: <http://www.cs.mcgill.ca/%7Ecolt2009/papers/022.pdf>.
- 5 Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.*, 47(2-3):235–256, 2002. doi:10.1023/A:1013689704352.
- 6 Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. The nonstochastic multiarmed bandit problem. *SIAM journal on computing*, 32(1):48–77, 2002. doi:10.1137/S0097539701398375.
- 7 Tomáš Balyo, Nils Froleys, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Series of Publications B*. Department of Computer Science, University of Helsinki, 2020. URL: https://helda.helsinki.fi/bitstream/handle/10138/318450/sc2020_proceedings.pdf.
- 8 Armin Biere. Adaptive Restart Strategies for Conflict Driven SAT Solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer, 2008. doi:10.1007/978-3-540-79719-7_4.
- 9 Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YaSAT Entering the SAT Competition 2017. In Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors, *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.
- 10 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleys, 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.
- 11 Armin Biere, Mathias Fleury, and Maximillian Heisinger. CADICAL, KISSAT, PARACOOBA Entering the SAT Competition 2021. In *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions*, volume B-2021-1 of *Department of Computer Science Series of Publications B*, pages 10–13. University of Helsinki, 2021 .
- 12 Armin Biere and Andreas Fröhlich. Evaluating CDCL Variable Scoring Schemes. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 405–422. Springer, 2015. doi:10.1007/978-3-319-24318-4_29.
- 13 Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux. On the Refinement of Conflict History Search Through Multi-Armed Bandit. In *32nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2020, Baltimore, MD, USA, November 9-11, 2020*, pages 264–271. IEEE, 2020. doi:10.1109/ICTAI50040.2020.00050.

- 14 Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux. Kissat_MAB: Combining VSIDS and CHB through Multi-Armed Bandit. In *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions*, volume B-2021-1 of *Department of Computer Science Series of Publications B*, pages 15–16. University of Helsinki, 2021.
- 15 Wei Chu, Lihong Li, Lev Reyzin, and Robert E. Schapire. Contextual Bandits with Linear Payoff Functions. In Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, volume 15 of *JMLR Proceedings*, pages 208–214. JMLR.org, 2011. URL: <http://proceedings.mlr.press/v15/chu11a/chu11a.pdf>.
- 16 Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71*, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery. doi:10.1145/800157.805047.
- 17 Todd Deshane, Wenjin Hu, Patty Jablonski, Hai Lin, Christopher Lynch, and Ralph Eric McGregor. Encoding First Order Proofs in SAT. In Frank Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 476–491. Springer, 2007. doi:10.1007/978-3-540-73595-3_35.
- 18 Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. doi:10.1007/978-3-540-24605-3_37.
- 19 Carla P Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of automated reasoning*, 24(1-2):67–100, 2000. doi:10.1023/A:1006314320276.
- 20 Shai Haim and Marijn Heule. Towards Ultra Rapid Restarts. *CoRR*, abs/1402.4413, 2014. arXiv:1402.4413.
- 21 William D. Harvey and Matthew L. Ginsberg. Limited Discrepancy Search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 607–615. Morgan Kaufmann, 1995. URL: <http://ijcai.org/Proceedings/95-1/Papers/080.pdf>.
- 22 Marijn Heule, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Race 2019: Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications B*. Department of Computer Science, University of Helsinki, 2019. URL: https://helda.helsinki.fi/bitstream/handle/10138/306988/sr2019_proceedings.pdf.
- 23 Marijn Heule, Matti Juhani Järvisalo, Martin Suda, et al., editors. *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*, volume B-2018-1 of *Department of Computer Science Series of Publications B*. Department of Computer Science, University of Helsinki, 2018. URL: https://helda.helsinki.fi/bitstream/handle/10138/237063/sc2018_proceedings.pdf.
- 24 Ted Hong, Yanjing Li, Sung-Boem Park, Diana Mui, David Lin, Ziyad Abdel Kaleq, Nagib Hakim, Helia Naeimi, Donald S. Gardner, and Subhasish Mitra. QED: Quick Error Detection tests for effective post-silicon validation. In Ron Press and Erik H. Volkerink, editors, *2011 IEEE International Test Conference, ITC 2010, Austin, TX, USA, November 2-4, 2010*, pages 154–163. IEEE Computer Society, 2010. doi:10.1109/TEST.2010.5699215.
- 25 Vitaly Kurin, Saad Godil, Shimon Whiteson, and Bryan Catanzaro. Improving SAT Solver Heuristics with Graph Networks and Reinforcement Learning. *CoRR*, abs/1909.11830, 2019. arXiv:1909.11830.
- 26 Tze Leung Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985. doi:10.1016/0196-8858(85)90002-8.

- 27 Nadjib Lazaar, Youssef Hamadi, Said Jabbour, and Michèle Sebag. Cooperation control in Parallel SAT Solving: a Multi-armed Bandit Approach. Research Report RR-8070, INRIA, 2012.
- 28 Jia Hui Liang, Chanseok, Vijay Ganesh, Krzysztof Czarnecki, and Pascal Poupart. MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB. In *Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions*, page 20–21, 2017.
- 29 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Exponential Recency Weighted Average Branching Heuristic for SAT Solvers. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 3434–3440. AAAI Press, 2016. URL: <http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12451>.
- 30 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning Rate Based Branching Heuristic for SAT Solvers. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2016. doi:10.1007/978-3-319-40970-2_9.
- 31 Jia Hui Liang, Chanseok Oh, Vijay Ganesh, Krzysztof Czarnecki, and Pascal Poupart. MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB. In *Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions*, page 52–53, 2016.
- 32 Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, pages 128–133, 1993. doi:10.1109/ISTCS.1993.253477.
- 33 Inês Lynce and João Marques-Silva. SAT in Bioinformatics: Making the Case with Haplotype Inference. 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 136–141. Springer, 2006. doi:10.1007/11814948_16.
- 34 J.P. Marques-Silva and K.A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999. doi:10.1109/12.769433.
- 35 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001. doi:10.1145/378239.379017.
- 36 Anastasia Paparrizou and Hugues Watez. Perturbing Branching Heuristics in Constraint 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 496–513. Springer, 2020. doi:10.1007/978-3-030-58475-7_29.
- 37 Lawrence Ryan. Efficient algorithms for clause-learning SAT solvers. Master’s thesis, Simon Fraser University, 2004.
- 38 Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, Cambridge, MA, USA, 1998. URL: <https://www.worldcat.org/oclc/37293240>.
- 39 William R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3-4):285–294, December 1933. doi:10.1093/biomet/25.3-4.285.
- 40 Toby Walsh. Search in a Small World. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2, IJCAI’99*, page 1172–1177, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. URL: <https://www.ijcai.org/Proceedings/99-2/Papers/071.pdf>.

- 41 Hugues Watez, Frédéric Koriche, Christophe Lecoutre, Anastasia Paparrizou, and Sébastien Tabary. Learning Variable Ordering Heuristics with Multi-Armed Bandits and Restarts. 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*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 371–378. IOS Press, 2020. doi:10.3233/FAIA200115.
- 42 Wei Xia and Roland H. C. Yap. Learning Robust Search Strategies Using a Bandit-Based Approach. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 6657–6665. AAAI Press, 2018. URL: <https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17192>.

On the Tractability of Explaining Decisions of Classifiers

Martin C. Cooper 

IRIT, Université de Toulouse III, France

João Marques-Silva 

IRIT, CNRS, Toulouse, France

Abstract

Explaining decisions is at the heart of explainable AI. We investigate the computational complexity of providing a formally-correct and minimal explanation of a decision taken by a classifier. In the case of threshold (i.e. score-based) classifiers, we show that a complexity dichotomy follows from the complexity dichotomy for languages of cost functions. In particular, submodular classifiers allow tractable explanation of positive decisions, but not negative decisions (assuming $P \neq NP$). This is an example of the possible asymmetry between the complexity of explaining positive and negative decisions of a particular classifier. Nevertheless, there are large families of classifiers for which explaining both positive and negative decisions is tractable, such as monotone or linear classifiers. We extend tractable cases to constrained classifiers (when there are constraints on the possible input vectors) and to the search for contrastive rather than abductive explanations. Indeed, we show that tractable classes coincide for abductive and contrastive explanations in the constrained or unconstrained settings.

2012 ACM Subject Classification Theory of computation \rightarrow Machine learning theory; Theory of computation \rightarrow Problems, reductions and completeness

Keywords and phrases machine learning, tractability, explanations, weighted constraint satisfaction

Digital Object Identifier 10.4230/LIPIcs.CP.2021.21

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.

1 Explanations of ML models

Recent work has shown that it is possible to apply formal reasoning to explainable AI, thus providing formal guarantees of correctness of explanations [39, 40, 23, 24, 14, 13, 20]¹. However, scalability quickly becomes an issue because testing the validity of an explanation may be NP-hard, or even $\#P$ -hard. As a result, more recent work focused on investigating classes of classifiers for which explanations can be found in polynomial time [2, 33, 1]. A natural question is thus which other classes of classifiers allow for formal explanations to be computed in polynomial time. This is our motivation for investigating the computational complexity of finding explanations of decisions taken by boolean classifiers. More concretely, the paper proposes conditions on the decision problems associated with classification functions, which enable finding in polynomial time a so-called abductive or contrastive explanation. Furthermore, the paper shows that several large classes of classifiers respect the proposed conditions.

We consider a boolean classification problem with two classes $\mathcal{K} = \{\oplus, \ominus\}$, defined on a set of features (or attributes) x_1, \dots, x_n , which will be represented by their indices $\mathcal{A} = \{1, \dots, n\}$. The features can either be real-valued or categorical. For real-valued features,

¹ There exist a wide range of explainable AI approaches offering no formal guarantees of correctness, e.g. [17].



we have $\lambda_i \leq x_i \leq \mu_i$, where λ_i, μ_i are given lower and upper bounds. For categorical features, we have $x_i \in \{1, \dots, d_i\}$. A concrete assignment to the features referenced by \mathcal{A} is represented by an n -dimensional vector $\mathbf{a} = (a_1, \dots, a_n)$, where a_j denotes the value assigned to feature j , represented by variable x_j , such that a_j is taken from the domain of x_j . The set of all n -dimensional vectors denotes the *feature space* \mathbb{A} .

Given a classifier with features \mathcal{A} , the corresponding *decision function* is a mapping from the feature space to the set of classes, i.e. $\tau : \mathbb{A} \rightarrow \mathcal{K}$. For example, for a linear classifier, the decision function picks \oplus if $\sum_i w_i x_i > t$, and \ominus if $\sum_i w_i x_i \leq t$, for some constants w_i ($i = 1, \dots, n$) and t . Given $\mathbf{a} \in \mathbb{A}$, with $\tau(\mathbf{a}) = c$, we consider the set of feature literals of the form $(x_i = a_i)$, where x_i denotes a variable and a_i a constant.

► **Definition 1.** A *PI-explanation* [39] is a subset-minimal set $\mathcal{P} \subseteq \mathcal{A}$, denoting feature literals, i.e. feature-value pairs (taken from \mathbf{a}), such that

$$\forall (\mathbf{x} \in \mathbb{A}). \bigwedge_{j \in \mathcal{P}} (x_j = a_j) \rightarrow \tau(\mathbf{x}) = c \quad (1)$$

is true.

PI-explanations are also referred to as abductive explanations [23]. PI-explanations are analogous to prime implicants of propositional formulae: finding subset-minimal (prime) implicants rather than shortest implicants is interesting from a computational point of view since deciding the existence of an implicant of size less than k is Σ_2^P -complete [43].

► **Example 2.** We consider as a running example the case of a bank which uses a function τ to decide whether to grant a loan to a couple represented by a feature vector $\mathbf{x} = (sal_1, sal_2, age_1, age_2)$, where sal_1, sal_2 are the salaries and age_1, age_2 the ages of the two people making up the couple. Suppose that $\tau(\mathbf{x}) = \oplus$ if and only if $(\max(sal_1, sal_2) \geq sal_{\min}) \wedge (\min(age_1, age_2) \leq age_{\max})$. If \mathbf{a} corresponds to a couple who both earn more than sal_{\min} and both are younger than age_{\max} , then there are four PI-explanations for $\tau(\mathbf{a}) = \oplus$: $\{1, 3\}$, $\{1, 4\}$, $\{2, 3\}$ and $\{2, 4\}$. For example, $\{1, 3\}$ means that the first and third features (sal_1 and age_1) are sufficient to explain the decision. On the other hand, if \mathbf{b} corresponds to a couple who both earn more than sal_{\min} and both are older than age_{\max} , then the only PI-explanation for $\tau(\mathbf{b}) = \ominus$ is $\{3, 4\}$ (i.e. that they are both too old).

2 Definitions

In order to study the complexity of finding explanations, and in particular to identify tractable cases, we need to place restrictions on the classifier τ . Let \mathcal{D} be a set of domains. For example, \mathcal{D} may include all intervals of the real numbers and all finite subsets of the integers. Let $\mathcal{T}^{\mathcal{D}}$ represent the family of functions $\tau : \prod_{i=1}^n D_i \rightarrow \mathcal{K}$ where each domain $D_i \in \mathcal{D}$ (i.e. the feature space \mathbb{A} is the Cartesian product of domains from \mathcal{D}). We call n the arity of τ . Recall that $\mathcal{K} = \{\ominus, \oplus\}$.

We say that $\tau : \mathbb{A} \rightarrow \mathcal{K}$ is a *\mathcal{F} -threshold classifier* if it can be represented by an objective function $f : \mathbb{A} \rightarrow \mathbb{R} \cup \{-\infty, \infty\}$ belonging to \mathcal{F} such that an input vector $\mathbf{x} \in \mathbb{A}$ is classified as positive ($\tau(\mathbf{x}) = \oplus$) iff $f(\mathbf{x})$ is strictly greater than some threshold t , negative otherwise. Concentrating on threshold classifiers is not really a restriction, since any binary classifier $\tau : \mathbb{A} \rightarrow \{0, 1\}$ (identifying \ominus with 0 and \oplus with 1) can be viewed as a threshold classifier with $f = \tau$ and threshold $t = 0$. It is the choice of the family of functions \mathcal{F} which determines the complexity of explaining decisions.

If \mathcal{F} is the set of real-valued linear functions, then \mathcal{F} -threshold classifiers are known as linear classifiers. Similarly, we can define larger families of threshold classifiers, such as monotone or submodular threshold-classifiers by restricting the objective function f to be monotone or submodular. A function f is *monotone* if $\forall \mathbf{x}, \mathbf{y}, \mathbf{x} \leq \mathbf{y}$ implies $f(\mathbf{x}) \leq f(\mathbf{y})$; f is *submodular* if $\forall \mathbf{x}, \mathbf{y}, f(\min(\mathbf{x}, \mathbf{y})) + f(\max(\mathbf{x}, \mathbf{y})) \leq f(\mathbf{x}) + f(\mathbf{y})$, where \min and \max are applied componentwise [16]. All linear functions are submodular but only those linear functions whose coefficients are non-negative are monotone. Similarly, f is *antitone* if $\forall \mathbf{x}, \mathbf{y}, \mathbf{x} \leq \mathbf{y}$ implies $f(\mathbf{x}) \geq f(\mathbf{y})$; f is *supermodular* if $\forall \mathbf{x}, \mathbf{y}, f(\min(\mathbf{x}, \mathbf{y})) + f(\max(\mathbf{x}, \mathbf{y})) \geq f(\mathbf{x}) + f(\mathbf{y})$; f is *modular* if $\forall \mathbf{x}, \mathbf{y}, f(\min(\mathbf{x}, \mathbf{y})) + f(\max(\mathbf{x}, \mathbf{y})) = f(\mathbf{x}) + f(\mathbf{y})$. It is worth pointing out that all these classes of functions (linear, modular, submodular, supermodular, monotone, antitone) are closed under addition. Modular functions are exactly those functions f that can be decomposed into a sum of unary functions $f(\mathbf{x}) = \sum_{i=1}^n f_i(x_i)$ [9]. By definition, modular functions are both submodular and supermodular and include linear functions as a special case.

Monotonicity [34] is a desirable property in applications where it is important to guarantee meritocratic fairness (do not favour a less-qualified candidate) [27]. It has been imposed even for classifiers as complex as neural networks [32].

Submodularity is a well-studied concept in Operations Research and Machine learning whose origins can be traced back to the the notion of diminishing marginal returns studied by Gaspard Monge [5]. It is well known that a submodular function over boolean domains can be minimized in polynomial time [36, 31, 6]. For example, if the objective function f is the sum of functions of pairs of variables, then minimizing f is equivalent to finding the minimum cut in a weighted graph [8]. A polynomial-time algorithm for minimizing a submodular function over any finite domains follows from the polynomial reduction to boolean domains obtained by replacing each variable x_i with domain $\{1, \dots, d\}$ by $d - 1$ boolean variables $x_{ir} = 1 \Leftrightarrow x_i \geq r$ ($r = 1, \dots, d - 1$) [9].

► **Example 3.** Consider again our example of a bank which uses a function τ to decide whether to grant a loan to a couple represented by the feature vector $\mathbf{x} = (sal_1, sal_2, age_1, age_2)$. Suppose that τ is a threshold classifier $\tau(\mathbf{x}) = \oplus \Leftrightarrow f(\mathbf{x}) > t$, where $f = \alpha f_1 + \beta f_2 + \gamma f_3$ and $f_1(\mathbf{x}) = \max(sal_1, sal_2) + \mu \min(sal_1, sal_2)$ (where $0 \leq \mu \leq 1$), and $f_2(\mathbf{x}) = 1$ iff $(\max(age_1, age_2) \geq age_{\min})$ (and $f_2(\mathbf{x}) = 0$ otherwise), and $f_3(\mathbf{x}) = 1$ iff $(\min(age_1, age_2) \leq age_{\max})$ (and $f_3(\mathbf{x}) = 0$ otherwise), where age_{\min}, age_{\max} and $\alpha, \beta, \gamma, \mu \geq 0$ are constants.

It can be verified that f_1 and f_2 are both submodular and monotone, and that f_3 is both submodular and antitone. Thus (by additivity of submodularity), f is submodular but it is neither monotone nor antitone (assuming $\alpha, \beta, \gamma > 0$). On the other hand, f is monotone if $\gamma = 0$.

We say that τ is a \mathcal{F} -multi-threshold classifier if it can be represented by functions $f_i \in \mathcal{F}$ ($i = 1, \dots, r$) such that an input vector $\mathbf{x} \in \mathbb{A}$ is classified as positive ($\tau(\mathbf{x}) = \oplus$) iff $(f_1(\mathbf{x}) > t_1) \wedge \dots \wedge (f_r(\mathbf{x}) > t_r)$ for some constants t_i ($i = 1, \dots, r$). For example, if \mathcal{F} is the set of real-valued linear functions, then for \mathcal{F} -multi-threshold classifiers the set of positive examples \mathbf{x} is a polytope.

We are specifically interested in families of classifiers $\mathcal{T} \subseteq \mathcal{T}^{\mathcal{D}}$ which are closed under replacing arguments by constants (sometimes known as restriction or conditioning [15]) since this is a necessary condition for the correctness of our polynomial-time algorithm. Fortunately, this is true for most families of functions of interest. For example, a linear/monotone/submodular threshold-classifier remains respectively linear/monotone/submodular if any of its arguments are replaced by constants. For $\tau \in \mathcal{T}^{\mathcal{D}}$ of arity n , $S \subseteq \{1, \dots, n\}$ and \mathbf{v} an assignment to the arguments indexed by S , let $\tau_{\mathbf{v}} : \Pi_{i \notin S} D_i \rightarrow \mathcal{K}$ be the function obtained

from τ by fixing the arguments in S to \mathbf{v} , i.e. for all $\mathbf{x} \in \Pi_{i \notin S} D_i$, $\tau_{\mathbf{v}}(\mathbf{x}) = \tau(\mathbf{v} \cup \mathbf{x})$. We say that \mathcal{T} is *closed under fixing arguments* if for all $\tau : \Pi_{i=1}^n D_i \rightarrow \mathcal{K}$ such that $\tau \in \mathcal{T}$, for all $S \subseteq \{1, \dots, n\}$ and for all $\mathbf{v} \in \Pi_{i \in S} D_i$, we have $\tau_{\mathbf{v}} \in \mathcal{T}$.

3 Tractability of finding one PI-explanation

To obtain a polynomial-time algorithm, we require that a particular decision problem be solvable in polynomial time. For a family $\mathcal{T} \subseteq \mathcal{T}^{\mathcal{D}}$ of boolean-valued functions, let $\text{TAUTOLOGY}(\mathcal{T})$ be the following decision problem: given a function $\tau \in \mathcal{T}$, is it true that $\tau \equiv \oplus$, i.e. for all $\mathbf{x} \in \mathbb{A}$, $\tau(\mathbf{x}) = \oplus$? To avoid exploring dead-end branches, our algorithm requires the answer to this question for functions obtained by fixing a subset of the arguments of a classifier, which is why we require that \mathcal{T} be closed under fixing arguments.

Firstly we consider the more general case in which the only assumption we make is that all functions in \mathcal{T} execute in polynomial time. In this case, $\text{TAUTOLOGY}(\mathcal{T}) \in \text{coNP}$ (since a counter-example can be verified in polynomial time). If, furthermore, \mathcal{T} is closed under fixing arguments, then using a greedy algorithm (as in Proposition 3.1 case (3) of [7]) we can deduce that n calls to an NP oracle are sufficient to find a PI-explanation. In the following, we investigate cases for which $\text{TAUTOLOGY}(\mathcal{T}) \in \text{P}$ and hence for which finding a PI-explanation is also polynomial-time by a similar greedy algorithm.

We now state conditions which guarantee a polynomial-time algorithm to find one PI-explanation for large classes of classifiers. The algorithm initialises \mathcal{P} to \mathcal{A} and greedily deletes literals from \mathcal{P} as long as this preserves property (1) of being an explanation.

► **Proposition 4.** *If \mathcal{T} is closed under fixing arguments and $\text{TAUTOLOGY}(\mathcal{T}) \in \text{P}$, then for any classifier $\tau \in \mathcal{T}$ and any positively-classified input \mathbf{a} , a PI-explanation of $\tau(\mathbf{a}) = \oplus$ can be found in polynomial time.*

Proof. An explanation is a set $\mathcal{P} \subseteq \{1, \dots, n\}$ such that equation (1) holds. The algorithm is a simple greedy algorithm that initialises \mathcal{P} to the trivial explanation $\{1, \dots, n\}$ (corresponding to the complete assignment \mathbf{a}) and for each $i \in \mathcal{P}$ tests whether i can be deleted to leave a valid explanation $\mathcal{P} \setminus \{i\}$:

```

 $\mathcal{P} \leftarrow \{1, \dots, n\}$ 
for  $i = 1, \dots, n$  :
    if  $\mathcal{P} \setminus \{i\}$  is a valid explanation then  $\mathcal{P} \leftarrow \mathcal{P} \setminus \{i\}$ 
    
```

Clearly, the final value $\tilde{\mathcal{P}}$ of \mathcal{P} is an explanation. Furthermore, it is minimal because if $\mathcal{P} \setminus \{i\}$ was not a valid explanation for some $\mathcal{P} \supseteq \tilde{\mathcal{P}}$, then neither is $\tilde{\mathcal{P}} \setminus \{i\}$.

Let \mathbf{v} be the partial assignment corresponding to the values a_j for $j \in \mathcal{P} \setminus \{i\}$. Testing whether $\mathcal{P} \setminus \{i\}$ is a valid explanation is equivalent to testing whether $\tau_{\mathbf{v}} \equiv \oplus$ and hence can be performed in polynomial time since \mathcal{T} is closed under fixing arguments and $\text{TAUTOLOGY}(\mathcal{T}) \in \text{P}$. The algorithm needs to solve exactly n instances of $\text{TAUTOLOGY}(\mathcal{T})$. It follows that one PI-explanation can be found in polynomial time. ◀

Proposition 4 can be seen as a special case of the complexity of finding maximal solutions to problems for which the instance-solution relation is in P (Proposition 3.1 of [7]).

As we will now see, Proposition 4 applies to a large range of classifiers, such as linear, submodular or monotone threshold-classifiers as well as multi-threshold classifiers.

Consider threshold classifiers of the form $\tau(\mathbf{x}) = \oplus$ iff $f(\mathbf{x}) > t$, for some real-valued objective function $f \in \mathcal{F}$ and some constant t . Then

$$\tau \equiv \oplus \quad \Leftrightarrow \quad \min_{\mathbf{x} \in \mathbb{A}} f(\mathbf{x}) > t. \quad (2)$$

Thus, if \mathcal{T} is the set of \mathcal{F} -threshold classifiers, then $\text{TAUTOLOGY}(\mathcal{T}) \in \text{P}$ if functions in \mathcal{F} can be minimised in polynomial time. Examples of classes of functions that can be minimised in polynomial time are the objective functions of extended linear classifiers (referred to as XLCs) [33], monotone functions over real/integer intervals [34] and submodular functions over finite ordered domains [31, 9].

Now consider the case of multi-threshold classifiers of the form $\tau(\mathbf{x}) = \oplus$ iff $\bigwedge_{i=1}^r f_i(\mathbf{x}) > t_i$, for some real-valued functions $f_i \in \mathcal{F}$ and some constants t_i ($i = 1, \dots, r$). Then

$$\tau \equiv \oplus \iff \bigwedge_{i=1}^r (\min_{\mathbf{x} \in \mathbb{A}} f_i(\mathbf{x}) > t_i). \quad (3)$$

Thus, if \mathcal{T} is the set of \mathcal{F} -multi-threshold classifiers, then again we have that $\text{TAUTOLOGY}(\mathcal{T}) \in \text{P}$ if each function in \mathcal{F} can be minimised in polynomial time. For example, f_1 could be monotone, f_2 submodular and the other f_i linear.

We end this section by showing that a polytime tautology test is not only a sufficient but also a necessary condition for tractability of finding a PI-explanation. Let $\text{PIEXPL}^+(\mathcal{T})$ be the problem of finding a PI-explanation of a positive decision taken by a classifier in \mathcal{T} .

► **Theorem 5.** *If \mathcal{T} is closed under fixing arguments, then $\text{PIEXPL}^+(\mathcal{T}) \in \text{FP}$ iff $\text{TAUTOLOGY}(\mathcal{T}) \in \text{P}$.*

Proof. The “if” part of the proof is Proposition 4. For the “only if” part, suppose that \mathcal{T} is closed under fixing arguments and $\text{PIEXPL}^+(\mathcal{T}) \in \text{FP}$. Let $\tau \in \mathcal{T}$. Let \mathbf{a} be an arbitrary choice of feature vector. Then τ is a tautology iff both $\tau(\mathbf{a}) = \oplus$ and the empty set is a PI-explanation of $\tau(\mathbf{a}) = \oplus$. Note that in the case that the empty set is a PI-explanation, it is necessarily the unique PI-explanation. Thus we can decide $\text{TAUTOLOGY}(\mathcal{T})$ in polynomial time. ◀

4 Explanations of negative decisions

In the previous section we exclusively studied the problem of finding an explanation of a *positive* decision $\tau(\mathbf{x}) = \oplus$. We show in this section that the complexity of this problem can change drastically if we require an explanation of a negative decision $\tau(\mathbf{x}) = \ominus$. For a family $\mathcal{T} \subseteq \mathcal{T}^{\mathcal{D}}$ of boolean functions, let $\text{UNSAT}(\mathcal{T})$ be the following decision problem: given a boolean function $\tau \in \mathcal{T}$, is it true that $\tau \equiv \ominus$, i.e. for all $\mathbf{x} \in \mathbb{A}$, $\tau(\mathbf{x}) = \ominus$? By an entirely similar proof based on a greedy algorithm, we can deduce the following proposition which mirrors Proposition 4.

► **Proposition 6.** *If \mathcal{T} is closed under fixing arguments and $\text{UNSAT}(\mathcal{T}) \in \text{P}$, then for any classifier $\tau \in \mathcal{T}$ and any negatively-classified input \mathbf{a} , a PI-explanation of $\tau(\mathbf{a}) = \ominus$ can be found in polynomial time.*

A simple case in which all features are boolean is \mathcal{T}_{DNF} , the family of DNF classifiers. Since deciding the (un)satisfiability of a DNF is trivial, we have $\text{UNSAT}(\mathcal{T}_{\text{DNF}}) \in \text{P}$ and so a PI-explanation of a negative decision can be found in polynomial time. On the other hand, by Theorem 5, and the co-NP-completeness of deciding whether a DNF is a tautology, a PI-explanation of a positive decision cannot be found in polynomial time (assuming $\text{P} \neq \text{NP}$).

Now consider threshold classifiers of the form $\tau(\mathbf{x}) = \oplus$ iff $f(\mathbf{x}) > t$, for some real-valued objective function $f \in \mathcal{F}$ and some constant t . Then

$$\tau \equiv \ominus \iff \max_{\mathbf{x} \in \mathbb{A}} f(\mathbf{x}) \leq t. \quad (4)$$

Thus, if \mathcal{T} is the set of \mathcal{F} -threshold classifiers, then $\text{UNSAT}(\mathcal{T}) \in \text{P}$ if functions in \mathcal{F} can be *maximised* in polynomial time. Examples of functions that can be maximised in polynomial time are linear, monotone, antitone (over real/integer intervals) or supermodular functions (over finite ordered domains). Note that submodular function maximisation cannot be achieved in polynomial time (assuming $\text{P} \neq \text{NP}$) [12].

Thus, for a given family of classifiers (such as submodular threshold classifiers), the complexity of finding an explanation of a positive decision may be polynomial-time whereas the complexity of finding an explanation of a negative decision may be intractable.

We end this section with a theorem that is the equivalent of Theorem 5 for negative decisions. Let $\text{PIEXPL}^-(\mathcal{T})$ be the problem of finding a PI-explanation of a negative decision taken by a classifier in \mathcal{T} .

► **Theorem 7.** *If \mathcal{T} is closed under fixing arguments, then $\text{PIEXPL}^-(\mathcal{T}) \in \text{FP}$ iff $\text{UNSAT}(\mathcal{T}) \in \text{P}$.*

Proof. The “if” part of the proof is Proposition 6. For the “only if” part, suppose that \mathcal{T} is closed under fixing arguments and $\text{PIEXPL}^-(\mathcal{T}) \in \text{FP}$. Let $\tau \in \mathcal{T}$. Let \mathbf{a} be an arbitrary choice of feature vector. Then τ is unsatisfiable iff both $\tau(\mathbf{a}) = \ominus$ and the empty set is a PI-explanation of $\tau(\mathbf{a}) = \ominus$. Thus we can decide $\text{UNSAT}(\mathcal{T})$ in polynomial time. ◀

5 Explanation of classifiers with constrained features

It may be that some constraints exist between features, so that not all vectors in \mathbb{A} are possible. For example, *gender = male* and *pregnant = yes* are incompatible, and clearly we must have *years_of_employment ≤ age*. This affects the definition of a PI-explanation. Suppose that there are constraints on the possible feature vectors \mathbf{x} given by a predicate $C(\mathbf{x})$. In the context of constraints C , a PI-explanation of a decision $\tau(\mathbf{a}) = c$ is now a subset-minimal set $\mathcal{P} \subseteq \mathcal{A}$ of feature literals such that

$$\forall (\mathbf{x} \in \mathbb{A}). \left(C(\mathbf{x}) \wedge \bigwedge_{j \in \mathcal{P}} (x_j = a_j) \right) \rightarrow \tau(\mathbf{x}) = c. \quad (5)$$

► **Example 8.** Consider a medicine that doctors are allowed to prescribe to everybody who has the flu except to pregnant women. A PI-explanation why Alice (who is pregnant) was not prescribed the medicine is that she is pregnant; there is no need to mention that she is a woman given the constraint that there are no pregnant men. There are two PI-explanations why Bob was prescribed the medicine: (1) that he is not pregnant and he had the flu, (2) that he is a man and he had the flu. Note that the rule for prescribing the medicine can be stated without mentioning gender: prescribe to people who have the flu but are not pregnant. The PI-explanations remain the same. In particular, the explanation (2) for Bob being prescribed the medicine mentions gender even though this feature is not mentioned in the rule. If we did not take into account the constraint that men cannot be pregnant, then the explanation (2) would not be valid.

Equating \ominus with 0 and \oplus with 1, we have the following equivalence which follows from equations (1), (5) and the logical equivalence $C \wedge A \rightarrow B \equiv A \rightarrow B \vee \neg C$

► **Proposition 9.** *A PI-explanation of a classifier τ under constraints C is precisely a PI-explanation of the unconstrained classifier $\tau \vee \neg C$.*

■ **Table 1** Examples of tractable families of constrained threshold-classifiers over finite domains.

decision	objective function f	constraints \mathcal{C}
positive	submodular	max and min-closed
positive	monotone	min-closed
positive	antitone	max-closed
negative	supermodular	max and min-closed
negative	monotone	max-closed
negative	antitone	min-closed

Consider a threshold classifier with objective function f under constraints C . We can reduce to the unconstrained case by introducing the function g where

$$g(\mathbf{x}) = \begin{cases} 0 & \text{if } C(\mathbf{x}) \\ \infty & \text{if } \neg C(\mathbf{x}). \end{cases} \quad (6)$$

Then a PI-explanation for $f(\mathbf{a}) > t$ under constraints C is a PI-explanation of $f(\mathbf{a}) + g(\mathbf{a}) > t$ (in the unconstrained setting). We saw in Section 3 that finding a PI-explanation of a positive decision taken by a threshold classifier is polynomial-time if the objective function can be minimised in polynomial time. Thus, for example, if $f + g$ is submodular over finite domains, then a PI-explanation can be found in polynomial time. Assume in the following that f is finite-valued and g is defined as in equation (6). A necessary condition for $f + g$ to be submodular is that g be both min-closed and max-closed [10], where *min-closed* means $\mathcal{C}(\mathbf{x}) \wedge \mathcal{C}(\mathbf{y}) \Rightarrow \mathcal{C}(\min(\mathbf{x}, \mathbf{y}))$ and *max-closed* means $\mathcal{C}(\mathbf{x}) \wedge \mathcal{C}(\mathbf{y}) \Rightarrow \mathcal{C}(\max(\mathbf{x}, \mathbf{y}))$ [26]. Over finite domains, the class of monotone objective functions can be extended to a maximal tractable class of constrained minimisation problems by adding min-closed constraints and the class of antitone objective functions can be extended to a maximal tractable class by adding max-closed constraints [9].

As we have already seen, explanations of positive and negative decisions may have very different complexities. Indeed, a PI-explanation for $f(\mathbf{a}) \leq t$ under constraints C is a PI-explanation of $f(\mathbf{a}) - g(\mathbf{a}) \leq t$ (in the unconstrained setting). The sign of g has changed so that the inequality is satisfied whenever g is infinite. As we saw in Section 4, a PI-explanation of a negative decision of a threshold classifier can be found in polynomial time if the objective function can be maximised in polynomial time. Thus, for example, if $f - g$ is a supermodular function (over finite domains), then a PI-explanation can be found in polynomial time. A necessary condition for $f - g$ to be supermodular is that g be both min-closed and max-closed [10]. For the class of monotone functions f , the maximisation of $f - g$ is tractable if the relations C (corresponding to the functions g) are max-closed, and for the class of antitone functions f , the maximisation of $f - g$ is tractable if the relations C are min-closed [9].

This allows us to identify the tractable families of constrained threshold-classifiers listed in Table 1.

6 Contrastive explanations

PI-explanations are also known as abductive explanations, since they are answers to the question “Why is $\tau(\mathbf{a}) = c$?” A contrastive explanation [35, 22, 21] is an answer to a different question: “Why is it not the case that $\tau(\mathbf{a}) \neq c$?” It gives a set of features which if changed in \mathbf{a} can lead to a change of class. Contrastive explanations tend to be smaller than abductive explanations and hence can be easier to interpret by a human user [35].

► **Definition 10.** Given that $\tau(\mathbf{a}) = c$, a *contrastive explanation* is a subset-minimal set $\mathcal{S} \subseteq \mathcal{A}$ such that

$$\exists(\mathbf{x} \in \mathbb{A}). \left(\left(\bigwedge_{j \notin \mathcal{S}} (x_j = a_j) \right) \wedge \tau(\mathbf{x}) \neq c \right). \quad (7)$$

If $\tau \equiv c$, then there is no contrastive explanation of $\tau(\mathbf{a}) = c$.

► **Example 11.** Consider the classifier studied in Example 2: a bank uses a function τ , given by $\tau(\mathbf{x}) = \oplus$ if and only if $(\max(\text{sal}_1, \text{sal}_2) \geq \text{sal}_{\min}) \wedge (\min(\text{age}_1, \text{age}_2) \leq \text{age}_{\max})$, to decide whether to grant a loan to a couple represented by a feature vector $\mathbf{x} = (\text{sal}_1, \text{sal}_2, \text{age}_1, \text{age}_2)$. If \mathbf{a} corresponds to a couple who both earn more than sal_{\min} and both are younger than age_{\max} , then the contrastive explanations of the decision $\tau(\mathbf{a}) = \oplus$ are $\{1, 2\}$ and $\{3, 4\}$. If \mathbf{b} corresponds to a couple who both earn more than sal_{\min} but both are older than age_{\max} , then the contrastive explanations of the decision $\tau(\mathbf{b}) = \ominus$ are $\{3\}$ and $\{4\}$.

Let $\text{INVALID}(\mathcal{T})$ be the following decision problem: given a boolean function $\tau \in \mathcal{T}$, does there exists $\mathbf{x} \in \mathbb{A}$ such that $\tau(\mathbf{x}) = \ominus$. Similarly, let $\text{SAT}(\mathcal{T})$ be the problem: given a boolean function $\tau \in \mathcal{T}$, does there exists $\mathbf{x} \in \mathbb{A}$ such that $\tau(\mathbf{x}) = \oplus$. The following proposition is the contrastive equivalent of Proposition 4 and Proposition 6.

► **Proposition 12.** Suppose that \mathcal{T} is closed under fixing arguments. If $\text{INVALID}(\mathcal{T}) \in P$, then for any classifier $\tau \in \mathcal{T}$ and any \mathbf{a} such that $\tau(\mathbf{a}) = \oplus$, a contrastive explanation of $\tau(\mathbf{a}) = \oplus$ can be found in polynomial time. If $\text{SAT}(\mathcal{T}) \in P$, then for any classifier $\tau \in \mathcal{T}$ and any \mathbf{a} such that $\tau(\mathbf{a}) = \ominus$, a contrastive explanation of $\tau(\mathbf{a}) = \ominus$ can be found in polynomial time.

Proof. We say that \mathcal{S} can lead to a class change if equation (7) holds. The algorithm is analogous to the algorithm for PI-explanations. It requires n tests of equation (7) to find a contrastive explanation:

```

 $\mathcal{S} \leftarrow \{1, \dots, n\}$ 
if  $\mathcal{S}$  cannot lead to a class change then report that no CXp exists ;
for  $i = 1, \dots, n$  :
    if  $\mathcal{S} \setminus \{i\}$  can lead to a class change then  $\mathcal{S} \leftarrow \mathcal{S} \setminus \{i\}$ 
    
```

Testing whether \mathcal{S} can lead to a class change from \oplus is a test of invalidity (after fixing features in $\mathcal{A} \setminus \mathcal{S}$), whereas testing whether \mathcal{S} can lead to a class change from \ominus is a test of satisfiability (after fixing features in $\mathcal{A} \setminus \mathcal{S}$). Thus, the above algorithm finds a contrastive explanation of $\tau(\mathbf{a}) = c$ in polynomial time if $\text{INVALID}(\mathcal{T}) \in P$ (in the case $c = \oplus$) or $\text{SAT}(\mathcal{T}) \in P$ (in the case $c = \ominus$). ◀

For threshold classifiers of the form $\tau(\mathbf{x}) = \oplus$ iff $f(\mathbf{x}) > t$, invalidity corresponds to $\min_{\mathbf{x} \in \mathbb{A}} f(\mathbf{x}) \leq t$ and satisfiability corresponds to $\max_{\mathbf{x} \in \mathbb{A}} f(\mathbf{x}) > t$. Thus, if \mathcal{T} is the set of \mathcal{F} -threshold classifiers, then $\text{INVALID}(\mathcal{T}) \in P$ if functions in \mathcal{F} can be minimised in polynomial time and $\text{SAT}(\mathcal{T}) \in P$ if functions in \mathcal{F} can be maximised in polynomial time.

Let $\text{CEXPL}^+(\mathcal{T})$ (respectively, $\text{CEXPL}^-(\mathcal{T})$) be the problem of finding a contrastive explanation of a positive (negative) decision taken by a classifier in \mathcal{T} or determining that no contrastive explanation exists. The following theorem follows from Proposition 12 and the fact that deciding the existence of a contrastive explanation of $\tau(\mathbf{a}) = c$ is equivalent to deciding $\neg(\tau \equiv c)$.

► **Theorem 13.** If \mathcal{T} is closed under fixing arguments, then $\text{CEXPL}^+(\mathcal{T}) \in FP$ iff $\text{INVALID}(\mathcal{T}) \in P$, and $\text{CEXPL}^-(\mathcal{T}) \in FP$ iff $\text{SAT}(\mathcal{T}) \in P$.

In the context of constraints C , a contrastive explanation of a decision $\tau(\mathbf{a}) = c$ is now a subset-minimal set $\mathcal{S} \subseteq \mathcal{A}$ of feature literals such that

$$\exists(\mathbf{x} \in \mathbb{A}). \left(\left(\bigwedge_{j \notin \mathcal{S}} (x_j = a_j) \right) \wedge \tau(\mathbf{x}) \neq c \wedge C(\mathbf{x}) \right). \quad (8)$$

Equating \ominus with 0 and \oplus with 1, and using the logical equivalence $\neg B \wedge C \equiv \neg(B \vee \neg C)$, we have the following proposition.

► **Proposition 14.** *A contrastive explanation of a classifier τ under constraints C is precisely a contrastive explanation of the unconstrained classifier $\tau \vee \neg C$.*

In the case of constrained threshold classifiers, with objective function f and threshold t , let g be as defined by equation (6). Then testing invalidity under constraints C is equivalent to determining whether $\min_{\mathbf{x} \in \mathbb{A}} (f(\mathbf{x}) + g(\mathbf{x})) \leq t$ and testing satisfiability is equivalent to determining whether $\max_{\mathbf{x} \in \mathbb{A}} (f(\mathbf{x}) - g(\mathbf{x})) > t$. It follows that the tractable cases for finding contrastive explanations or PI-explanations are identical. Example are shown in Table 1, where, in both cases, the decision corresponds to the original decision (i.e. the value of $\tau(\mathbf{a})$).

In fact, from Theorem 5, Theorem 7, Theorem 13, Proposition 9 and Proposition 14, we can deduce the following theorem which says that tractable classes of finding abductive or contrastive explanations coincide. It follows from the fact that $\text{INVALID}(\mathcal{T}) \in \text{P}$ iff $\text{TAUTOLOGY}(\mathcal{T}) \in \text{P}$ and that $\text{SAT}(\mathcal{T}) \in \text{P}$ iff $\text{UNSAT}(\mathcal{T}) \in \text{P}$ (since a problem is in P iff its complement is in P).

► **Theorem 15.** *In the unconstrained or constrained setting, if \mathcal{T} is closed under fixing arguments, $\text{PIEXPL}^+(\mathcal{T}) \in \text{FP}$ iff $\text{CEXPL}^+(\mathcal{T}) \in \text{FP}$, and $\text{PIEXPL}^-(\mathcal{T}) \in \text{FP}$ iff $\text{CEXPL}^-(\mathcal{T}) \in \text{FP}$.*

7 A language dichotomy for threshold classifiers

We consider threshold classifiers over finite (i.e. categorical) domains whose objective function can be decomposed into functions of bounded arity:

$$f(\mathbf{x}) = \sum_{i=1}^m f_i(\mathbf{x}[\sigma_i]) \quad (9)$$

where each σ_i (the scope on which the function f_i is applied) is a list of indices from $\{1, \dots, n\}$ and $\mathbf{x}[\sigma_i]$ is the projection of the vector \mathbf{x} on these indices. Given a set (language) \mathcal{L} of functions, we denote by $\mathcal{T}_{\mathcal{L}}$ the set of threshold classifiers whose objective function f is the sum of functions $f_i \in \mathcal{L}$. Recall that $\text{PIEXPL}^+(\mathcal{T}_{\mathcal{L}})$ is the problem of finding a PI-explanation of a positive decision taken by a classifier in $\mathcal{T}_{\mathcal{L}}$.

Cost Function Networks (CFNs) (also known as Valued Constraint Satisfaction Problems) are defined by sets of functions f_i (and their associated scopes) over finite domains whose sum f (given by equation (9)) is an objective function to be minimized [11]. CFNs are a generic framework covering many well-studied optimisation problems. For example, Bayesian networks can be transformed into CFNs after taking logarithms of probabilities [11]. Let $\text{CFN}(\mathcal{L})$ denote the problem of determining, given an objective function f of the form given in equation (9) where each $f_i \in \mathcal{L}$, together with a real constant t , whether

$$\min f(\mathbf{x}) \leq t.$$

A technical point is that, due to the necessarily bounded precision of the values of functions, this is equivalent to the problem of determining, given f and $t \in \mathbb{R}$, whether $\min f(\mathbf{x})$ is strictly less than t .

The complexity of $\text{CFN}(\mathcal{L})$ has been extensively studied for finite languages (i.e. languages \mathcal{L} such that $|\mathcal{L}|$ is finite). It is now known that there is a dichotomy: depending on the language \mathcal{L} , $\text{CFN}(\mathcal{L})$ is either in P or is NP-complete. This result was known for languages of finite-valued cost-functions [41] and the dichotomy for the more general case, in which costs can be infinite, follows from the recently-discovered language dichotomy for constraint satisfaction problems [4, 44, 29, 30]. The following proposition will lead us to a similar dichotomy for explaining decisions.

► **Proposition 16.** *Let \mathcal{L} be a set of non-negative functions closed under fixing arguments. Then $\text{PIEXPL}^+(\mathcal{T}_{\mathcal{L}})$ is in FP if and only if $\text{CFN}(\mathcal{L})$ is in P.*

Proof. If \mathcal{L} is closed under fixing arguments, then so is $\mathcal{T}_{\mathcal{L}}$. The “if” part of the proof follows directly from Proposition 4 and the subsequent discussion in Section 3, so we concentrate on the “only if” part.

By Theorem 5 we know that if $\text{PIEXPL}^+(\mathcal{T}_{\mathcal{L}}) \in \text{FP}$ then $\text{TAUTOLOGY}(\mathcal{T}_{\mathcal{L}}) \in \text{P}$. $\text{TAUTOLOGY}(\mathcal{T}_{\mathcal{L}})$ is the problem of determining, for a function f expressible as the sum of functions $f_i \in \mathcal{L}$ (as in equation (9)) and a constant t , whether $f(x) > t$ for all $x \in \mathbb{A}$. This is the complement of $\text{CFN}(\mathcal{L})$ which is the problem of determining whether $\min_{x \in \mathbb{A}} f(x) \leq t$. Hence, if $\text{TAUTOLOGY}(\mathcal{T}_{\mathcal{L}}) \in \text{P}$ then $\text{CFN}(\mathcal{L}) \in \text{P}$, which completes the proof. ◀

We now consider constrained classifiers. Let Γ be a language of constraint relations. For each constraint relation in Γ we can construct a corresponding $\{0, \infty\}$ -valued function g , as given by equation (6). Let \mathcal{C}_{Γ} denote the set of all such $\{0, \infty\}$ -valued functions for relations in Γ . Then $\mathcal{L} \cup \mathcal{C}_{\Gamma}$ can be viewed as a language of cost functions. Let $\text{CONPIEXPL}^+(\mathcal{T}_{\mathcal{L}}, \Gamma)$ (respectively, $\text{CONPIEXPL}^-(\mathcal{T}_{\mathcal{L}}, \Gamma)$) denote the problem of finding one PI-explanation of a positive (negative) decision taken by a classifier in $\mathcal{T}_{\mathcal{L}}$ under a finite set of constraints from Γ .

► **Proposition 17.** *Let \mathcal{L} be a set of non-negative functions closed under fixing arguments and Γ a finite set of constraint relations. Then $\text{CONPIEXPL}^+(\mathcal{T}_{\mathcal{L}}, \Gamma)$ is in FP if and only if $\text{CFN}(\mathcal{L} \cup \mathcal{C}_{\Gamma})$ is in P.*

Proof. We know from the discussion in Section 5 that $\text{CONPIEXPL}^+(\mathcal{T}_{\mathcal{L}}, \Gamma)$ is equivalent to $\text{PIEXPL}^+(\mathcal{T}_{\mathcal{L} \cup \mathcal{C}_{\Gamma}})$. Thus the result follows immediately from Proposition 16. ◀

We now consider finding explanations for negative decisions. Although, as we will show, there is again a dichotomy, it is not the same since in this case we are studying a (constrained) maximisation problem rather than a (constrained) minimisation problem. Given a finite language \mathcal{L} of real-valued functions, all bounded above by $B \in \mathbb{R}$, let \mathcal{L}_{inv} denote the set $\{B - f : f \in \mathcal{L}\}$. Clearly, maximising a sum of functions from \mathcal{L} is equivalent to minimising a sum of functions from \mathcal{L}_{inv} .

► **Proposition 18.** *Let \mathcal{L} be a set of non-negative finite-valued functions closed under fixing arguments. Then $\text{PIEXPL}^-(\mathcal{T}_{\mathcal{L}})$ is in FP if and only if $\text{CFN}(\mathcal{L}_{\text{inv}})$ is in P.*

Proof. The “if” part follows from Proposition 6 and the subsequent discussion in Section 4. For the “only if” part, we know from Theorem 7 that if $\text{PIEXPL}^-(\mathcal{T}_{\mathcal{L}})$ is in FP then $\text{UNSAT}(\mathcal{T}_{\mathcal{L}})$ is in P. $\text{UNSAT}(\mathcal{T}_{\mathcal{L}})$ is the problem of determining, for a function f expressible as the sum of m functions $f_i \in \mathcal{L}$ and a constant t , whether $f(x) \leq t$ for all $x \in \mathbb{A}$. This is equivalent to determining whether $mB - f(x) \geq mB - t$ for all $x \in \mathbb{A}$. This is the complement of the problem of determining whether $\min(mB - f) < t'$ (for $t' = mB - t$). This is precisely $\text{CFN}(\mathcal{L}_{\text{inv}})$. Hence, if $\text{UNSAT}(\mathcal{T}_{\mathcal{L}}) \in \text{P}$, then $\text{CFN}(\mathcal{L}_{\text{inv}}) \in \text{P}$, which completes the proof. ◀

We now generalise this result to constrained classifiers.

► **Proposition 19.** *Let \mathcal{L} be a set of non-negative functions closed under fixing arguments and Γ a finite set of constraint relations. Then $\text{CONPIEXPL}^-(\mathcal{T}_{\mathcal{L}}, \Gamma)$ is in FP if and only if $\text{CFN}(\mathcal{L}_{\text{inv}} \cup \mathcal{C}_{\Gamma})$ is in P.*

Proof. It is easy to see that $\text{CONPIEXPL}^-(\mathcal{T}_{\mathcal{L}}, \Gamma)$ is equivalent to $\text{CONPIEXPL}^+(\mathcal{T}_{\mathcal{L}_{\text{inv}}}, \Gamma)$. Thus the result follows immediately from Proposition 17. ◀

Given the known P/NP-complete dichotomy for $\text{CFN}(\mathcal{L})$ for finite languages \mathcal{L} , discussed above, we can immediately deduce the following theorem.

► **Theorem 20.** *Let \mathcal{L} be a finite language of non-negative functions closed under fixing arguments and Γ a finite set of constraint relations. Then each of $\text{PIEXPL}^+(\mathcal{T}_{\mathcal{L}})$, $\text{CONPIEXPL}^+(\mathcal{T}_{\mathcal{L}}, \Gamma)$, $\text{PIEXPL}^-(\mathcal{T}_{\mathcal{L}})$, $\text{CONPIEXPL}^-(\mathcal{T}_{\mathcal{L}}, \Gamma)$ is either in FP or is NP-hard.*

Indeed, by Theorem 15, we have an identical dichotomy result for contrastive explanations.

8 Diversity of explanations

We have concentrated up until now on the problem of finding a single explanation. This is because the problem of finding all explanations has the obvious disadvantage that the number of explanations may be exponential. For example, in a first-past-the-post election in which a A wins with $m \geq k$ out of the $n = 2k - 1$ votes cast, and each vote is considered as a feature, there are C_m^k PI-explanations for this victory; for a candidate B who lost with only $p \leq k$ votes, there are C_{n-p}^{k-p} contrastive explanations for why they did not win.

Rather than providing a single explanation to the user or listing all explanations, we can envisage providing a relatively small number of *diverse* explanations. A similar strategy of finding a number of diverse good-quality solutions to a Weighted Constraint Satisfaction Problem has been used successfully in computational protein design [37], among other examples [18, 19, 25].

An obvious measure of diversity of a set of explanations $\{S_1, \dots, S_k\}$ is the minimum Hamming distance $|S_i \Delta S_j|$ between pairs of distinct explanations S_i, S_j , where Δ is the symmetric difference operator between two sets. This leads to the following computational problem.

k -DIV-PIEXPL⁺: Given a binary classifier $\tau : \mathbb{A} \rightarrow \{\ominus, \oplus\}$, a positively-classified input \mathbf{a} and an integer m , find k PI-explanations S_1, \dots, S_k of $\tau(\mathbf{a}) = \oplus$ such that for all i, j such that $1 \leq i < j \leq k$, $|S_i \Delta S_j| \geq m$.

The definitions for negatively-classified inputs \mathbf{a} (k -DIV-PIEXPL[−]) and/or for contrastive explanations (k -DIV-CEXPL⁺, k -DIV-CEXPL[−]) are entirely similar. Since Hamming distance is a submodular function, one might hope that there would be interesting tractable classes. Unfortunately, since we are, in a sense, maximising this distance rather than minimising it, these four problems turn out to be NP-hard even in the simplest non-trivial case.

► **Proposition 21.** *Even in the case of $k = 2$ and for a linear classifier τ over domains of size 2, the following four problems are NP-hard: (a) k -DIV-PIEXPL⁺, (b) k -DIV-PIEXPL[−], (c) k -DIV-CEXPL⁺, (d) k -DIV-CEXPL[−].*

Proof.

(a) Without loss of generality, we suppose that the domains D_i ($i = 1, \dots, n$) are all $\{0, 1\}$ and $\tau(\mathbf{x}) = \oplus$ iff $\sum_{i=1}^n \alpha_i x_i > t$. We prove NP-hardness for the particular case in which $\mathbf{a} = (1, \dots, 1)$ and the values $t, \alpha_1, \dots, \alpha_n$ are strictly positive integers which satisfy the following inequalities:

$$\alpha_1 \leq \dots \leq \alpha_m < \alpha_{m+1} \leq \dots \leq \alpha_n \quad (10)$$

$$\sum_{i=1}^m \alpha_i + 2 \sum_{i=m+1}^n \alpha_i = 2(t+1). \quad (11)$$

To solve 2-DIV-PIEXPL⁺ we require sets $S_1, S_2 \subseteq \{1, \dots, n\}$ satisfying (1) $|S_1 \Delta S_2| \geq m$ and (2) S_1, S_2 are minimal (for inclusion) sets such that the minimum value of $\sum_{i=1}^n \alpha_i x_i$ is at least $t+1$ for inputs \mathbf{x} with $x_i = a_i = 1$ for all $i \in S_j$ ($j = 1, 2$). Since the values α_i are positive, the minimum is attained when $x_i = 0$ for all $i \notin S_j$, and so this is equivalent to

$$\sum_{i \in S_j} \alpha_i \geq t+1 \quad (j = 1, 2). \quad (12)$$

Summing these two inequalities (for $j = 1, 2$) gives

$$\sum_{i \in S_1} \alpha_i + \sum_{i \in S_2} \alpha_i \geq 2(t+1). \quad (13)$$

Since, by (10), we have $\alpha_r < \alpha_s$ for $r \leq m < s$, and $|S_1 \Delta S_2| \geq m$, we know that the left hand side of the sum in equation (13) is at most equal to the left hand side of equation (11), which is equal to $2(t+1)$. It follows that we actually have equality in inequality (13) and $S_1 \Delta S_2 = \{1, \dots, m\}$ and $S_1 \cap S_2 = \{m+1, \dots, n\}$. Equality in (13) implies that we must also have equality in the inequalities (12) for $j = 1, 2$. Equality implies minimality for subset inclusion since all weights α_i are strictly positive. Denoting $t+1 - \sum_{i=m+1}^n \alpha_i$ by T and $S_j \cap \{1, \dots, m\}$ by P_j (for $j = 1, 2$), we can deduce that we require a partition P_1, P_2 of $\{1, \dots, m\}$ such that

$$\sum_{i \in P_1} \alpha_i = T = \sum_{i \in P_2} \alpha_i.$$

This is precisely the partition problem which is well known to be NP-complete [28]. It follows that k -DIV-PIEXPL⁺ is NP-hard.

- (b) We consider the same linear classifier τ as in case (a), except that equation (11) is replaced by $\sum_{i=1}^m \alpha_i = 2t$, and this time we consider the vector $\mathbf{a} = (0, \dots, 0)$ which is classified negatively by τ . To solve k -DIV-PIEXPL⁻, we require two sets S_1, S_2 such that (1) $|S_1 \Delta S_2| \geq m$ and (2) S_1, S_2 are minimal (for inclusion) sets such that $\sum_{i \notin S_j} \alpha_i \leq t$ ($j = 1, 2$). Given equation (10), this can only be attained when $S_1 \Delta S_2 = \{1, \dots, m\}$ and $S_1 \cap S_2 = \{m+1, \dots, n\}$, so that $\sum_{i \notin S_1} \alpha_i = \sum_{i \notin S_2} \alpha_i = t$. Thus, we need to find two sets $P_j = \{1, \dots, n\} \setminus S_j$ ($j = 1, 2$) which partition $\{1, \dots, m\}$ and such that

$$\sum_{i \in P_1} \alpha_i = t = \sum_{i \in P_2} \alpha_i$$

Thus, again we have a polynomial reduction from the partition problem. Hence k -DIV-PIEXPL⁻ is NP-hard.

- (c) Consider the same linear classifier τ as in case (b), but this time $\mathbf{a} = (1, \dots, 1)$. To solve k -DIV-CEXPL⁺, we require two sets $S_1, S_2 \subseteq \{1, \dots, n\}$ such that $\sum_{i \notin S_j} \alpha_i \leq t$ ($j = 1, 2$) and $|S_1 \Delta S_2| \geq m$. Since this is exactly the same problem encountered in case (b), we can again deduce NP-hardness.

- (d) Consider the same linear classifier τ as in case (a), but with $\mathbf{a} = (0, \dots, 0)$. To solve $k\text{-DIV-CEXPL}^-$, we require $S_1, S_2 \subseteq \{1, \dots, n\}$ such that $\sum_{i \in S_j} \alpha_i \geq t + 1$ ($j = 1, 2$) and $|S_1 \Delta S_2| \geq m$. Since this is exactly the problem encountered in case (a), we can again deduce NP-hardness. \blacktriangleleft

It is well known that the partition problem is one of the easiest NP-hard problems to solve in practice [38]. Thus, Proposition 21 precludes (assuming $P \neq NP$) a worst-case polynomial-time algorithm for finding a diverse set of explanations, but leaves the door open to the existence of practically-efficient algorithms.

9 Absolute explanations

Given a classifier we may want to have an absolute (global) explanation for a given class c , rather than an explanation specific to a particular decision. This answers questions of the type “Why can a customer be granted (or refused) a loan”. An absolute explanation is a minimal but arbitrary partial assignment to the features that guarantees that the output of the classifier τ will be the class c [24]. It does not depend on a concrete input instance but rather the entire feature space.

A *literal* is an assignment of a value to a feature which we can write in the form $(x_i = u)$ or simply as the pair $\langle i, u \rangle$ where $i \in \mathcal{A}$ and u belongs to D_i the domain of feature i . A set of literals \mathcal{U} is *well-defined* if each feature i occurs at most once in \mathcal{U} . For simplicity of presentation, we implicitly assume from now on that all subsets of literals are well-defined. This means that each subset of literals \mathcal{U} corresponds to a partial assignment \mathbf{a} to some subset of features $\mathcal{P} \subseteq \mathcal{A}$. This allows us to equate \mathcal{U} with the pair $\langle \mathcal{P}, \mathbf{a} \rangle$.

► **Definition 22.** *Given a classifier τ , an absolute explanation (XP) for a class c is a subset-minimal set of literals $\mathcal{U} = \langle \mathcal{P}, \mathbf{a} \rangle$ such that*

$$\forall (\mathbf{x} \in \mathbb{A}). \bigwedge_{j \in \mathcal{P}} (x_j = a_j) \rightarrow \tau(\mathbf{x}) = c \quad (14)$$

is true.

Equation (14) is identical to equation (1) in the definition of a PI-explanation, the difference being that an XP is a set of literals rather than a set of features. A subtle difference between PI-explanations and XP’s is that whereas a PI-explanation always exists, since we are given an instance \mathbf{a} such that $\tau(\mathbf{a}) = c$, an XP may not exist (which corresponds to the case when τ never takes the value c).

Associating a set of literals with the term corresponding to their conjunction, we can observe that a model τ is logically equivalent to the disjunction of the absolute explanations for the class \oplus . This observation shows that, in the case of finite domains, a black-box model can in theory be reconstructed from its absolute explanations.

Another global notion, dual to the notion of absolute explanation, is that of a counterexample [24]. This is an answer to questions such as “Why can a customer not be granted a loan”.

► **Definition 23.** *Given a classifier τ , a counterexample (CEX) for a class c is a subset-minimal set of literals $\mathcal{U} = \langle \mathcal{P}, \mathbf{a} \rangle$ such that*

$$\forall (\mathbf{x} \in \mathbb{A}). \bigwedge_{j \in \mathcal{P}} (x_j = a_j) \rightarrow \tau(\mathbf{x}) \neq c \quad (15)$$

is true.

Clearly, in the case of binary classifiers with $\mathcal{K} = \{\ominus, \oplus\}$, a counterexample for class \oplus (\ominus) is an absolute explanation of class \ominus (\oplus). Analogously to the fact, observed above, that a model τ is logically equivalent to the disjunction of the absolute explanations for the class \oplus , it is also logically equivalent to the conjunction of the negations of the counterexamples of the class \oplus .

► **Example 24.** We return to the function τ used by a bank to decide whether to grant a loan to a couple represented by a feature vector $\mathbf{x} = (sal_1, sal_2, age_1, age_2)$, as in Example 2: $\tau(\mathbf{x}) = \oplus$ if and only if $(\max(sal_1, sal_2) \geq sal_{\min}) \wedge (\min(age_1, age_2) \leq age_{\max})$. If $s_0 \geq sal_{\min} > s_1, s_2$ and $a_0 \leq age_{\max}$, then $\{\langle 1, s_0 \rangle, \langle 3, a_0 \rangle\}$ is an XP of a positive decision (and a CEx of a negative decision) whereas $\{\langle 1, s_1 \rangle, \langle 2, s_2 \rangle\}$ is an XP of a negative decision (and a CEx of a positive decision).

Despite the similarity between the definitions of PI-explanations and absolute explanations, the complexity of finding one XP is not the same as the complexity of finding one PI-explanation. This is due to the fact that we are not given a specific instance, but rather a class c , and we actually have to find an instance which belongs to class c . Let $XP^+(\mathcal{T})$ (respectively, $XP^-(\mathcal{T})$) be the problem of finding an absolute explanation of a positive (negative) decision taken by a classifier in \mathcal{T} (or returning “none” if none exists).

► **Theorem 25.** *If \mathcal{T} is closed under fixing arguments, and domains of all features are finite, then $XP^+(\mathcal{T}) \in FP$ iff $SAT(\mathcal{T}) \in P$ and $TAUTOLOGY(\mathcal{T}) \in P$.*

Proof. For the “if” part, it is sufficient to give a polynomial-time algorithm. Consider $\tau \in \mathcal{T}$. A call to $SAT(\mathcal{T})$ tells us whether or not an XP exists. In the case that an XP exists, we can find an instance \mathbf{a} such that $\tau(\mathbf{a}) = \oplus$ by the following incremental algorithm.

```

Initialise  $\mathbf{a}$  to the empty assignment ;
for  $i = 1, \dots, n$  :
    for each value  $d \in D_i$ 
        extend  $\mathbf{a}$  by assigning  $a_i = d$ ;
        if  $\tau_{\mathbf{a}}$  is satisfiable then exit the inner for loop;

```

The partial assignment \mathbf{a} is initialised to the empty assignment and successively, for each feature i , at most $|D_i|$ calls to $SAT(\mathcal{T})$ are sufficient to find a value for a_i which extends the present partial assignment so that $\tau_{\mathbf{a}}$ remains satisfiable. The final value of \mathbf{a} is a complete assignment such that $\tau(\mathbf{a}) = \oplus$. Since $TAUTOLOGY(\mathcal{T}) \in P$, by Proposition 4 we can find a PI-explanation \mathcal{P} of $\tau(\mathbf{a}) = \oplus$ in polynomial time. Then $\langle \mathcal{P}, \mathbf{a}[\mathcal{P}] \rangle$ is necessarily an XP, where $\mathbf{a}[\mathcal{P}]$ is the partial assignment of \mathbf{a} on features \mathcal{P} .

For the “only if” part, an XP exists iff $\tau \not\equiv \ominus$ and is non-empty iff $\tau \not\equiv \oplus$. Hence, a polynomial-time algorithm for $XP^+(\mathcal{T})$ necessarily decides both $SAT(\mathcal{T})$ and $TAUTOLOGY(\mathcal{T})$ in polynomial time. ◀

► **Corollary 26.** *If \mathcal{T} is closed under fixing arguments, and domains of all features are finite, then $XP^-(\mathcal{T}) \in FP$ iff $SAT(\mathcal{T}) \in P$ and $TAUTOLOGY(\mathcal{T}) \in P$.*

Proof. First, observe that an absolute explanation (XP) of a negative decision taken by a classifier τ is an XP of a positive decision taken by the classifier $\bar{\tau}$. To complete the proof, it suffices to notice that $\bar{\tau}$ is satisfiable iff τ is not a tautology (and $\bar{\tau}$ is a tautology iff τ is not satisfiable). So, by Theorem 25, $XP^-(\mathcal{T}) \in FP$ iff $SAT(\mathcal{T}) \in P$ and $TAUTOLOGY(\mathcal{T}) \in P$. ◀

For a family \mathcal{T} of threshold classifiers, Theorem 25 implies that $XP^+(\mathcal{T}) \in FP$ iff the corresponding family of objective functions can be both minimized and maximized in polynomial time. Examples are monotone functions and modular functions. Modular functions, which by definition are both submodular and supermodular, are separable (i.e. expressible as the sum of unary functions on the features) [42].

In the case of constrained classifiers we have the following proposition which follows directly from Proposition 9.

► **Proposition 27.** *An absolute explanation (XP) of a classifier τ under constraints C is precisely an XP of the unconstrained classifier $\tau \vee \neg C$.*

With f the objective function of the threshold classifier τ and g the function, given by Equation 6, associated with the constraints C , $TAUTOLOGY(\mathcal{T})$ corresponds to minimizing $f + g$ and $SAT(\mathcal{T})$ corresponds to maximizing $f - g$. By Theorem 25 together with the discussion above and in Section 5, $XP^+(\mathcal{T})$ is tractable for objective functions f which are either modular, monotone or antitone and constraint relations C which are both min and max-closed.

10 Discussion and Conclusion

We have investigated the complexity of finding one subset-minimal abductive or contrastive explanation for different families of classifiers.

There remain many interesting open questions:

- Since, as yet, there is no known characterisation of the complexity of cost-function languages over infinite domains, the complexity of classifiers with real-valued features is still an open problem.
- We have investigated the problem of finding a subset-minimal explanation. The problem of finding a cardinality-minimum explanation is naturally harder [39, 3] even though it has been observed that there is often not a significant difference between the size of subset-minimal and cardinality-minimum explanations [23]. It is known that the problem of finding a cardinality-minimum explanation is NP-hard for decision trees [3] and is tractable for linear classifiers [33]. It is an open theoretical question whether there are any other interesting tractable cases.
- Instead of searching for one explanation, we may want to find many explanations. Unfortunately, the fact that a greedy algorithm can find one explanation in polynomial time provides no guarantee that explanations can be enumerated in polynomial delay. Again, for linear classifiers, there is a polynomial-delay algorithm for enumerating PI-explanations [33], and it is an open question whether this is true for other families of classifiers. It is known to be false for monotone classifiers (assuming $P \neq NP$) [34].

References

- 1 Gilles Audemard, Steve Bellart, Louenas Bounia, Frédéric Koriche, Jean-Marie Lagniez, and Pierre Marquis. On the computational intelligibility of boolean classifiers. *CoRR*, abs/2104.06172, 2021. [arXiv:2104.06172](#).
- 2 Gilles Audemard, Frédéric Koriche, and Pierre Marquis. On tractable XAI queries based on compiled representations. In *KR*, pages 838–849, 2020. [doi:10.24963/kr.2020/86](#).

- 3 Pablo Barceló, Mikaël Monet, Jorge Pérez, and Bernardo Subercaseaux. Model interpretability through the lens of computational complexity. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *NeurIPS 2020*, 2020. URL: <https://proceedings.neurips.cc/paper/2020/hash/b1adda14824f50ef24ff1c05bb66faf3-Abstract.html>.
- 4 Andrei A. Bulatov. A dichotomy theorem for nonuniform CSPs. In *FOCS*, pages 319–330, 2017. doi:10.1109/FOCS.2017.37.
- 5 Rainer E. Burkard, Bettina Klinz, and Rüdiger Rudolf. Perspectives of Monge properties in optimization. *Discret. Appl. Math.*, 70(2):95–161, 1996.
- 6 Deeparnab Chakrabarty, Yin Tat Lee, Aaron Sidford, and Sam Chiu-wai Wong. Subquadratic submodular function minimization. In *STOC*, pages 1220–1231, 2017.
- 7 Zhi-Zhong Chen and Seinosuke Toda. The complexity of selecting maximal solutions. *Inf. Comput.*, 119(2):231–239, 1995. doi:10.1006/inco.1995.1087.
- 8 David A. Cohen, Martin C. Cooper, Peter Jeavons, and Andrei A. Krokhin. A maximal tractable class of soft constraints. *J. Artif. Intell. Res.*, 22:1–22, 2004.
- 9 David A. Cohen, Martin C. Cooper, Peter Jeavons, and Andrei A. Krokhin. The complexity of soft constraint satisfaction. *Artif. Intell.*, 170(11):983–1016, 2006.
- 10 Martin C. Cooper, Simon de Givry, Martí Sánchez-Fibla, Thomas Schiex, Matthias Zytnicki, and Tomás Werner. Soft arc consistency revisited. *Artif. Intell.*, 174(7-8):449–478, 2010.
- 11 Martin C. Cooper, Simon de Givry, and Thomas Schiex. Graphical models: Queries, complexity, algorithms (tutorial). In *STACS*, pages 4:1–4:22, 2020.
- 12 Nadia Creignou, Sanjeev Khanna, and Madhu Sudan. *Complexity classifications of Boolean constraint satisfaction problems*, volume 7 of *SIAM monographs on discrete mathematics and applications*. SIAM, 2001.
- 13 Adnan Darwiche. Three modern roles for logic in AI. In *PODS*, pages 229–243, 2020. doi:10.1145/3375395.3389131.
- 14 Adnan Darwiche and Auguste Hirth. On the reasons behind decisions. In *ECAI*, pages 712–720, 2020. doi:10.3233/FAIA200158.
- 15 Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002. doi:10.1613/jair.989.
- 16 Satoru Fujishige. *Submodular Functions and Optimisation*, volume 58 of *Annals of Discrete Mathematics*. Elsevier, 2nd edition, 2005.
- 17 Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. A survey of methods for explaining black box models. *ACM Comput. Surv.*, 51(5):93:1–93:42, 2019. doi:10.1145/3236009.
- 18 Emmanuel Hebrard, Brahim Hnich, Barry O'Sullivan, and Toby Walsh. Finding diverse and similar solutions in constraint programming. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence*, pages 372–377. AAAI Press / The MIT Press, 2005. URL: <http://www.aaai.org/Library/AAAI/2005/aaai05-059.php>.
- 19 John Horan and Barry O'Sullivan. Towards diverse relaxations of over-constrained models. In *ICTAI 2009, 21st IEEE International Conference on Tools with Artificial Intelligence*, pages 198–205. IEEE Computer Society, 2009. doi:10.1109/ICTAI.2009.89.
- 20 Alexey Ignatiev. Towards trustable explainable AI. In *IJCAI*, pages 5154–5158, 2020. doi:10.24963/ijcai.2020/726.
- 21 Alexey Ignatiev, Nina Narodytska, Nicholas Asher, and João Marques-Silva. From contrastive to abductive explanations and back again. In Matteo Baldoni and Stefania Bandini, editors, *AIxIA 2020*, volume 12414 of *Lecture Notes in Computer Science*, pages 335–355. Springer, 2020. doi:10.1007/978-3-030-77091-4_21.
- 22 Alexey Ignatiev, Nina Narodytska, Nicholas Asher, and João Marques-Silva. On relating ‘why?’ and ‘why not?’ explanations. *CoRR*, abs/2012.11067, 2020. arXiv:2012.11067.

- 23 Alexey Ignatiev, Nina Narodytska, and João Marques-Silva. Abduction-based explanations for machine learning models. In *AAAI*, pages 1511–1519, 2019. doi:10.1609/aaai.v33i01.33011511.
- 24 Alexey Ignatiev, Nina Narodytska, and João Marques-Silva. On relating explanations and adversarial examples. In *NeurIPS*, pages 15857–15867, 2019. URL: <http://papers.nips.cc/paper/9717-on-relating-explanations-and-adversarial-examples>.
- 25 Linnea Ingmar, Maria Garcia de la Banda, Peter J. Stuckey, and Guido Tack. Modelling diversity of solutions. In *AAAI 2020*, pages 1528–1535. AAAI Press, 2020. URL: <https://aaai.org/ojs/index.php/AAAI/article/view/5512>.
- 26 Peter Jeavons and Martin C. Cooper. Tractable constraints on ordered domains. *Artif. Intell.*, 79(2):327–339, 1995. doi:10.1016/0004-3702(95)00107-7.
- 27 Matthew Joseph, Michael J. Kearns, Jamie Morgenstern, and Aaron Roth. Fairness in learning: Classic and contextual bandits. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *NIPS 2016*, pages 325–333, 2016. URL: <https://proceedings.neurips.cc/paper/2016/hash/eb163727917cbb1ee208541a643e74-Abstract.html>.
- 28 Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi:10.1007/978-1-4684-2001-2_9.
- 29 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.
- 30 Andrei A. Krokhin and Stanislav Zivný. The complexity of valued CSPs. In Andrei A. Krokhin and Stanislav Zivný, editors, *The Constraint Satisfaction Problem: Complexity and Approximability*, volume 7 of *Dagstuhl Follow-Ups*, pages 233–266. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/DFU.Vol7.15301.9.
- 31 Yin Tat Lee, Aaron Sidford, and Sam Chiu-wai Wong. A faster cutting plane method and its implications for combinatorial and convex optimization. In *FOCS*, pages 1049–1065, 2015.
- 32 Xingchao Liu, Xing Han, Na Zhang, and Qiang Liu. Certified monotonic neural networks. In Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *NeurIPS 2020*, 2020. URL: <https://proceedings.neurips.cc/paper/2020/hash/b139aeda1c2914e3b579aafd3ceeb1bd-Abstract.html>.
- 33 João Marques-Silva, Thomas Gerspacher, Martin C. Cooper, Alexey Ignatiev, and Nina Narodytska. Explaining naive Bayes and other linear classifiers with polynomial time and delay. In Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *NeurIPS 2020*, 2020. URL: <https://proceedings.neurips.cc/paper/2020/hash/eccd2a86bae4728b38627162ba297828-Abstract.html>.
- 34 João Marques-Silva, Thomas Gerspacher, Martin C. Cooper, Alexey Ignatiev, and Nina Narodytska. Explanations for monotonic classifiers. In Marina Meila and Tong Zhang, editors, *ICML 2021*, volume 139 of *Proceedings of Machine Learning Research*, pages 7469–7479. PMLR, 2021. URL: <http://proceedings.mlr.press/v139/marques-silva21a.html>.
- 35 Tim Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artif. Intell.*, 267:1–38, 2019.
- 36 James B. Orlin. A faster strongly polynomial time algorithm for submodular function minimization. *Math. Program.*, 118(2):237–251, 2009. doi:10.1007/s10107-007-0189-2.
- 37 Manon Ruffini, Jelena Vucinic, Simon de Givry, George Katsirelos, Sophie Barbe, and Thomas Schiex. Guaranteed diversity & quality for the weighted CSP. In *ICTAI 2019*, pages 18–25. IEEE, 2019. doi:10.1109/ICTAI.2019.00012.
- 38 Ethan L. Schreiber, Richard E. Korf, and Michael D. Moffitt. Optimal multi-way number partitioning. *J. ACM*, 65(4):24:1–24:61, 2018. doi:10.1145/3184400.
- 39 Andy Shih, Arthur Choi, and Adnan Darwiche. A symbolic approach to explaining bayesian network classifiers. In *IJCAI*, pages 5103–5111, 2018. doi:10.24963/ijcai.2018/708.

- 40 Andy Shih, Arthur Choi, and Adnan Darwiche. Compiling bayesian network classifiers into decision graphs. In *AAAI*, pages 7966–7974, 2019. doi:10.1609/aaai.v33i01.33017966.
- 41 Johan Thapper and Stanislav Zivny. The complexity of finite-valued CSPs. *J. ACM*, 63(4):37:1–37:33, 2016.
- 42 Donald M. Topkis. Minimizing a submodular function on a lattice. *Oper. Res.*, 26(2):305–321, 1978. doi:10.1287/opre.26.2.305.
- 43 Christopher Umans. The minimum equivalent DNF problem and shortest implicants. *J. Comput. Syst. Sci.*, 63(4):597–611, 2001. doi:10.1006/jcss.2001.1775.
- 44 Dmitriy Zhuk. A proof of CSP dichotomy conjecture. In *FOCS*, pages 331–342, 2017. doi:10.1109/FOCS.2017.38.

A Collection of Constraint Programming Models for the Three-Dimensional Stable Matching Problem with Cyclic Preferences

Ágnes Cseh ✉ 

Hasso-Plattner-Institute, Universität Potsdam, Germany


Institute of Economics, Centre for Economic and Regional Studies, Pécs, Hungary

Guillaume Escamocher ✉ 

Insight Centre for Data Analytics, School of Computer Science and Information Technology,
University College Cork, Ireland

Begüm Genç ✉ 

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

We introduce five constraint models for the 3-dimensional stable matching problem with cyclic preferences and study their relative performances under diverse configurations. While several constraint models have been proposed for variants of the two-dimensional stable matching problem, we are the first to present constraint models for a higher number of dimensions. We show for all five models how to capture two different stability notions, namely weak and strong stability. Additionally, we translate some well-known fairness notions (i.e. sex-equal, minimum regret, egalitarian) into 3-dimensional matchings, and present how to capture them in each model.

Our tests cover dozens of problem sizes and four different instance generation methods. We explore two levels of commitment in our models: one where we have an individual variable for each agent (individual commitment), and another one where the determination of a variable involves pairing the three agents at once (group commitment). Our experiments show that the suitability of the commitment depends on the type of stability we are dealing with. Our experiments not only led us to discover dependencies between the type of stability and the instance generation method, but also brought light to the role that learning and restarts can play in solving this kind of problems.

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, fairness

Digital Object Identifier 10.4230/LIPIcs.CP.2021.22

Supplementary Material *Dataset:* <https://doi.org/10.5281/zenodo.5156119>

Funding This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant numbers 12/RC/2289-P2, 16/SP/3804 and 16/RC/3918, which are co-funded under the European Regional Development Fund.

Ágnes Cseh: The Federal Ministry of Education and Research of Germany in the framework of KI-LAB-ITSE (project number 01IS19066), OTKA grant K128611, János Bolyai Research Fellowship.

Acknowledgements COST Action CA16228 European Network for Game Theory.



© Ágnes Cseh, Guillaume Escamocher, Begüm Genç, and Luis Quesada;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 22; pp. 22:1–22:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In the classic stable marriage problem, we are given a bipartite graph, where the two sets of vertices represent men and women, respectively. Each vertex has a strictly ordered preference list over his or her possible partners. A matching is *stable* if it is not *blocked* by any edge, that is, no man-woman pair exists who are mutually inclined to abandon their partners and marry each other. Stable matchings were first formally defined in the seminal paper of Gale and Shapley [20], who introduced the terminology based on marriage that since then became wide-spread. The notion was then extended to non-bipartite graphs by Irving [27]. Variants of stable matching problems are widely used in employer allocation markets [44], university admission decisions [3, 7], campus housing assignments [8, 42] and bandwidth allocation [19]. Typically, the aim is to solve the decision problem on whether a stable matching exists, or even to solve an optimisation problem considering different fairness notions among stable matchings, such as egalitarian, minimum-regret, or sex-equal.

A natural generalisation of the problem, as suggested by Knuth in his influential book [31], is to extend the two-sided stable marriage problem to three sets of agents. Two input variants of this extension have been defined in the literature. In the first variant, called the *3-gender stable marriage problem* (3GSM) problem [2, 38], each agent has a preference list over the n^2 pairs of agents from the other two sets, assuming that each agent set contains n agents. Another way of generalising stable matching to three agent sets is the *3-dimensional stable matching problem with cyclic preferences* (3DSM-CYC) [38], 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 agents from the third set only have preferences over agents from the first set. In both problem variants, the aim is to find a matching that does not admit a blocking triple, where a blocking triple can have slightly different definitions depending on whether the preference lists contain ties or whether a strict improvement for all agents is required. We explore these different notions in Section 1.1.

1.1 3-dimensional stable matching

In the 3GSM problem variant, the default stability notion is called weak stability, according to which a blocking triple is defined as a set of three agents, all of whom would strictly improve their current match if they would form a triple in the solution. Deciding whether a stable matching exists in a given instance is NP-complete even if the preference lists are complete [38, 47]. A highly restricted preference structure was later identified that allows for a polynomial-time algorithm for the same decision problem [12]. Research then evolved in the direction of preference lists with ties, which gives rise to four different stability definitions, namely weak, strong, super, and ultra stability, and in the direction of consistent preferences, which is a naturally restricted preference domain [26].

The derived research results appear to be more diverse when it comes to the 3DSM-CYC problem variant. Firstly, two stability notions have been investigated: weak and strong. 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. Biró and McDermid [5] 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.

For this setting, Boros et al. [6] proved that each 3DSM-CYC instance admits a weakly stable matching for $n \leq 3$, where n is the size of each vertex set in the tripartition. Eriksson et al. [15] later extended this result to $n \leq 4$. Additionally, Pashkovich and Poirrier [41] further proved that not only one, but at least two stable matchings exist for each instance with $n = 5$. By this time, the conjecture on the guaranteed existence of a weakly stable matching in 3DSM-CYC with complete lists became one of the most riveting open questions in the matching under preferences literature [31, 33, 50]. Surprisingly, Lam and Plaxton [32] recently disproved this conjecture by showing that weakly stable matchings for 3DSM-CYC need not exist for an arbitrary n , moreover, it is NP-complete to determine whether a given 3DSM-CYC instance with complete lists admits a weakly stable matching.

Application-oriented research has focused on the so-called “3-sided matching with cyclic and size preferences” problem, defined by Cui and Jia [11]. They 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. The characterising feature of this variant is that a triple might contain more than one user, as servers aim at maximizing the number of users assigned to them. This feature clearly differentiates the problem from the classic 3DSM-CYC setting. Building upon this work, Panchal and Sharma [40] provided a distributed algorithm that finds a stable solution. Raveendran et al. [43] 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.

1.2 Constraint Programming approaches for finding stable matchings

Gent et al. [22] were the first to propose Constraint Programming (CP) models for the classic stable marriage problem. They showed that it is possible to obtain man-optimal and woman-optimal stable matchings immediately from the solution by enforcing Arc Consistency (AC). Later, Unsworth and Prosser [48, 49] presented a binary constraint for the the same problem and showed that their encoding is better in terms of space and time when compared to Gent et al.’s approach. They also investigated sex-equal stable matchings in their studies.

The next milestone was reached by Manlove et al. [34], who proposed three CP models for the Hospital / Residents problem (HR), which is the many-to-one generalisation of the stable marriage problem. They also explored side constraints for their models such as the case with forbidden pairs, residents who may form groups, or residents who may swap their hospitals. The existing research shows that CP models for the stable marriage problem with incomplete lists and for HR are tractable [22, 34]. O’Malley further explored CP models in his thesis for the stable marriage problem, and presented four constraint models [39]. Later on, Siala and O’Sullivan [45] improved the cloned model of Manlove et al. [34] by using a global constraint that achieves Bound Consistency in linear time.

In 2012 Eirinakis et al. [14] used the poset graph of rotations to enumerate all solutions of HR, and presented an improved version to the direct CP model of Manlove et al. [34]. Subsequently, Siala and O’Sullivan [46] used the rotation poset to model stable matchings as SAT formulation for all three types of problems: one-to-one, one-to-many, and many-to-many. They presented empirical results for finding sex-equal stable matchings, and showed that their approach outperforms the model presented in their previous paper [45]. Additionally, Drummond et al. [13] used SAT encoding for finding stable matchings that include couples.

To the best of our knowledge, CP or SAT models for 3-dimensional stable matchings have not been studied before.

2 Preliminaries

In this section we introduce the terminology and notation for the problem variants we will study. First we formalise the 3DSM-CYC problem and define the two known stability concepts for it. Then, we define three standard fairness notions that were constructed to distinguish balanced stable solutions on bipartite and non-bipartite stable matching instances.

2.1 3-dimensional stable matching with cyclic preferences

Input and output. Formally, a 3DSM-CYC instance is defined over three disjoint sets of agents of size n , denoted by $A = \{a_1, \dots, a_n\}$, $B = \{b_1, \dots, b_n\}$, and $C = \{c_1, \dots, c_n\}$. 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 can be 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 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 already defines the triple as one already in M .

2.2 Fair stable solutions

In this paper, we translate some standard fairness notions from the classic stable marriage problem to 3DSM-CYC. In most stable matching problems, several stable solutions might be present, which gives way to choosing a fair or balanced one among them. We now review the most prevalent fairness notions in such decisions [25, 29, 33, 10].

Egalitarian stable matchings. Possibly the most natural way to define a good stable matching is captured by the notion of *egalitarian stable matchings*. Each agent's satisfaction can be measured by how high she ranks her partner in the matching. In order to gain a comprehensive measure, we sum up those ranks for all matched agents. A stable matching is called egalitarian, if it minimises this sum among all stable matchings. Finding an egalitarian stable matching in the classic stable marriage problem can be done in polynomial time [28, 24], while it is NP-hard, but 2-approximable if the underlying instance is non-bipartite [17, 18]. An egalitarian stable matching in a 3DSM-CYC instance is defined as the extreme point of the following function.

$$\min_{M \text{ is a stable matching}} \left\{ \sum_{(a_i, b_j, c_k) \in M} \text{rank}_{a_i}(b_j) + \text{rank}_{b_j}(c_k) + \text{rank}_{c_k}(a_i) \right\} \quad (1)$$

Minimum regret stable matchings. Another popular fairness notion, called *minimum regret stable matching*, intuitively maximises the satisfaction of the least satisfied person in the instance. In this context, each agent's regret is measured by how high she ranks her partner in the matching – the larger this rank is, the more regret she experiences. The regret of matching M is defined as the largest regret in the instance, i.e. the worst rank that appears in the matching. Finding a minimum regret stable matching can be done in polynomial time both in bipartite and non-bipartite instances [23, 24]. A minimum regret stable matching in a 3DSM-CYC instance is defined as the extreme point of the following function.

$$\min_{M \text{ is a stable matching}} \left\{ \max_{(a_i, b_j, c_k) \in M} \{ \text{rank}_{a_i}(b_j), \text{rank}_{b_j}(c_k), \text{rank}_{c_k}(a_i) \} \right\} \quad (2)$$

Sex-equal stable matchings. A third condition is called *sex-equality*, which aims at reaching the same satisfaction level of each agent set. The satisfaction of a set of agents is measured by summing up the satisfaction level, that is, the rank of the matching partner, of each agent in the set. In the classic stable marriage setting, a sex-equal stable matching minimises the difference between the satisfaction level of the two sets. Finding a sex-equal stable matching is NP-hard in those instances [30, 35]. Even though the notion cannot be defined for non-bipartite instances, it translates readily to 3DSM-CYC instances. The difference of satisfaction level between any two of the three agent sets can be computed exactly as in the classic stable marriage setting. Then, the sum of the three pairwise differences must be minimised. We define a sex-equal stable matching as the extreme point of the following function.

$$\min_{M \text{ is a stable matching}} \left\{ \left| \sum_{(a_i, b_j, c_k) \in M} \text{rank}_{a_i}(b_j) - \text{rank}_{b_j}(c_k) \right| + \left| \sum_{(a_i, b_j, c_k) \in M} \text{rank}_{b_j}(c_k) - \text{rank}_{c_k}(a_i) \right| + \left| \sum_{(a_i, b_j, c_k) \in M} \text{rank}_{c_k}(a_i) - \text{rank}_{a_i}(b_j) \right| \right\} \quad (3)$$

2.3 Our contribution

This paper is the first to model 3-dimensional stable matchings, specifically the 3DSM-CYC problem, including its optimisation variants using side constraints. We propose the following CP models to find a stable matching in the 3DSM-CYC problem: DIV (divided agent sets), UNI (unified agent sets), and HS (hitting set). We implement each one of the models under both weak and strong stability. For the DIV and UNI models, we investigate two kinds of domain values: one based on the unique identifiers of the agents themselves, referred as *agent-based (agents)*, and the other based on the ranks of agents in one's preference list, referred as *rank-based (ranks)*. We first use the models to find any satisfying solution to a given 3DSM-CYC instance. Subsequently, we extend all models to optimisation variants under different fairness criteria and conclude with some empirical findings.

3 Methodology

In this section we present the details of our five proposed models. For each model, we propose the mandatory matching and stability constraints. We also propose how to model the fairness constraints for different optimisation versions. Furthermore, if we identified any, we state the redundant constraints that help the models with better pruning the search space.

3.1 Agent-based DIV model

The DIV-agents model consists of $3n$ variables $X = \{x_1, \dots, x_n\}$, $Y = \{y_1, \dots, y_n\}$, and $Z = \{z_1, \dots, z_n\}$, where the domain of each variable v is set as $D(v) = \{1, \dots, n\}$. For agent-based domain values, assigning $x_i = j$ (respectively $y_i = j$, or $z_i = j$) corresponds to matching a_i to b_j (respectively b_i to c_j , or c_i to a_j). A stable matching M , if any exists, is found by using the following constraints.

- (matching) For all $1 \leq i, j, k \leq n$, we add the constraint $x_i = j \wedge y_j = k \Rightarrow z_k = i$. This is to ensure that each solution corresponds to a feasible, if not stable, matching.
- (stability) Under *weak* stability, for all $1 \leq i, j, k \leq n$, and for all i', j', k' such that a_i prefers b_j to $b_{j'}$, b_j prefers c_k to $c_{k'}$ and c_k prefers a_i to $a_{i'}$ we add the constraint $x_i \neq j' \vee y_j \neq k' \vee z_k \neq i'$. This is to ensure that there is no strongly blocking triple. When solving the problem under *strong* stability, the condition to post the constraint becomes: a_i prefers b_j to $b_{j'}$ or $j' = j$, and b_j prefers c_k to $c_{k'}$ or $k' = k$, and c_k prefers a_i to $a_{i'}$ or $i' = i$, and $i' \neq i \vee j' \neq j \vee k' \neq k$. Here, as well as in the other models, the difference between weak and strong stability constraints is that the latter also cover the case when exactly two agents of a potential blocking triple are matched together.
- (redundancy) For all $1 \leq i, j, k \leq n$, we add the constraint $y_j = k \wedge z_k = i \Rightarrow x_i = j$.
- (redundancy) For all $1 \leq i, j, k \leq n$, we add the constraint $z_k = i \wedge x_i = j \Rightarrow y_j = k$.
- (redundancy) We add ALLDIFFERENT(X) and ALLDIFFERENT(Y) and ALLDIFFERENT(Z) to ensure each agent has exactly one partner from each set.
- (optimisation) When solving a fair version of the problem, we add a constraint to minimise the objective in one of the following ways, depending on which notion of fairness is desired:
 - For egalitarian M , we model Eqn. 1 as: $\sum(\text{rank}_{a_i}(b_j) + \text{rank}_{b_j}(c_k) + \text{rank}_{c_k}(a_i))$ for all i, j, k such that $x_i = j \wedge y_j = k \wedge z_k = i$.
 - For minimum regret M , we model Eqn. 2 as: $\max(\max(\text{rank}_{a_i}(b_j), \text{rank}_{b_j}(c_k), \text{rank}_{c_k}(a_i)))$ for all i, j, k such that $x_i = j \wedge y_j = k \wedge z_k = i$.

- For sex-equal M , we model Eqn. 3 as: $|S_A - S_B| + |S_B - S_C| + |S_C - S_A|$ where $S_A = \sum(\text{rank}_{a_i}(b_j))$ for all i, j such that $x_i = j$, $S_B = \sum(\text{rank}_{b_j}(c_k))$ for all j, k such that $y_j = k$, and $S_C = \sum(\text{rank}_{c_k}(a_i))$ for all k, i such that $z_k = i$.

3.2 Rank-based DIV model

Variables and domains are the same in the rank-based DIV model (DIV-ranks) as they are in the agent-based DIV model (DIV-agents), but this time assigning $x_i = j$ (respectively $y_i = j$, or $z_i = j$) corresponds to matching a_i to her j^{th} preferred agent (respectively b_i to her j^{th} preferred agent, or c_i to her j^{th} preferred agent), who might be different from b_j . A stable matching M , if any exists, is found by using the following constraints.

- (matching) For all $1 \leq i, j, k \leq n$, we add the constraint $x_i = \text{rank}_{a_i}(b_j) \wedge y_j = \text{rank}_{b_j}(c_k) \Rightarrow z_k = \text{rank}_{c_k}(a_i)$. This is to ensure that each solution corresponds to a feasible, if not stable, matching.
- (stability) Under *weak* stability, for all $1 \leq i, j, k \leq n$, we add the constraint $x_i \leq \text{rank}_{a_i}(b_j) \vee y_j \leq \text{rank}_{b_j}(c_k) \vee z_k \leq \text{rank}_{c_k}(a_i)$. This is to ensure that there is no strongly blocking triple. When solving the problem under *strong* stability, the inequalities are strict but the following part is added to each disjunction: $\vee(x_i = \text{rank}_{a_i}(b_j) \wedge y_j = \text{rank}_{b_j}(c_k) \wedge z_k = \text{rank}_{c_k}(a_i))$.
- (redundancy) For all $1 \leq i, j, k \leq n$, we add the constraint $y_j = \text{rank}_{b_j}(c_k) \wedge z_k = \text{rank}_{c_k}(a_i) \Rightarrow x_k = \text{rank}_{a_i}(b_j)$.
- (redundancy) For all $1 \leq i, j, k \leq n$, we add the constraint $z_k = \text{rank}_{c_k}(a_i) \wedge x_i = \text{rank}_{a_i}(b_j) \Rightarrow y_j = \text{rank}_{b_j}(c_k)$.
- (optimisation) We add a constraint to minimise the objective in one of the following ways, depending on which notion of fairness is desired:
 - For egalitarian M , we model Eqn. 1 as: $\sum_{i=1}^n (x_i + y_i + z_i)$.
 - For minimum regret M , we model Eqn. 2 as: $\max(\max(x_i, y_i, z_i))$ for all $1 \leq i \leq n$.
 - For sex-equal M , we model Eqn. 3 as: $|S_A - S_B| + |S_B - S_C| + |S_C - S_A|$ where $S_A = \sum_{i=1}^n (x_i)$, $S_B = \sum_{j=1}^n (y_j)$, and $S_C = \sum_{k=1}^n (z_k)$.

Note that, as opposed to agent-based domains, there are no ALLDIFFERENT constraints in rank-based models. The reason for this is that with rank-based domains it is possible for two agents in the same agent set to be assigned the same value, for example if they both got assigned to their most preferred agent.

3.3 Agent-based UNI model

The UNI-agents model consists of n variables $X = \{x_1, \dots, x_n\}$, where the domain of each variable v is set as $D(v) = \{(1, 1), \dots, (1, n), (2, 1), \dots, (n, n)\}$. Each tuple domain variable is implemented as an integer domain variable by representing the tuple (j, k) with the integer $(j - 1)n + k$. For agent-based domain values, assigning (j, k) to x_i corresponds to having the triple (a_i, b_j, c_k) in the matching. A stable matching M , if any exists, is found by using the following constraints.

In both the current and following subsections, we denote by $x_{i,B}$ and $x_{i,C}$ respectively the first and second elements of the pair assigned to x_i .

- (matching) For all $1 \leq i < i' \leq n$, we add the constraint $x_{i,B} \neq x_{i',B} \wedge x_{i,C} \neq x_{i',C}$. This is to ensure that each solution corresponds to a feasible, if not stable, matching.

- (stability) Under *weak* stability, for all $1 \leq i, j, k \leq n$, for all $1 \leq i', i'', j', k' \leq n$ such that a_i prefers b_j to $b_{j'}$, b_j prefers c_k to $c_{k'}$ and c_k prefers a_i to $a_{i'}$, we add the constraint $(x_{i,B} \neq j') \vee (x_{i'',C} \neq (j, k')) \vee (x_{i',C} \neq k)$. This is to ensure that no triple is blocking. Because in UNI only the agents from A have their own associated variables, determining whether b_j was assigned to $c_{k'}$ requires checking for each agent $a_{i''}$ from A whether both b_j and $c_{k'}$ were assigned to $a_{i''}$. This is the reason for the additional index i'' . When solving the problem under *strong* stability, the condition to post the constraint becomes: a_i prefers b_j to $b_{j'}$ or $j' = j$, and b_j prefers c_k to $c_{k'}$ or $k' = k$, and c_k prefers a_i to $a_{i'}$ or $i' = i$, and $i' \neq i \vee j' \neq j \vee k' \neq k$.
- (redundancy) We impose the constraint $\text{ALLDIFFERENT}(X)$.
- (redundancy) Denote by $F(i)$ the set of tuples that have an agent i as their first element, and $S(i)$ the tuples that have i as their second. Then, for all agents $\forall i \in n$ we have $\sum_{j \in F(i)} \text{COUNT}(j, X) = 1$ and $\sum_{j \in S(i)} \text{COUNT}(j, X) = 1$.
- (optimisation) We add a constraint to minimise the objective in one of the following ways, depending on which notion of fairness is desired:
 - For egalitarian M , we model Eqn. 1 as: $\sum_{i=1}^n (\text{rank}_{a_i}(b_{x_{i,B}}) + \text{rank}_{b_{x_{i,B}}}(c_{x_{i,C}}) + \text{rank}_{c_{x_{i,C}}}(a_i))$.
 - For minimum regret M , we model Eqn. 2 as: $\max(\max(\text{rank}_{a_i}(b_{x_{i,B}}), \text{rank}_{b_{x_{i,B}}}(c_{x_{i,C}}), \text{rank}_{c_{x_{i,C}}}(a_i)))$ for all $1 \leq i \leq n$.
 - For sex-equal M , we model Eqn. 3 as: $|S_A - S_B| + |S_B - S_C| + |S_C - S_A|$ where $S_A = \sum_{i=1}^n (\text{rank}_{a_i}(b_{x_{i,B}}))$, $S_B = \sum_{i=1}^n (\text{rank}_{b_{x_{i,B}}}(c_{x_{i,C}}))$, and $S_C = \sum_{i=1}^n (\text{rank}_{c_{x_{i,C}}}(a_i))$.

3.4 Rank-based UNI model

Variables and domains are implemented the same in the rank-based UNI model (UNI-ranks) as they are in the agent-based UNI model (UNI-agents), but this time assigning (j, k) to x_i corresponds to matching a_i to her j^{th} preferred agent from B , and matching the latter to her k^{th} preferred agent from C . A stable matching M , if any exists, is found by using the following constraints.

- (matching) For all $1 \leq i < i' \leq n$, we add the constraint $\text{pref}_{a_i}(x_{i,B}) \neq \text{pref}_{a_{i'}}(x_{i',B}) \wedge \text{pref}_{\text{pref}_{a_i}(x_{i,B})}(x_{i,C}) \neq \text{pref}_{\text{pref}_{a_{i'}}(x_{i',B})}(x_{i',C})$, where $\text{pref}_{a_i}(r)$ (respectively $\text{pref}_{b_j}(r)$, or $\text{pref}_{c_k}(r)$) represents the agent $b \in B$ (respectively $c \in C$, or $a \in A$) such that $\text{rank}_{a_i}(b) = r$ (respectively $\text{rank}_{b_j}(c) = r$, or $\text{rank}_{c_k}(a) = r$). This is to ensure that each solution corresponds to a feasible, if not stable, matching.
- (stability) Under *weak* stability for all $1 \leq i, j, k \leq n$, for all $1 \leq i', i'', j'' \leq n$ such that c_k strictly prefers a_i to $a_{i'}$, we add the constraint $(x_{i,B} \leq \text{rank}_{a_i}(b_j)) \vee (x_{i'',B} \neq \text{rank}_{a_{i''}}(b_j)) \vee (x_{i'',C} \leq \text{rank}_{b_j}(c_k)) \vee (x_{i',C} \neq (\text{rank}_{a_{i'}}(b_{j''}), \text{rank}_{b_{j''}}(c_k)))$. This is to ensure that no triple is blocking. When solving the problem under *strong* stability, c_k 's preference of a_i to $a_{i'}$ is not strict (i' can be equal to i) but the two inequalities are, and to each disjunction is added the following part: $\vee (x_i = (\text{rank}_{a_i}(b_j), \text{rank}_{b_j}(c_k)))$.
- (optimisation) We add a constraint to minimise the objective in one of the following ways, depending on which notion of fairness is desired:
 - For egalitarian M , we model Eqn. 1 as: $\sum_{i=1}^n (x_{i,B} + x_{i,C} + \text{rank}_{\text{pref}_{\text{pref}_{a_i}(x_{i,B})}(x_{i,C})}(a_i))$.
 - For minimum regret M , we model Eqn. 2 as: $\max(\max(x_{i,B}, x_{i,C}, \text{rank}_{\text{pref}_{\text{pref}_{a_i}(x_{i,B})}(x_{i,C})}(a_i)))$ for all $1 \leq i \leq n$.
 - For sex-equal M , we model Eqn. 3 as: $|S_A - S_B| + |S_B - S_C| + |S_C - S_A|$ where $S_A = \sum_{i=1}^n (x_{i,B})$, $S_B = \sum_{i=1}^n (x_{i,C})$, and $S_C = \sum_{i=1}^n (\text{rank}_{\text{pref}_{\text{pref}_{a_i}(x_{i,B})}(x_{i,C})}(a_i))$.

3.5 HS model

In the HS model, let T be the set of all possible triples as $\{(1, 1, 1), (1, 1, 2), \dots, (n, n, n)\}$. Without loss of generality, assume that the triples in T are ordered, so $t_i \in T$ refers to the i^{th} triple of T . 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. Then, finding a stable matching is equivalent to finding a hitting set of the non-blocking triples in T .

- Let M be a set variable whose upper bound is $\{i : t_i \in T\}$.
- (matching) Ensure that each agent from each set is matched by having:
 - $\forall a \in A : \sum_{t_i \in T: a \in t_i} (i \in M) = 1;$
 - $\forall b \in B : \sum_{t_i \in T: b \in t_i} (i \in M) = 1;$
 - $\forall c \in C : \sum_{t_i \in T: c \in t_i} (i \in M) = 1.$
- (stability) The stable matching is a hitting set of the non-blocking triples: $\forall t_j \in T : M \cap \{i : t_i \in BT(t_j)\} \neq \emptyset$. The type of stability is addressed in the computation of the BT sets. The model as such is not concerned with this aspect.

In this model, M is constrained to be a set of triples representing the stable matching as defined in Section 2.2, so egalitarian M , minimum regret M , and sex-equal M are defined as in Equations 1, 2, and 3 respectively.

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 tuple in the mapping.

4 Experiments

We performed our experiments on machines with Intel(R) Xeon(R) CPU with 2.40GHz running on Ubuntu 18.04. Our initial experiments on small instances comparing all models are performed using Gecode 6.3.0 [21]. Then, we conduct further experiments by using our best performing models for larger instances on a constraint solver based on lazy-clause generation, namely Chuffed 0.10.4. [9]. For DIV and UNI models, instances were first processed by MiniZinc 2.5.5 [37] before being given to the solvers. The HS model has been directly encoded using Gecode 6.2.0. In Section 4.1 we describe the datasets in use. Then, in Sections 4.2 and 4.3 we compare the proposed models.

4.1 Dataset description

For every size n present in our experiments, we generated 100 instances with n agents in each agent set and a complete list for each agent. Half of these instances are random and the other half have some or all of the preferences 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 real-life applications of master lists occur in resident matching programs [4], dormitory room assignments [42], cooperative download applications such as BitTorrent [1], and 3-sided networking services [11]. The detailed distribution of the 100 instances generated for each size is as follows:

- **Random:** 50 random instances from uniform distribution.
- **ML_oneset:** 20 instances where the preference lists of the agents in one of the agent sets are based on master lists, and the preference lists of the agents in the other two agent sets are random.

- **ML_1swap**: 15 instances, where each agent set has a randomly chosen master list that all agents in the set follow. Then, we randomly choose two agents from each agent's preference list, and swap their positions.
- **ML_2swaps**: 15 instances, where each agent set has a randomly chosen master list that all agents in the set follow. First, we randomly choose two agents from each agent's preference list, and swap their positions. Subsequently, we randomly choose two more agents from each list such that the new agents were not involved in the first swap, then we swap their positions.

Note that neither the type of stability (weak or strong) nor the fairness objective is part of the preferences themselves. For this reason, the 100 instances generated for each size n are used for both types of stability and for all satisfiability and optimisation versions.

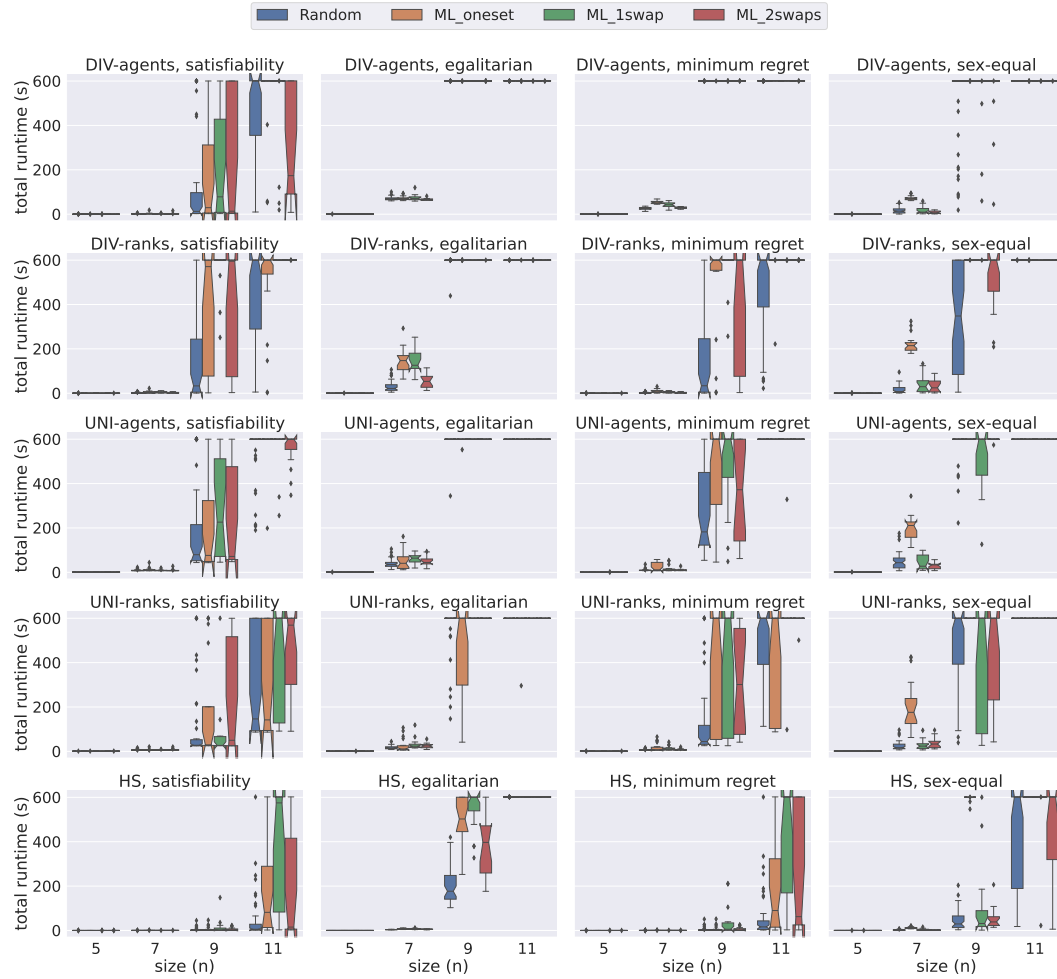
We did not consider instances where all preference lists in all three sets are exact master lists, because the complete set of stable matchings for these instances is known [16]. ML_oneset instances always have a strongly stable matching (Lemma 1), but, contrary to the case with master lists in all three sets, not all solutions have been characterised. Therefore there is value in modelling fairness versions of the problem for instances with this structure.

4.2 Model comparison

In this section, we first test each one of our five models (i.e. HS, DIV-agents, DIV-ranks, UNI-agents, and UNI-ranks) using different heuristic search strategies on small instances in Gecode to find out which strategy performs best. In Gecode experiments, we did not use extra propagation techniques such as lazy clause generation or restarts. Considering that the HS model is implemented in Gecode, and not all search strategies are common to both Gecode and MiniZinc, for a fair comparison between all five models we used `indomain_min` (assigning the smallest value in the domain) and `indomain_max` (assigning the largest value in the domain) strategies combined with a search on variables in the given order. The former is referred as *nonemin*, and the latter as *nonemax*. Additionally, we further tested the DIV and UNI models using alternative built-in search strategies that exist in MiniZinc, notably `indomain_split`, a heuristic that bisects a variable's domain then tries the lower half before trying the upper one. We observed that a strategy that is based on choosing the variable with the smallest domain size using `indomain_min` results in the best performance for DIV model, while using `indomain_split` instead of `indomain_min` leads to the best performance for UNI model. We refer to these as *failmin* and *failsplit* strategies, respectively. Therefore, all the remaining results and plots were obtained by running HS with *nonemax* strategy, DIV-agents and DIV-ranks with *failmin*, and UNI-agents and UNI-ranks with *failsplit*.

During the experiments, we used a time-limit of 10 minutes for each instance. Considering the huge number of all combinations of different parameters in each model, we adapted a look-ahead approach for our tests, i.e. we started performing tests on all models using small instances $n = 4$. Then, we incremented the n for each model that has the potential to be the best. If a model times out on most of the instances for a given combination of parameters, we do not test it further on these instances.

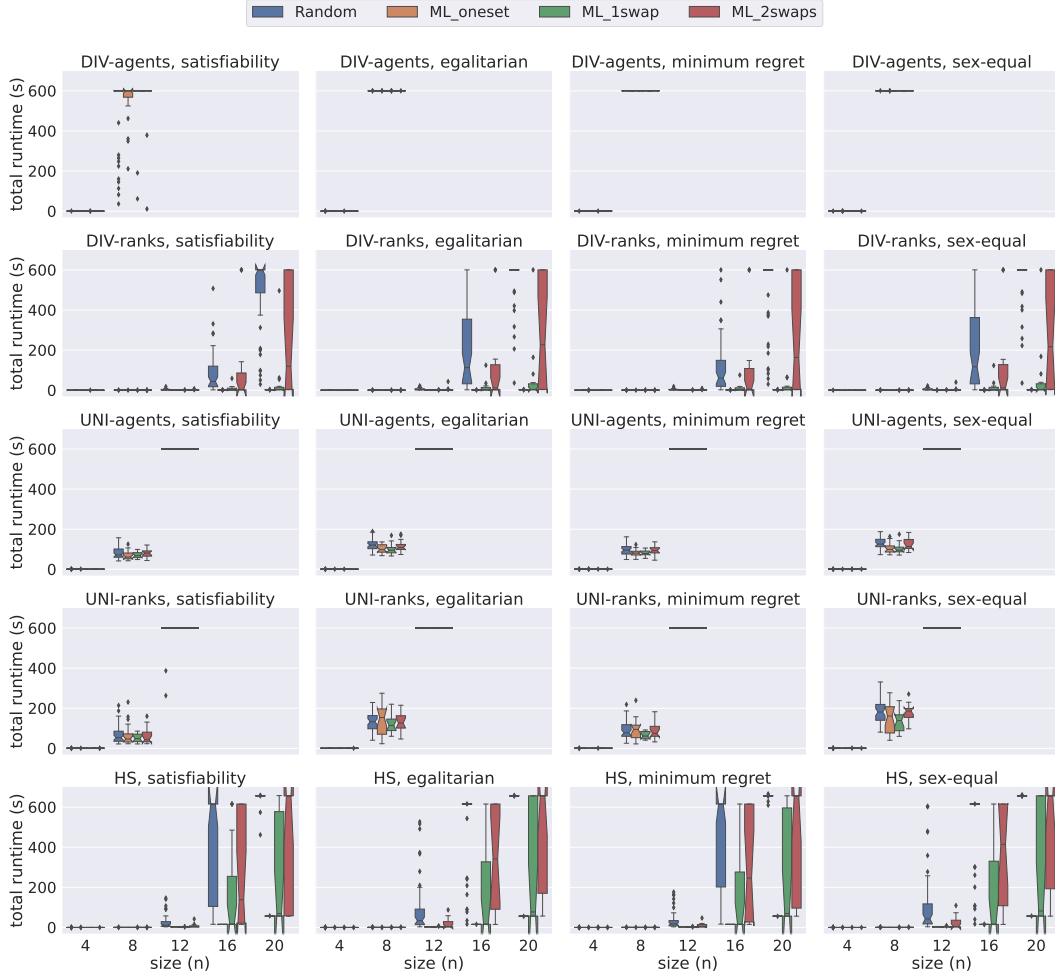
We use notched boxplots [36] in Figures 1, 2, 3, and 4. Figure 1 presents a comparison of total time required by all five models on instances of size $5 \leq n \leq 11$ under weak stability solved using Gecode. The first insight gained is that the HS model handles the instances of small sizes very well. When we examine performance based on the dataset generation methods, we observe that usually a weakly stable matching for the instances in Random is found faster than other datasets when using Gecode. Additionally we observe that the



■ **Figure 1** A comparison of total time spent by all models under weak stability using Gecode.

models require more time to solve the instances in ML_1swap for satisfiability, egalitarian, and minimum regret versions. On the other hand, for the sex-equal version of the problem, ML_oneset is the most challenging dataset for all models. We conclude from this figure that HS is the best model when dealing with small instances under weak stability using Gecode.

Figure 2 demonstrates the results obtained from the same datasets under strong stability. Note that the problem model for finding a matching under weak stability is a relaxed version of the model under strong stability. However, it is interesting to observe from the experiments that, on average, strongly stable matchings are found faster than their weak counterpart. We can clearly observe this behaviour on Figure 2. All models except DIV-agents were able to solve all satisfiable instances of size between $4 \leq n \leq 11$ within the given time limit. Therefore, in order to provide more insight into the performance of models, we use a larger scale, i.e. $\{4, 8, 12, 16, 20\}$ in Figure 2. Under strong stability, we clearly observe that HS and DIV-ranks scale better when compared to the other models. For instance, for the satisfiability problem with size $n = 8$, both HS and DIV-ranks quickly solve all the instances. However, both UNI-agents and UNI-ranks require longer time than HS and DIV-ranks, whereas DIV-agents fails to solve many instances within the given time-limit. Both UNI and HS follow the same commitment approach (group commitment). We believe this is hampering their



■ **Figure 2** A comparison of total time spent by all models under strong stability using Gecode.

scalability as there are fewer solutions in the strong stability case, which increases the chances of making wrong choices thus leading to higher penalties in the case of group commitment as we have to undo the three pairings. Considering that the time performance of UNI models and DIV-agents are considerably worse for $4 \leq n \leq 12$ when compared to others, we decided to discard them from further experiments. HS is not performing that bad, but its scalability is affected by the computation of the BT sets. The size of each BT set is $O(n^3)$, which is a remarkable overhead when dealing with big instances. We elaborate more on this limitation in the next section. Therefore, we conclude from this figure that DIV-ranks is the most efficient model when working with large n under strong stability using Gecode.

Note that HS was implemented in Gecode directly because it is cheaper to carry out the computation of the BT sets in C++ than in MiniZinc. However, this decision does not put the other models in a disadvantageous position since the main contribution to the running time comes from the solving time and in all cases the solving phase is carried out in C++.

In addition to the Gecode experiments, we tested our four models DIV-agents, DIV-ranks, UNI-agents, and UNI-ranks on Chuffed. Chuffed is the state-of-the art lazy clause solver that performs propagation by recording the reasons of propagation at each step. This helps with efficiently creating nogoods during the search and avoiding failures. Note that due to

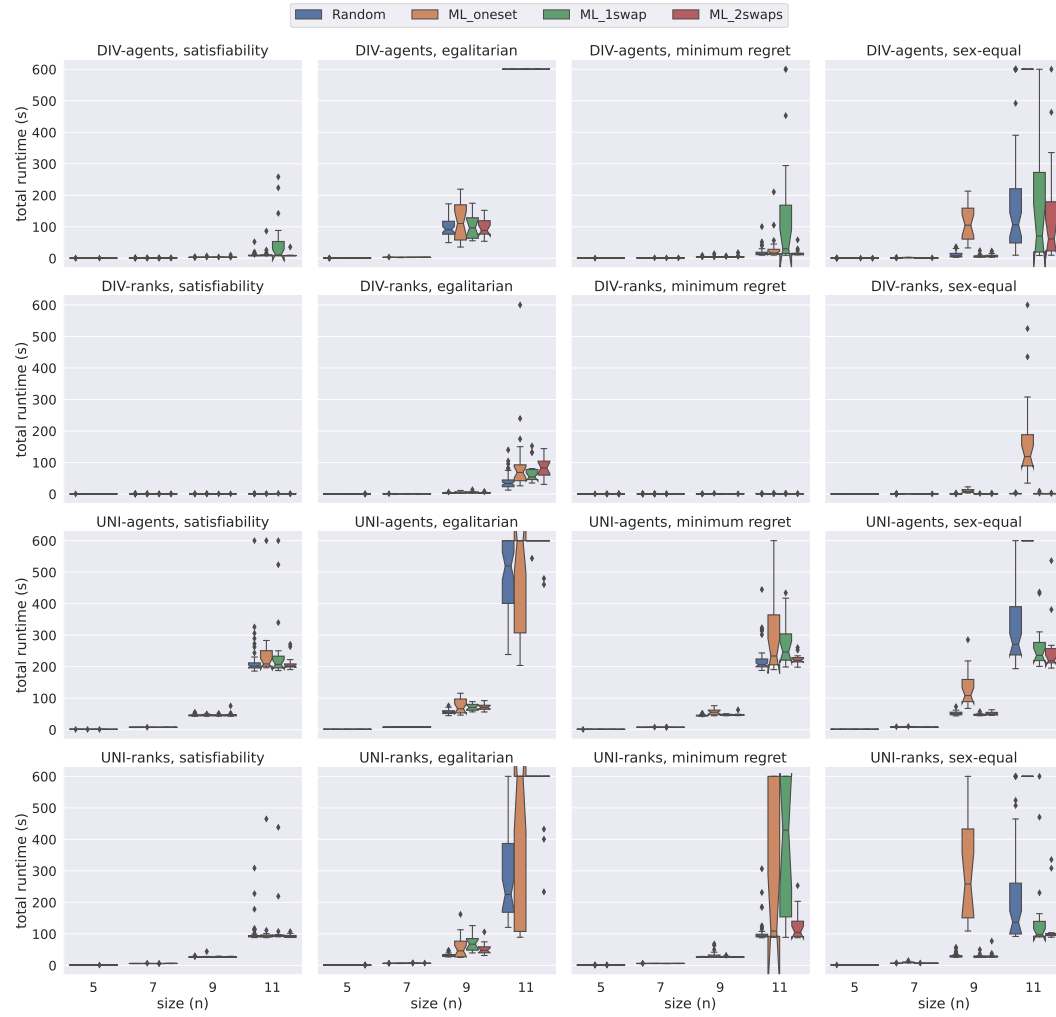


Figure 3 A comparison of total time spent by all models except HS under weak stability using Chuffed.

implementing the HS model directly in Gecode, our Chuffed experiment results do not include the performance of the HS model. Figure 3 presents a comparison of DIV-agents, DIV-ranks, UNI-agents, and UNI-ranks models on instances of size $5 \leq n \leq 11$ under weak stability. In these plots, we can clearly observe that DIV-ranks model has an advantage over the others in terms of total time required to complete the experiments. It is interesting to observe that contrasting with the findings of Gecode experiments in Figure 1, where DIV-agents seems to have an analogous performance with DIV-ranks, if not better, we observe in Figure 3 that DIV-ranks has a clear advantage over DIV-agents when using Chuffed. We believe this shows that DIV-ranks benefits greatly from nogood learning. Additionally, using Figure 3, we can verify our previous observation about ML_oneset being a more challenging dataset generation method for the sex-equal variant under weak stability.

Lastly, Figure 4 demonstrates a comparison of DIV-agents, DIV-ranks, UNI-agents, and UNI-ranks models on instances of size $5 \leq n \leq 11$ under strong stability. A very straightforward intuition of these tests is that the UNI-agents and UNI-ranks models are not able to scale well to larger instances. On the other hand, we observe that DIV-agents handles an increase in



■ **Figure 4** A comparison of total time spent by all models except HS under strong stability using Chuffed.

the number of agents better than the UNI models, but it still performs worse than DIV-ranks when $n \geq 9$. Considering the stable and rapid performance of DIV-ranks using Chuffed and also combining this with our observation on Figure 2, we conclude that the DIV-ranks model is the best one to solve 3DSM-CYC under strong stability.

4.3 Scalability

Considering DIV-ranks is the best performing model in the majority of cases, we performed further experiments using this model on instances with $n \in \{20, 23, 26, 29, 32, 35, 40, 45, 50, 60, 70, 80, 90, 100, 110, 120, 130\}$.

Figure 5 presents a comparison of the median total time required by DIV-ranks using failmin strategy on all four datasets both under weak and strong stability. An interesting insight from this figure is that all four problem variants (i.e. satisfiability, egalitarian, minimum regret, and sex-equal) result in similar performances under strong stability, where instances in Random require the longest time to be solved and ML_oneset requires the least. However,

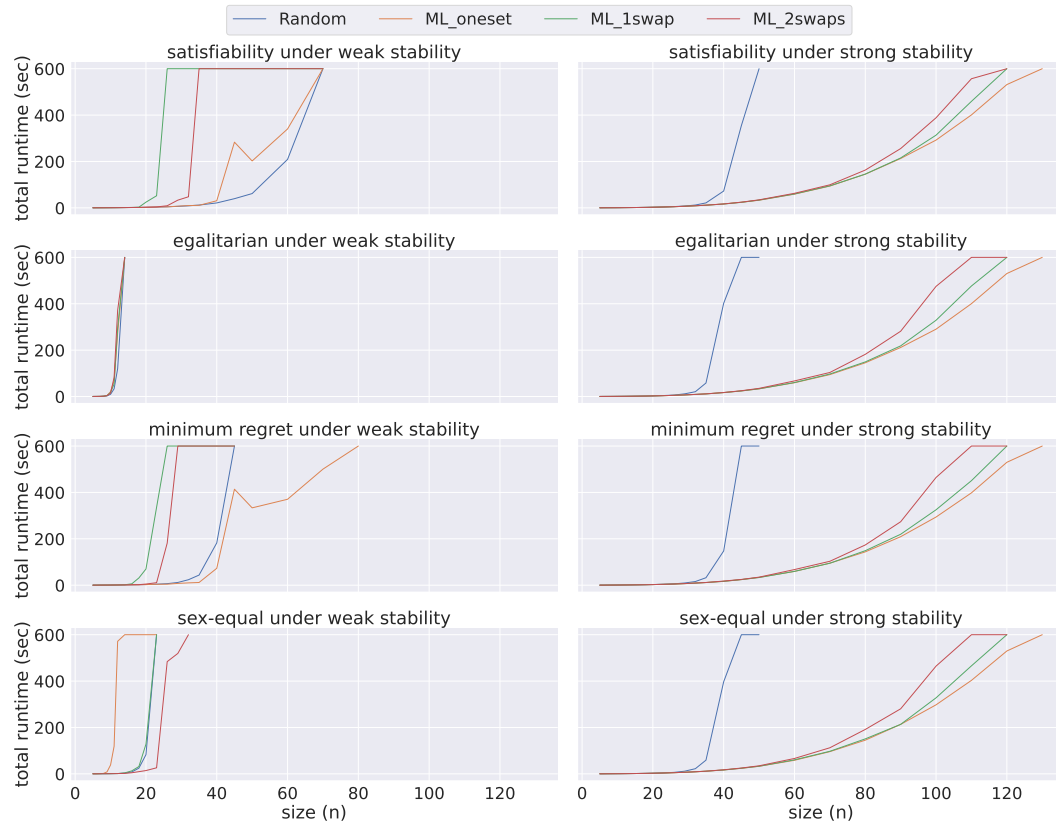


Figure 5 An overview of the performance of DIV-ranks using Chuffed under both weak and strong stability when solving problem instances of different sizes for each dataset.

we cannot make such a generalisation for weakly stable matchings. For instance, ML_oneset dataset is the most challenging dataset for the sex-equal problem variant, but it is also the least challenging for the minimum regret variant under weak stability.

5 Conclusion and future work

We proposed a collection of Constraint Programming models to solve the 3-dimensional stable matching problem with cyclic preferences (3DSM-CYC) using both strong and weak stability notions. Additionally, we extended some well-known fairness notions (egalitarian, minimum regret, and sex-equal) to 3DSM-CYC. The five proposed models are fundamentally different from each other in terms of their commitment (individual or group), and also their domain values (agents or ranks). Our experiments show that nogood learning benefits some models more than others. An unexpected observation is that strong stability turns out to be easier to solve than weak stability. Following a comprehensive empirical evaluation, we conclude that the performances of the proposed models differ with respect to the type of stability and dataset generation method.

The models proposed can be easily adapted to take advantage of the good performance of strong stability by first trying to find a strongly stable matching. Other future work could extend our models to more types of instances, for example by allowing preference lists to be incomplete. One could also look at other redundant constraints in order to best take

advantage of the properties that some instances exhibit with regard to fairness objectives. We remarked that HS pays a high price for the generation of the BT sets but it is possible to generate the sets by demand instead of doing it eagerly.

References

- 1 D. J. Abraham, A. Levavi, D. F. Manlove, and G. O'Malley. The stable roommates problem with globally-ranked pairs. *Internet Mathematics*, 5:493–515, 2008.
- 2 Ahmet Alkan. Nonexistence of stable threesome matchings. *Mathematical Social Sciences*, 16(2):207–209, 1988. URL: <https://www.sciencedirect.com/science/article/pii/0165489688900534>.
- 3 Michel Balinski and Tayfun Sönmez. A tale of two mechanisms: Student placement. *Journal of Economic Theory*, 84(1):73–94, 1999. URL: <https://www.sciencedirect.com/science/article/pii/S0022053198924693>.
- 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 and Eric McDermid. Three-sided stable matchings with cyclic preferences. *Algorithmica*, 58(1):5–18, 2010. doi:10.1007/s00453-009-9315-2.
- 6 Endre Boros, Vladimir Gurvich, Steven Jaslar, and Daniel Krasner. Stable matchings in three-sided systems with cyclic preferences. *Discret. Math.*, 289(1-3):1–10, 2004. doi:10.1016/j.disc.2004.08.012.
- 7 Sebastian Braun, Nadja Dwenger, and Dorothea Kübler. Telling the truth may not pay off: An empirical study of centralized university admissions in Germany. *The B.E. Journal of Economic Analysis and Policy*, 10, article 22, 2010.
- 8 Yan Chen and Tayfun Sönmez. Improving efficiency of on-campus housing: an experimental study. *American Economic Review*, 92:1669–1686, 2002.
- 9 Geoffrey Chu. *Improving combinatorial optimization*. PhD thesis, University of Melbourne, Australia, 2011. URL: <http://hdl.handle.net/11343/36679>.
- 10 Sofie De Clercq, Steven Schockaert, Martine De Cock, and Ann Nowé. Solving stable matching problems using answer set programming. *Theory Pract. Log. Program.*, 16(3):247–268, 2016. doi:10.1017/S147106841600003X.
- 11 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.
- 12 Vladimir I. Danilov. Existence of stable matchings in some three-sided systems. *Math. Soc. Sci.*, 46(2):145–148, 2003. doi:10.1016/S0165-4896(03)00073-8.
- 13 Joanna Drummond, Andrew Perrault, and Fahiem Bacchus. SAT is an effective and complete method for solving stable matching problems with couples. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 518–525. AAAI Press, 2015. URL: <http://ijcai.org/Abstract/15/079>.
- 14 Pavlos Eirinakis, Dimitris Magos, Ioannis Mourtos, and Panayiotis Miliotis. Finding all stable pairs and solutions to the many-to-many stable matching problem. *INFORMS J. Comput.*, 24(2):245–259, 2012. doi:10.1287/ijoc.1110.0449.
- 15 Kimmo Eriksson, Jonas Sjöstrand, and Pontus Strimling. Three-dimensional stable matching with cyclic preferences. *Math. Soc. Sci.*, 52(1):77–87, 2006. doi:10.1016/j.mathsocsci.2006.03.005.
- 16 Guillaume Escamocher and Barry O'Sullivan. Three-dimensional matching instances are rich in stable matchings. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26-29, 2018, Proceedings*, volume 10848, pages 182–197. Springer, 2018. doi:10.1007/978-3-319-93031-2_13.

- 17 Tomás Feder. A new fixed point approach for stable networks and stable marriages. *J. Comput. Syst. Sci.*, 45(2):233–284, 1992. doi:10.1016/0022-0000(92)90048-N.
- 18 Tomás Feder. Network flow and 2-satisfiability. *Algorithmica*, 11(3):291–319, 1994. doi:10.1007/BF01240738.
- 19 Anh-Tuan Gai, Dmitry Lebedev, Fabien Mathieu, Fabien de Montgolfier, Julien Reynier, and Laurent Viennot. Acyclic preference systems in P2P networks. In *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007, Proceedings*, volume 4641, pages 825–834. Springer, 2007. doi:10.1007/978-3-540-74466-5_88.
- 20 D. Gale and L. S. Shapley. College admissions and the stability of marriage. *Am. Math. Mon.*, 120(5):386–391, 2013. doi:10.4169/amer.math.monthly.120.05.386.
- 21 Gecode Team. Gecode: Generic constraint development environment, 2019. Available from <http://www.gecode.org>.
- 22 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.
- 23 Dan Gusfield. Three fast algorithms for four problems in stable marriage. *SIAM J. Comput.*, 16(1):111–128, 1987. doi:10.1137/0216010.
- 24 Dan Gusfield and Robert W. Irving. *The Stable marriage problem - structure and algorithms*. MIT Press, 1989.
- 25 Magnús M. Halldórsson, Robert W. Irving, Kazuo Iwama, David F. Manlove, Shuichi Miyazaki, Yasufumi Morita, and Sandy Scott. Approximability results for stable marriage problems with ties. *Theor. Comput. Sci.*, 306(1-3):431–447, 2003. doi:10.1016/S0304-3975(03)00321-9.
- 26 Chien-Chung Huang. Two's company, three's a crowd: Stable family and threesome roommates problems. In *Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings*, volume 4698, pages 558–569. Springer, 2007. doi:10.1007/978-3-540-75520-3_50.
- 27 Robert W. Irving. An efficient algorithm for the "stable roommates" problem. *J. Algorithms*, 6(4):577–595, 1985. doi:10.1016/0196-6774(85)90033-1.
- 28 Robert W. Irving, Paul Leather, and Dan Gusfield. An efficient algorithm for the "optimal" stable marriage. *J. ACM*, 34(3):532–543, 1987. doi:10.1145/28869.28871.
- 29 Kazuo Iwama and Shuichi Miyazaki. A survey of the stable marriage problem and its variants. In *International Conference on Informatics Education and Research for Knowledge-Circulating Society (ICKS 2008)*, pages 131–136, 2008.
- 30 Akiko Kato. Complexity of the sex-equal stable marriage problem. *Japan Journal of Industrial and Applied Mathematics*, 10:1–19, 1993.
- 31 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.
- 32 Chi-Kit Lam and C. Gregory Plaxton. On the existence of three-dimensional stable matchings with cyclic preferences. In *Algorithmic Game Theory - 12th International Symposium, SAGT 2019, Athens, Greece, September 30 - October 3, 2019, Proceedings*, volume 11801, pages 329–342. Springer, 2019. doi:10.1007/978-3-030-30473-7_22.
- 33 David F. Manlove. *Algorithmics of Matching Under Preferences*, volume 2. WorldScientific, 2013. doi:10.1142/8591.
- 34 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.

- 35 Eric McDermid and Robert W Irving. Sex-equal stable matchings: Complexity and exact algorithms. *Algorithmica*, 68(3):545–570, 2014.
- 36 Robert McGill, John W. Tukey, and Wayne A. Larsen. Variations of box plots. *The American Statistician*, 32(1):12–16, 1978. doi:10.1080/00031305.1978.10479236.
- 37 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.
- 38 Cheng Ng and Daniel S. Hirschberg. Three-dimensional stable matching problems. *SIAM J. Discrete Math.*, 4(2):245–252, 1991. doi:10.1137/0404023.
- 39 Gregg O'Malley. *Algorithmic aspects of stable matching problems*. PhD thesis, University of Glasgow, UK, 2007. URL: <http://theses.gla.ac.uk/64/>.
- 40 Nikita Panchal and Seema Sharma. An efficient algorithm for three dimensional cyclic stable matching. *International Journal of Engineering Research and Technology*, 3(4), 2014.
- 41 Kanstantsin Pashkovich and Laurent Poirrier. Three-dimensional stable matching with cyclic preferences. *Optim. Lett.*, 14(8):2615–2623, 2020. doi:10.1007/s11590-020-01557-4.
- 42 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. *Int. J. Game Theory*, 36(3-4):519–535, 2008. doi:10.1007/s00182-007-0083-4.
- 43 Neetu Raveendran, Yiyong Zha, Yunfei Zhang, Xin Liu, and Zhu Han. Virtual core network resource allocation in 5G systems using three-sided matching. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2019.
- 44 Alvin E. Roth and Marilda A. Oliveira Sotomayor. *Two-Sided Matching: A Study in Game-Theoretic Modeling and Analysis*. Cambridge University Press, 1990. doi:10.1017/COL052139015X.
- 45 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.
- 46 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.
- 47 Ashok Subramanian. A new approach to stable matching problems. *SIAM J. Comput.*, 23(4):671–700, 1994. doi:10.1137/S0097539789169483.
- 48 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.
- 49 Chris Unsworth and Patrick Prosser. An n-ary constraint for the stable marriage problem. *CoRR*, abs/1308.0183, 2013. arXiv:1308.0183.
- 50 Gerhard J. Woeginger. Core stability in hedonic coalition formation. In *Proceedings of SOFSEM 2013, 39th International Conference on Current Trends in Theory and Practice of Computer Science*, volume 7741, pages 33–50. Springer, 2013. doi:10.1007/978-3-642-35843-2_4.

A Tractability proof for ML_{oneset}

► **Lemma 1.** *Each 3DSM-CYC instance with complete lists and one master list admits at least one strongly stable matching.*

Proof. Assume that agent set C is equipped with a master list. For each agent $c_k \in C$, let us relabel the preferred agents of c_k from set A in her preference list so that $\text{rank}_{c_k}(a_i) < \text{rank}_{c_k}(a_{i+1})$ for each $1 \leq i \leq n-1$. We run a serial dictatorship algorithm as follows. First

a_1 chooses her first choice agent in B , who then chooses her first choice agent in C . This triple is removed from the instance. Then we iterate the same starting with a_2 , who chooses her first choice among agents in B not yet removed, and so on. Let us relabel agents in B and C so that triples (a_i, b_i, c_i) form the output matching of this algorithm.

First observe that an agent a_i can only prefer an agent b_j to her partner in M if $j < i$. Similarly, a_i prefers b_i to all agents with $j > i$. These observations hold for agents b_i and c_i as well. Each weakly blocking triple thus must include a decrease in the variable index where the strict preference occurs, and simultaneously can contain no decrease in the index elsewhere, because this would make the corresponding agent less satisfied as she was in M . ◀

Bounds on Weighted CSPs Using Constraint Propagation and Super-Reparametrizations

Tomáš Dlask¹ 

Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic

Tomáš Werner 

Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic

Simon de Givry 

Université Fédérale de Toulouse, ANITI, INRAE, UR 875, France

Abstract

We propose a framework for computing upper bounds on the optimal value of the (maximization version of) Weighted CSP (WCSP) using super-reparametrizations, which are changes of the weights that keep or increase the WCSP objective for every assignment. We show that it is in principle possible to employ arbitrary (under certain technical conditions) constraint propagation rules to improve the bound. For arc consistency in particular, the method reduces to the known Virtual AC (VAC) algorithm. Newly, we implemented the method for singleton arc consistency (SAC) and compared it to other strong local consistencies in WCSPs on a public benchmark. The results show that the bounds obtained from SAC are superior for many instance groups.

2012 ACM Subject Classification Mathematics of computing → Linear programming; Theory of computation → Linear programming; Theory of computation → Constraint and logic programming

Keywords and phrases Weighted CSP, Super-Reparametrization, Linear Programming, Constraint Propagation

Digital Object Identifier 10.4230/LIPIcs.CP.2021.23

Funding *Tomáš Dlask:* Supported by the Czech Science Foundation (grant 19-09967S) and the Grant Agency of the Czech Technical University in Prague (grant SGS19/170/OHK3/3T/13).

Tomáš Werner: Supported by the Czech Science Foundation (grant 19-09967S) and the OP VVV project CZ.02.1.01/0.0/0.0/16_019/0000765.

Simon de Givry: Supported by the “Agence nationale de la Recherche” (ANR-19-PIA3-0004 ANITI).

1 Introduction

In the *weighted constraint satisfaction problem (WCSP)* we maximize the sum of (weight) functions over many discrete variables, where each function depends only on a (usually small) subset of the variables. A popular approach to tackle this NP-hard combinatorial optimization problem is via its linear programming (LP) relaxation [21, 32, 30, 29, 20, 1]. The dual of this LP relaxation minimizes an upper bound on the WCSP optimal value over reparametrizations (also known as equivalence-preserving transformations) of the original WCSP instance. For large instances this is done only approximately, by methods based on block-coordinate descent [16, 14, 27, 28, 32, 17] or constraint propagation [5, 18, 32, 19]. Fixed points of these methods are characterized by a local consistency of the CSP formed by the active tuples (to be defined later) of the transformed WCSP [32, 16, 14, 5, 27].

¹ Corresponding author



This approach is limited in that it cannot enforce an arbitrary level of local consistency, unless new weight functions are introduced. Namely, it can achieve at most pairwise consistency [33, 34], which for binary WCSPs reduces to arc consistency (reparametrized WCSP corresponding to global optima of the dual LP relaxation have been called *optimally soft arc consistent* in [6, 5]).

In this paper, we study a different LP formulation of the WCSP, which was proposed in [17] but never pursued later. It differs from the above mentioned basic LP relaxation and does not belong to the known hierarchy of LP relaxations obtained by introducing new weight functions of higher arities [24, 33, 19, 1] (which leads to a fine-grained version of the Sherali-Adams hierarchy [22] for the WCSP). Our LP formulation again minimizes the upper bound on the WCSP optimal value, but this time over *super-reparametrizations* of the initial WCSP instance. Its remarkable feature is that it allows using almost arbitrary (up to some technical assumptions) constraint propagation techniques to improve the bound, without introducing new weight functions. On the other hand, it may neither preserve the value of the individual assignments nor the set of optimal assignments, but it nevertheless provides a valid, and possibly tighter, bound on the WCSP optimal value.

2 Notation

For clarity of presentation, we will consider only binary WCSPs with finite weights. However, it would be straightforward to generalize the approach described in the paper to WCSPs of any arity and with some weights infinite (i.e., including hard constraints).

Let V be a finite set of *variables* and D a finite *domain* of each variable. An *assignment* $x \in D^V$ assigns² a value $x_i \in D$ to each variable $i \in V$. Let $E \subseteq \binom{V}{2}$ be a set of variable pairs, so that (V, E) is an undirected graph. The *weighted constraint satisfaction problem* (WCSP) seeks to maximize the function

$$F(x|f) = \sum_{i \in V} f_i(x_i) + \sum_{\{i,j\} \in E} f_{ij}(x_i, x_j) \quad (1)$$

over all assignments $x \in D^V$. Here, $f_i: D \rightarrow \mathbb{R}$ and $f_{ij}: D^2 \rightarrow \mathbb{R}$ (where we assume that $f_{ij}(k, l) = f_{ji}(l, k)$) are *weight functions*, whose values together form a vector $f \in \mathbb{R}^T$ where

$$T = \underbrace{\{(i, k) \mid i \in V, k \in D\}}_{V \times D} \cup \{\{(i, k), (j, l)\} \mid \{i, j\} \in E, k, l \in D\} \quad (2)$$

is a set of *tuples*. For $t \in T$, we denote $f_t = f_i(k)$ if $t = (i, k) \in V \times D$, and $f_t = f_{ij}(k, l) = f_{ji}(l, k)$ if $t = \{(i, k), (j, l)\} \in T - (V \times D)$. The WCSP instance is defined by (D, V, E, f) . However, as the structure (D, V, E) will be the same throughout the paper, we will refer to WCSP instances only as f (thus, we identify WCSP instances with vectors $f \in \mathbb{R}^T$).

We say that an assignment $x \in D^V$ *uses* a tuple $t = (i, k)$ if $x_i = k$, and x *uses* a tuple $t = \{(i, k), (j, l)\}$ if $x_i = k$ and $x_j = l$. In the *constraint satisfaction problem* (CSP), we are given a set $A \subseteq T$ of *allowed tuples* (while the tuples $T - A$ are called *forbidden*) and look for an assignment x (a *solution* to the CSP) that uses only the allowed tuples, i.e., $(i, x_i) \in A$ for all $i \in V$ and $\{(i, x_i), (j, x_j)\} \in A$ for all $\{i, j\} \in E$. The CSP is *satisfiable* if it has a solution. The CSP instance is defined by (D, V, E, A) but, as (D, V, E) will be always the same, we will refer to it only as A (i.e., we identify CSP instances with subsets of T).

² As usual, D^V denotes the set of all mappings from V to D , so $x \in D^V$ is the same as $x: V \rightarrow D$.

For a tuple $t \in T$, we denote

$$U(t) = \begin{cases} \{(i, k') \mid k' \in D\} & \text{if } t = (i, k) \\ \{(i, k'), (j, l')\} \mid k', l' \in D\} & \text{if } t = \{(i, k), (j, l)\} \end{cases}$$

so that, e.g., for all $i \in V$ and $k, k' \in D$ we have $U((i, k)) = U((i, k'))$. By

$$\mathcal{U} = \{U(t) \mid t \in T\} = \{\{(i, k) \mid k \in D\} \mid i \in V\} \cup \{\{(i, k), (j, l)\} \mid k, l \in D\} \mid \{i, j\} \in E\}$$

we denote the natural partition of T into $|V| + |E|$ subsets. Clearly, any assignment uses exactly one tuple from each set $U \in \mathcal{U}$.

For a CSP $A \subseteq T$ and $t \in T$, we denote $A|_t = A - (U(t) - \{t\})$. That is, x is a solution to CSP $A|_t$ if and only if x is a solution to CSP A and x uses tuple t . E.g., in $A|_{(i,k)}$ we search for solutions x to CSP A satisfying $x_i = k$ (this is often denoted also as $A|_{x_i=k}$).

3 Bounding the WCSP Optimal Value

We define the function $B: \mathbb{R}^T \rightarrow \mathbb{R}$ by

$$B(f) = \sum_{i \in V} \max_{k \in D} f_i(k) + \sum_{\{i,j\} \in E} \max_{k,l \in D} f_{ij}(k,l) = \sum_{U \in \mathcal{U}} \max_{t \in U} f_t. \quad (3)$$

For $f \in \mathbb{R}^T$, we call a tuple $t \in T$ *active* if $f_t = \max_{t' \in U(t)} f_{t'}$. Thus, a tuple $t = (i, k) \in T$ is active if $f_i(k) = \max_{k' \in D} f_i(k')$, and a tuple $t = \{(i, k), (j, l)\} \in T$ is active if $f_{ij}(k, l) = \max_{k', l' \in D} f_{ij}(k', l')$. The set of all tuples that are active for f is denoted³ by $A^*(f) \subseteq T$.

► **Theorem 1** ([32]). *For every WCSP $f \in \mathbb{R}^T$ and every assignment $x \in D^V$ we have:*

- (a) $B(f) \geq F(x|f)$,
- (b) $B(f) = F(x|f)$ if and only if x is a solution to CSP $A^*(f)$.

Proof. (a) can be checked by comparing expressions (1) and (3) term by term.

(b) says that $B(f) = F(x|f)$ if and only if assignment x uses only the active tuples of f . This is again straightforward from (1) and (3). ◀

Theorem 1 says that $B(f)$ is an *upper bound* on the WCSP optimal value. Moreover, it shows that $B(f) = F(x|f)$ implies that x is a maximizer of the WCSP objective (1).

3.1 Minimizing the Upper Bound by Reparametrizations

If WCSPs $f, g \in \mathbb{R}^T$ satisfy $F(x|f) = F(x|g)$ for all $x \in D^V$, we say that f is a *reparametrization* of g (or *equivalent* to g or an equivalence-preserving transformation of g) [16, 30, 21, 32, 33, 20, 6, 5, 28]. As the function $F(x|f)$ is linear in f for every x , equality $F(x|f) = F(x|g)$ can be written as $F(x|f - g) = 0$. Linearity of $F(x|\cdot)$ also implies that the set $\{h \in \mathbb{R}^T \mid F(x|h) = 0 \forall x \in D^V\}$ is a subspace⁴ of \mathbb{R}^T .

³ The set of active tuples $A^*(f)$ corresponds to the notion of $\text{Bool}(f)$ in [5]. The characteristic vector of the set $A^*(f)$ was denoted \bar{f} in [27, 32], $[f]$ in [33], and $\text{mi}[f]$ in [20].

⁴ For binary WCSPs with a connected graph (V, E) , this subspace can be parametrized as $h_i(k) = \sum_{j|\{i,j\} \in E} \varphi_{ij}(k)$ and $h_{ij}(k, l) = -\varphi_{ij}(k) - \varphi_{ji}(l)$ where $\varphi_{ij}, \varphi_{ji}: D \rightarrow \mathbb{R}$, $\{i, j\} \in E$, are arbitrary unary weight functions [32, Theorem 3]. For WCSPs of any arity, see [33, §3.2].

Given a WCSP $g \in \mathbb{R}^T$, this suggests to minimize the upper bound on its optimal value $\max_x F(x|g)$ by reparametrizations:

$$\min_{f \in \mathbb{R}^T} B(f) \quad \text{subject to} \quad F(x|f) = F(x|g) \quad \forall x \in D^V. \quad (4)$$

Although this problem has an exponential number of constraints, its feasible set is an affine subspace of \mathbb{R}^T and thus the number of constraints can be reduced to polynomial. Using the well-known trick, the problem can be transformed to a linear program⁵, which is the *dual LP relaxation* of the WCSP g [21, 32, 20]. WCSPs f optimal for (4) have been called *optimally soft arc consistent (OSAC)* in [6, 5]. If any optimal solution f to (4) satisfies $B(f) = F(x|f)$ for some x , the LP relaxation is tight. Otherwise, $B(f)$ is only a strict upper bound on the optimal value of WCSP g .

3.2 Minimizing the Upper Bound by Super-Reparametrizations

If WCSPs $f, g \in \mathbb{R}^T$ satisfy $F(x|f) \geq F(x|g)$ (that is, $F(x|f - g) \geq 0$) for all $x \in D^V$, we say that f is a *super-reparametrization*⁶ of g . The set $\{h \in \mathbb{R}^T \mid F(x|h) \geq 0 \quad \forall x \in D^V\}$ is a polyhedral convex cone. Following [17], we consider the problem

$$\min_{f \in \mathbb{R}^T} B(f) \quad \text{subject to} \quad F(x|f) \geq F(x|g) \quad \forall x \in D^V. \quad (5)$$

► **Theorem 2** ([17]). *The optimal value of problem (5) is $\max_{x \in D^V} F(x|g)$.*

Proof. By Theorem 1(a), every feasible f satisfies

$$B(f) \geq F(x|f) \geq F(x|g) \quad \forall x \in D^V. \quad (6)$$

Denoting $F^* = \max_x F(x|g)$, this implies $B(f) \geq F^*$. To see that this bound is attained, consider f defined by $f_t = F^* / (|V| + |E|)$ for all $t \in T$. It can be checked from (1) and (3) that $B(f) = F(x|f) = F^*$ for all x , so f is feasible and optimal. ◀

Theorem 2 says that any feasible solution f to (5) yields an upper bound $B(f)$ on the optimal value of WCSP g , which is attained if f is optimal for (5). Thus, finding a global optimum of (5) in fact means solving the WCSP g . This is not surprising, as the complexity of the WCSP is hidden in the exponential set of constraints of (5).

► **Theorem 3.** *Let $g \in \mathbb{R}^T$. Let $f \in \mathbb{R}^T$ be feasible for (5). Then f is optimal for (5) if and only if CSP $A^*(f)$ has a solution x satisfying $F(x|f) = F(x|g)$.*

Proof. By (6) and Theorem 2, a feasible f is optimal if and only if $B(f) = F(x|f) = F(x|g)$ for some x . The claim now follows from Theorem 1. ◀

Next, we state a useful corollary of Theorem 3.

► **Theorem 4.** *Let $g \in \mathbb{R}^T$. CSP $A^*(g)$ is unsatisfiable if and only if there exists $h \in \mathbb{R}^T$ such that $B(g + h) < B(g)$ and $F(x|h) \geq 0$ for all $x \in D^V$.*

⁵ Namely, by introducing auxiliary variables $z_U \in \mathbb{R}$, problem (4) is equivalent to minimizing $\sum_{U \in \mathcal{U}} z_U$ subject to $z_U \geq f_t \quad \forall U \in \mathcal{U}, t \in U$ and $F(x|f) = F(x|g) \quad \forall x \in D^V$, which is a linear program.

⁶ They are called *sup-reparametrizations* in [23] and *virtual potentials* in [17].

Proof. Theorem 3 in particular says that problem (5) attains its optimum at $f = g$ if and only if $A^*(g)$ is satisfiable. That is, $A^*(g)$ is unsatisfiable if and only if there exists $f \in \mathbb{R}^T$ such that $B(f) < B(g)$ and $F(x|f) - F(x|g) = F(x|f - g) \geq 0$ for all x . Substituting $h = f - g$ yields the desired claim. \blacktriangleleft

We will refer to vector h in Theorem 4 as a *certificate of unsatisfiability of CSP $A^*(g)$ for g* . Note that “for g ” is important because h depends not only on the set $A^*(g)$ but also on the vector g itself. We will discuss this in more detail later on in §3.2.1.

3.2.1 Iterative Scheme

Given a feasible solution f to (5), we call a vector $h \in \mathbb{R}^T$ an *improving vector* if vector $f' = f + h$ is feasible for (5) and $B(f') < B(f)$. Clearly, an improving vector exists if and only if f is not optimal for (5). Theorem 3 says that for f to be optimal, it is necessary (but not sufficient) that CSP $A^*(f)$ is satisfiable. Therefore, if $A^*(f)$ is unsatisfiable, there exists an improving vector. Any certificate h of unsatisfiability of $A^*(f)$ for f (i.e., h satisfies $B(f + h) < B(f)$ and $F(x|h) \geq 0$ for all x) is such an improving vector: indeed, $f' = f + h$ is feasible for (5) because $F(x|f') = F(x|f) + F(x|h) \geq F(x|f) \geq F(x|g)$ for all x .

This suggests an iterative scheme to progressively improve feasible solutions to (5): initialize $f := g$ and then repeat this iteration (see Figure 1 in the appendix for an example):

1. If CSP $A^*(f)$ is satisfiable, stop. Otherwise, find a certificate h of unsatisfiability of $A^*(f)$ for f .
2. Update $f := f + h$.

Recall that satisfiability of $A^*(f)$ is not sufficient for optimality of f , as we are neglecting the (difficult) condition $F(x|f) = F(x|g)$ in Theorem 3. Consequently, in Step 1 we are able to generate only improving vectors h satisfying $F(x|h) \geq 0$ for all x , while general improving vectors (as defined above) need to satisfy only $F(x|f + h) \geq F(x|g)$ for all x . The former condition implies the latter but not *vice versa*. Therefore, fixed points of the algorithm are *local minima* of problem (5), in the sense that a fixed point cannot be improved by moving in any direction h satisfying the former condition (but possibly can be improved by moving in a direction h satisfying the latter condition).

During the algorithm, this manifests itself as follows. At any time, f satisfies (6), hence also $B(f) \geq \max_x F(x|f) \geq \max_x F(x|g)$. In every iteration, $B(f)$ decreases and the number $\max_x F(x|f)$ increases or does not change (due to $F(x|h) \geq 0$ for all x). When these two numbers meet, $A^*(f)$ becomes satisfiable by Theorem 1 and the algorithm stops. Monotonic increase of $\max_x F(x|f)$ can be seen as “greediness” of the algorithm: if we could generate general improving vectors, $\max_x F(x|f)$ could also decrease. Any increase of $\max_x F(x|f)$ is undesirable because the bound $B(f)$ will never be able to get below it.

Due to this greediness, the achievable (i.e., after possible convergence) gap $B(f) - \max_x F(x|g)$ critically depends on the “quality” of the certificates h . For a given f , there can be many certificates h of unsatisfiability of $A^*(f)$ for f . Good certificates are those for which the difference $\max_x F(x|f + h) - \max_x F(x|f) \geq 0$ is small (ideally zero). Intuitively, this means $F(x|h)$ should be zero for most of the assignments x and small for the remaining assignments. In turn, one heuristic for this is to keep vector h sparse⁷.

⁷ Sparsity of h is not the whole answer, though, because, e.g., vectors h satisfying $F(x|h) = 0$ for all x can be dense, according to Footnote 4.

So far, we supposed that in Step 1 of the algorithm we were always able to decide if CSP $A^*(f)$ is satisfiable. This is unrealistic because the CSP is NP-complete. But the approach remains applicable even if we detect unsatisfiability of $A^*(f)$ (and provide a certificate h) only sometimes, e.g., using constraint propagation. Then the iterative scheme becomes:

1. Attempt to prove that CSP $A^*(f)$ is unsatisfiable. If we succeed, find a certificate h of unsatisfiability of $A^*(f)$ for f . If we fail, stop.
2. Update $f := f + h$.

In this case, the fixed points of the algorithm will be even weaker local minima of problem (5), but they nevertheless might be still non-trivial and useful.

In the rest of the paper we develop this approach in detail. More precisely, we will compute an improving vector h in two steps: first (in §4) we compute an *improving direction* $d \in \mathbb{R}^T$ from $A^*(f)$ using constraint propagation, and then (in §5) we compute a suitable *step length* $\alpha > 0$ such that $h = \alpha d$. This is because from the CSP $A^*(f)$ alone it is possible to obtain only improving directions, while the step α depends also on f .⁸

Relation to Existing Approaches

The Augmenting DAG algorithm [18, 32] and the VAC algorithm [5] are (up to the precise way of computing the certificates h) an example of the described approach, which uses arc consistency to prove unsatisfiability of $A^*(f)$. In this favorable case, there exist certificates h that satisfy $F(x|h) = 0$ for all x , so we are in fact solving (4) rather than (5). For stronger local consistencies such certificates in general do not exist (i.e., $F(x|h) > 0$ for some x).

The algorithm proposed in [17] can be also seen as an example of our approach. It interleaves iterations using arc consistency (in fact, the Augmenting DAG algorithm) and iterations using cycle consistency.

As an alternative to our approach, stronger local consistencies can be achieved by introducing new weight functions (of possibly higher arity) into the WCSP objective (1) and minimizing an upper bound by reparametrizations, as in [24, 1, 33, 34, 19]. In our particular case, after updating $f := f + h$ we could introduce⁹ a weight function with scope formed by the variables of all tuples $t \in T$ with $h_t \neq 0$. In this view, our approach can be seen as enforcing stronger local consistencies but omitting these compensatory higher-order weight functions, thus saving memory.

Finally, the described approach can be seen as an example of the primal-dual approach [12] to optimize linear programs using constraint propagation.

4 Deactivating Directions

Here we describe a special kind of directions, *deactivating directions* (this name will be justified in §5). Under additional conditions, these directions certify unsatisfiability of a CSP.

► **Definition 5.** Let $A \subseteq T$ and $S \subseteq A$, $S \neq \emptyset$. An S -deactivating direction for CSP A is a vector $d \in \mathbb{R}^T$ satisfying

- (a) $d_t < 0$ for all $t \in S$,
- (b) $d_t = 0$ for all $t \in A - S$,
- (c) $F(x|d) \geq 0$ for all $x \in D^V$. ┘

Note that for fixed A and S , all S -deactivating directions for A form a convex cone.

⁸ It follows from Theorem 4, 6, and 15 that a CSP $A \subseteq T$ is unsatisfiable if and only if there is a direction $d \in \mathbb{R}^T$ such that (i) $F(x|d) \geq 0$ for all x and (ii) for every $f \in \mathbb{R}^T$ such that $A = A^*(f)$ there exists $\alpha > 0$ such that $B(f + \alpha d) < B(f)$.

⁹ Notice that such an added weight function would not increase the bound (3) since its weights are non-positive due to the fact that it needs to decrease the value for some assignments.

► **Theorem 6.** *Let $A \subseteq T$ and $S \subseteq A$, $S \neq \emptyset$. An S -deactivating direction $d \in \mathbb{R}^T$ for A exists if and only if CSP $A|_t$ has no solution for any $t \in S$.*

Proof. For one direction, we proceed by contradiction. Let d be an S -deactivating direction for A and let x^* be a solution to A that uses at least one tuple from S . By (1), we have $F(x^*|d) < 0$ because $d_t = 0$ for all $t \in A - S$ by condition (b) in Definition 5 and $d_{t^*} < 0$ for all $t^* \in S$ by condition (a). This contradicts condition (c).

For the other direction, if CSP $A|_t$ has no solution for any $t \in S$, sets S and $P = T - A$ satisfy Property 7, so an S -deactivating direction exists by Theorem 8 (given below). ◀

Suppose $d \in \mathbb{R}^T$ is an S -deactivating direction for $A \subseteq T$. If for some $U \in \mathcal{U}$ it holds that $(A - S) \cap U = \emptyset$ (equivalently¹⁰, $A \cap U \subseteq S$) then, by Theorem 6, CSP A is unsatisfiable because (as noted in §2) every assignment uses exactly one tuple from every set from \mathcal{U} . In that case, d certifies unsatisfiability of CSP A .¹¹

4.1 Obtaining Deactivating Directions Using Constraint Propagation

Although the CSP is NP-complete, satisfiability of some CSPs can be disproved in polynomial time by constraint propagation. This is an iterative method, which in each iteration infers that certain tuples of a given CSP instance are not used by any solution and forbids these tuples. In contrast to usual usage of constraint propagation, we require that in every iteration it also provides an S -deactivating direction for the set of tuples S it forbids. By Theorem 6, such a direction always exists.

We will argue that finding an S -deactivating direction for a CSP A is not harder than inferring that $A|_t$ has no solution for any $t \in S$. Formally, we consider an algorithm (such as a constraint propagation method) satisfying the following property:

► **Property 7.** *The algorithm takes a set $A \subseteq T$ on input and returns sets $S \subseteq A$ and $P \subseteq T - A$ such that CSP $(T - P)|_t$ is unsatisfiable for every $t \in S$.* ◻

The condition in Property 7 is equivalent to requiring that for any CSP A' where all tuples from P are forbidden (i.e., $A' \subseteq T - P$), $A'|_t$ is unsatisfiable for all $t \in S$.¹² Note that this implies that CSP $A|_t$ is unsatisfiable for all $t \in S$ due to $A \subseteq T - P$.

Returning $S = \emptyset$ indicates that the algorithm is not able to forbid any tuple. In addition, the algorithm provides a “proof” set $P \subseteq T - A$ which can be interpreted as a set of tuples which are needed to verify that $A|_t$ is unsatisfiable for each $t \in S$. It is natural that P is a subset of forbidden tuples since such tuples suffice to disprove satisfiability of a CSP.

► **Theorem 8.** *Let $A \subseteq T$. If an algorithm satisfying Property 7 takes A on input and returns sets S, P with $S \neq \emptyset$, then $d \in \mathbb{R}^T$ defined as¹³*

$$d_t = \begin{cases} -1 & \text{if } t \in S \\ |\{U \in \mathcal{U} \mid U \cap S \neq \emptyset\}| & \text{if } t \in P \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

is S -deactivating for A .

¹⁰ Indeed, for any $A, S, U \subseteq T$ we have $(A - S) \cap U = \emptyset \iff A \cap U \subseteq S$. We will use this equivalence repeatedly in the sequel.

¹¹ Let us emphasise that this is different from the certificate of unsatisfiability of CSP $A^*(f)$ for f (in the sense of Theorem 4) because deactivating directions do not contain the step length. Following Footnote 8, the step length can be computed for any WCSP f with $A^*(f) = A$ using Theorem 15.

¹² This holds due to $A \subseteq T - P$ and the fact that if A' is unsatisfiable, then any $A \subseteq A'$ is unsatisfiable.

¹³ $|\{U \in \mathcal{U} \mid U \cap S \neq \emptyset\}|$ is the number of scopes in \mathcal{U} which contain at least one tuple from S . In other words, every assignment uses at most $|\{U \in \mathcal{U} \mid U \cap S \neq \emptyset\}|$ tuples from S .

Proof. Conditions (a) and (b) of Definition 5 are clearly satisfied and it only remains to show that $F(x|d) \geq 0 \forall x \in D^V$. We proceed by contradiction: let $x^* \in D^V$ such that $F(x^*|d) < 0$. Denote $m = |\{U \in \mathcal{U} \mid U \cap S \neq \emptyset\}|$. Necessarily, x^* uses at least one tuple $t^* \in S$ (otherwise, $F(x^*|d)$ would not be negative). Additionally, x^* does not use any tuple from P . The reason is that every x uses exactly one tuple from each set $U \in \mathcal{U}$, so it can use at most m tuples t with $d_t = -1$. If at least one tuple from P was used, we would have $F(x|d) \geq 0$ by (7).

Since x^* uses only tuples from the set $T - P$, it is a solution to CSP $T - P$. But x^* uses at least one tuple $t^* \in S$, i.e., $(T - P)|_{t^*}$ is satisfiable, which contradicts Property 7. ◀

Note that Property 7 can be easily satisfied by any algorithm which prunes the CSP search space by forbidding tuples which are not used in any solution. Such tuples form the set S and set P can always be trivially chosen as $P = T - A$. Unfortunately, deactivating direction (7) calculated using $P = T - A$ would have many positive components. Consequently, an update of weights along such deactivating direction may substantially increase the values $F(x|f)$, which is undesirable as explained in §3.2.1.

Ideally, P should be as small as possible because then the deactivating directions do not increase the weights much and thus allow subsequent improvement of the bound. Though finding the smallest set P satisfying Property 7 is probably intractable¹⁴, in practice we can often easily find a small such set. E.g., P can simply be the set of forbidden tuples which the algorithm needed to visit to make its decision. Alternatively, it may only be necessary to check the support of some tuple to forbid it. Importantly, P need not be the same for each CSP instance, even for a fixed level of local consistency. For example, if the arc consistency closure of A is empty, then A is unsatisfiable, but a domain wipe-out may occur sooner or later depending on A , which affects which tuples needed to be visited.

We will now give examples of deactivating directions corresponding to well-known consistency conditions.

► **Example 9.** Let us consider *arc consistency* (AC). A CSP A is arc consistent if the equivalence¹⁵ $(i, k) \in A \iff \bigvee_{l \in D} (\{(i, k), (j, l)\} \in A)$ holds for all $\{i, j\} \in E, k \in D$.

Let $k \in D$ and $\{i, j\} \in E$. If $(i, k) \in A$ and $\{(i, k), (j, l)\} \notin A$ for all $l \in D$, the AC propagator infers that $A|_{(i, k)}$ is unsatisfiable and forbids the tuple (i, k) , that is, returns $S = \{(i, k)\}$. An S -deactivating direction d for A can be in this case simply

$$d_t = \begin{cases} -1 & \text{if } t = (i, k) \\ 1 & \text{if } t \in \{(i, k), (j, l)\} \mid l \in D \\ 0 & \text{otherwise} \end{cases} \quad (8)$$

because to forbid the tuple (i, k) , it sufficed to verify that $\{(i, k), (j, l)\} \notin A$ for all $l \in D$. Thus, $P = \{(i, k), (j, l)\} \mid l \in D$.

For the other case, let $k \in D$ and $\{i, j\} \in E$. If $(i, k) \notin A$, AC propagator forbids tuples $S = \{(i, k), (j, l)\} \mid l \in D$ based on $P = \{(i, k)\}$. In this particular case, it is a good idea to choose an S -deactivating direction d for A as

¹⁴ Set P is related (but not equivalent) to an *unsatisfiable core* of CSP $A|_t$. Finding a minimal unsatisfiable core is a “highly intractable problem” [15].

¹⁵ Note, for convenience we use a slightly unusual definition of arc consistency, allowing to restrict not only domains but also constraint relations.

$$d_t = \begin{cases} -1 & \text{if } t \in \{(i, k), (j, l)\} \mid l \in D\} \\ 1 & \text{if } t = (i, k) \\ 0 & \text{otherwise} \end{cases}. \quad (9)$$

Notice that all the binary tuples are set to -1 in (9), instead of just the tuples S as in (7). Although (7) would also provide an S -deactivating direction, it is better to use (9) because both directions d defined by (8) and (9) satisfy $F(x|d) = 0$ for all x . Thus, WCSPs g and $g + \alpha d$ are equivalent¹⁶ for any $\alpha \in \mathbb{R}$ which is desirable (see §3.2.1). \lrcorner

► **Example 10.** We now consider *cycle consistency* as defined in [17].¹⁷ Let \mathcal{C} be a (polynomially sized) set of cycles in the graph (V, E) . A CSP A is cycle consistent if for each tuple $(i, k) \in A \cap (V \times D)$ and each cycle $C \in \mathcal{C}$ that passes through node $i \in V$, there exists an assignment x with $x_i = k$ that uses only allowed tuples in cycle C . It can be shown that the cycle repair procedure in [17] in fact constructs a deactivating direction whenever an inconsistent cycle is found. Moreover, the constructed direction in this case coincides with (7) for a suitable set P which contains a subset of the forbidden tuples within the cycle. \lrcorner

► **Example 11.** Recall that a CSP A is *singleton arc consistent (SAC)* if for every tuple $t = (i, k) \in A \cap (V \times D)$, the CSP $A|_t$ has a non-empty arc-consistency closure. Good (i.e., sparse) deactivating directions for SAC can be obtained as follows. For some $(i, k) \in A \cap (V \times D)$, we enforce arc consistency of CSP $A|_{(i, k)}$, during which we store the causes for forbidding each tuple. If $A|_{(i, k)}$ is found to have an empty AC closure, we backtrack and identify only those tuples which were necessary to prove the empty AC closure. These tuples form the set P . The deactivating direction is then constructed as in Theorem 8 with $S = \{(i, k)\}$. Note that SAC does not have bounded support as many other kinds of local consistencies [3], so the size of P can be significantly different for different CSP instances. \lrcorner

4.2 Composing Deactivating Directions

Recall that constraint propagation iteratively forbids some tuples of a given CSP $A \subseteq T$, until it is no longer able to forbid any tuple or it becomes explicit that the CSP is unsatisfiable. The latter happens if all tuples of some set $U \in \mathcal{U}$ become forbidden¹⁸ (i.e., $U \cap A = \emptyset$), because (as noted in §2) every assignment uses exactly one tuple from every set from \mathcal{U} .

Formally, consider a propagation rule to enforce a local consistency condition Φ , such that if CSP A is not Φ -consistent then it finds a non-empty set $S \subseteq A$ of tuples to forbid and an S -deactivating direction¹⁹ for A . This rule is applied to the given CSP iteratively, each time forbidding a different set of tuples. This is outlined in Algorithm 1, which stores the generated sets S_r of tuples being forbidden and the corresponding S_r -deactivating directions d^r . Note that, by line 4 of the algorithm, $A_r = A - \bigcup_{q=0}^{r-1} S_q$ for every $r = 0, \dots, s+1$.

The generated sequence of S_r -deactivating directions d^r for A_r can be composed into a single $(\bigcup_{q=0}^s S_q)$ -deactivating direction for A using the following composition rule (the proof is given in the appendix):

¹⁶ Such reparametrizations correspond to soft arc consistency operations *extend* and *project* in [5].

¹⁷ This is different from *cyclic consistency* as defined in [4]. E.g., reparametrizations are sufficient to enforce cyclic consistency, whereas super-reparametrizations are needed for cycle consistency.

¹⁸ If $U = \{(i, k) \mid k \in D\}$ for some $i \in V$, this is often called “domain wipe-out”.

¹⁹ The deactivating direction can be constructed in any way, e.g. (but not necessarily) using Theorem 8.

■ **Algorithm 1** Propagation phase: given a CSP $A \subseteq T$, propagation is applied to A while deactivating directions are stored.

```

1 Initialize  $s := 0$ ,  $A_0 := A$ 
2 while  $A_s$  is not  $\Phi$ -consistent do
3   Find set  $S_s \subseteq A_s$  and an  $S_s$ -deactivating direction  $d^s$  for  $A_s$ .
4    $A_{s+1} := A_s - S_s$ 
5   if  $\exists U \in \mathcal{U} : U \cap A_{s+1} = \emptyset$  then
6     return unsatisfiable,  $(A_r)_{r=0}^{s+1}$ ,  $(S_r)_{r=0}^s$ ,  $(d^r)_{r=0}^s$ 
7    $s := s + 1$ 
8 return possibly satisfiable
    
```

■ **Algorithm 2** Composition phase: the sequences $(S_r)_{r=0}^s$ and $(d^r)_{r=0}^s$ generated by Algorithm 1 for given $R \subseteq \{0, \dots, s\}$, $R \neq \emptyset$ are composed to an M -deactivating direction d' for A .

```

1 Initialize  $r := \max R$ ,  $d' := d^r$ ,  $M := S_r$ .
2 while  $r > 0$  do
3    $r := r - 1$ 
4   if  $r \in R$  or  $\exists t \in S_r : d'_t \neq 0$  then
5      $d' := d' + \delta d^r$  (where  $\delta$  is given by (10) where  $d, S$  are replaced by  $d^r, S_r$ )
6      $M := M \cup S_r$ 
7 return  $d', M$ 
    
```

► **Proposition 12.** Let $A \subseteq T$ and $S, S' \subseteq A$ where $S \cap S' = \emptyset$. Let d be an S -deactivating direction for A . Let d' be an S' -deactivating direction for $A - S$. Let

$$\delta = \begin{cases} 0 & \text{if } d'_t \leq -1 \text{ for all } t \in S, \\ \max\{(-1 - d'_t)/d_t \mid t \in S, d'_t > -1\} & \text{otherwise.} \end{cases} \quad (10)$$

Then $d'' = d' + \delta d$ is an $(S \cup S')$ -deactivating direction for A .

Proposition 12 allows us to combine S_r -deactivating direction d^r for $A_r = A_{r-1} - S_{r-1}$ with S_{r-1} -deactivating direction d^{r-1} for A_{r-1} into a single $(S_{r-1} \cup S_r)$ -deactivating direction for A_{r-1} . By induction, we are able to gradually build a $(\bigcup_{q=0}^s S_q)$ -deactivating direction for A , which certifies unsatisfiability of A whenever Algorithm 1 returns “unsatisfiable”.

However, it is not always needed to construct a full $(\bigcup_{q=0}^s S_q)$ -deactivating direction as not every step of the propagator is necessary to prove unsatisfiability. Instead, one can choose any $U \in \mathcal{U}$ such that $U \cap A_{s+1} = \emptyset$ (equivalent to $U \cap (A - \bigcup_{q=0}^s S_q) = \emptyset$, i.e., $U \cap A \subseteq \bigcup_{q=0}^s S_q$) and construct an M -deactivating direction for a (possibly smaller) set $M \subseteq \bigcup_{q=0}^s S_q$, so that $U \cap A \subseteq M$. Such direction still certifies unsatisfiability of A and can be sparser than a $(\bigcup_{q=0}^s S_q)$ -deactivating direction, which is desirable as explained in §3.2.1.

This is outlined in Algorithm 2, which composes only a subsequence of directions given by the set $R \subseteq \{0, \dots, s\}$ and constructs an M -deactivating direction where $M \supseteq \bigcup_{r \in R} S_r$. Although Algorithm 2 is applicable for any set R , in our case R is obtained by first choosing any $U \in \mathcal{U}$ such that $U \cap (A - \bigcup_{q=0}^s S_q) = \emptyset$ and then setting $R = \{r \in \{0, \dots, s\} \mid S_r \cap U \neq \emptyset\}$, so that $U \cap (A - M) = \emptyset$ due to $U \cap A \subseteq \bigcup_{r \in R} S_r \subseteq M$. Correctness of Algorithm 2 is given by the following theorem, whose proof is given in the appendix.

► **Theorem 13.** *Algorithm 2 returns an M -deactivating direction d' for A with $M \supseteq \bigcup_{r \in R} S_r$.*

► **Remark 14.** This is similar to what the VAC [5] or Augmenting DAG algorithm [18, 32] do for arc consistency. To attempt to disprove satisfiability of CSP $A^*(f)$, these algorithms enforce AC of $A^*(f)$, during which the causes for forbidding tuples are stored. If the AC closure of $A^*(f)$ is found empty (which corresponds to $U \cap A_{s+1} = \emptyset$ for some $U \in \mathcal{U}$), these algorithms do not iterate through all previously forbidden tuples but only trace back the causes for forbidding the elements of the wiped-out domain (here, the elements of U). ─

5 Line Search and the Final Algorithm

In §4 we showed how to construct an S -deactivating direction $d \in \mathbb{R}^T$ for a CSP A , which certifies unsatisfiability of A whenever $A \cap U \subseteq S$ (i.e., $(A - S) \cap U = \emptyset$) for some $U \in \mathcal{U}$. For a WCSP $f \in \mathbb{R}^T$, to obtain a certificate $h \in \mathbb{R}^T$ of unsatisfiability of CSP $A^*(f)$ for f in the sense of Theorem 4, we need a step length $\alpha > 0$ so that $h = \alpha d$, as mentioned in §3.2.1. The step length is obtained using the following (somewhat more general) result, whose proof is given in the appendix.

► **Theorem 15.** *Let $f \in \mathbb{R}^T$. Let d be an S -deactivating direction for $A^*(f)$. Denote²⁰*

$$\beta = \min\{(\max_{t \in U(t')} f_t - f_{t'})/d_{t'} \mid t' \in T, d_{t'} > 0\},$$

$$\gamma = \min\{(f_t - f_{t'})/(d_{t'} - d_t) \mid U \in \mathcal{U}, A^*(f) \cap U \subseteq S, t \in U \cap S, t' \in U - S, d_{t'} > d_t\}.$$

Then $\beta, \gamma > 0$ and for every $U \in \mathcal{U}$ and $\alpha \in \mathbb{R}$, WCSP $f' = f + \alpha d$ satisfies:

- (a) *If $A^*(f) \cap U \not\subseteq S$ and $0 \leq \alpha \leq \beta$, then $\max_{t \in U} f'_t = \max_{t \in U} f_t$.*
- (b) *If $A^*(f) \cap U \subseteq S$ and $0 < \alpha \leq \min\{\beta, \gamma\}$, then $\max_{t \in U} f'_t < \max_{t \in U} f_t$.*
- (c) *If $A^*(f) \cap U \not\subseteq S$ and $0 < \alpha < \beta$, then $A^*(f') \cap U = (A^*(f) - S) \cap U$.*

If d is an S -deactivating direction for CSP $A^*(f)$ and for all $U \in \mathcal{U}$ we have $A^*(f) \cap U \not\subseteq S$ then, by Theorem 15(a,c), there is $\alpha > 0$ such that $f' = f + \alpha d$ satisfies $B(f') = B(f)$ and $A^*(f') = A^*(f) - S$. This justifies why such direction d is called S -deactivating: a suitable update of f along this direction makes tuples S inactive for f .

► **Remark 16.** This might suggest that to improve the current bound $B(f)$, we need not use Algorithm 2 to construct an S' -deactivating direction d' such that $A^*(f) \cap U \subseteq S'$ for some $U \in \mathcal{U}$, but instead perform steps using the intermediate S_r -deactivating directions d^r to create a sequence $f^{r+1} = f^r + \alpha_r d^r$ satisfying $B(f^0) = B(f^1) = \dots = B(f^s) > B(f^{s+1})$. Unfortunately, it is hard to make this work reliably as there are many choices for the intermediate step sizes $0 < \alpha_r < \beta_r$. We empirically found Algorithm 3 to be preferable. ─

If d is an S -deactivating direction for $A^*(f)$ and for some $U \in \mathcal{U}$ we have $A^*(f) \cap U \subseteq S$, then, by Theorem 15(a,b), there is $\alpha > 0$ such that $f' = f + \alpha d$ satisfies $B(f') < B(f)$. Thus, $h = \alpha d$ is a certificate of unsatisfiability of $A^*(f)$ for f in the sense of Theorem 4.

²⁰ β is always defined: by Definition 5 we have $F(x|d) \geq 0$ for all x , hence $\exists t: d_t < 0 \Rightarrow \exists t': d_{t'} > 0$. γ is defined and needed only in (b), where we assume that $U \cap A^*(f) \subseteq S$ for some $U \in \mathcal{U}$.

■ **Algorithm 3** The final algorithm to iteratively improve feasible solutions to (5) (i.e., an upper bound on the optimal value of WCSP g).

```

1 Initialize  $f := g$ .
2 while Algorithm 1 returns “unsatisfiable” on  $A^*(f)$  do
3   Generate sequences  $(A_r)_{r=0}^{s+1}$ ,  $(S_r)_{r=0}^s$ ,  $(d^r)_{r=0}^s$  by Algorithm 1.
4   Let  $U \in \mathcal{U} : U \cap A_{s+1} = \emptyset$  and set  $R := \{r \in \{0, \dots, s\} \mid S_r \cap U \neq \emptyset\}$ .
5   Compute  $M$ -deactivating direction  $d'$  using Algorithm 2.
6   Update  $f := f + \min\{\beta, \gamma\}d'$  following Theorem 15.
7 return  $B(f)$ 

```

Theorem 15 also proposes a possible (not necessarily optimal²¹) step size α . This allows us to formulate, in Algorithm 3, the iterative scheme outlined in §3.2.1. First, constraint propagation is applied to CSP $A^*(f)$ by Algorithm 1 until either $A^*(f)$ is proved unsatisfiable or no more propagation is possible. In the latter case, the algorithm halts and returns $B(f)$ as the best achieved upper bound on the optimal value of WCSP g . Otherwise, if $A^*(f)$ is proved unsatisfiable, we choose $U \in \mathcal{U}$ such that $U \cap A_{s+1} = \emptyset$, i.e., $A^*(f) \cap U \subseteq \bigcup_{r=0}^s S_r$ (which exists since Algorithm 1 returned “unsatisfiable”), define R so that $U \cap A^*(f) \subseteq \bigcup_{r \in R} S_r$, and compute an M -deactivating direction d' where $M \supseteq \bigcup_{r \in R} S_r$ using Theorem 13. Since $A^*(f) \cap U \subseteq M$, we can update WCSP f using Theorem 15. Consequently, the bound $B(f)$ strictly improves after each update on line 6.

In Algorithm 3 we additionally used a heuristic analogous to *capacity scaling* in network flow algorithms. We replace the active tuples $A^*(f)$ with “almost” active (θ -active) tuples $A_\theta^*(f) = \{t \in T \mid f_t \geq \max_{t' \in U(t)} f_{t'} - \theta\}$ for some threshold $\theta > 0$.²² This forces the algorithm to disprove satisfiability using tuples which are far from active, thus hopefully leading to larger step sizes and faster decrease of the bound. Initially θ is set to a high value and whenever we are unable to disprove satisfiability of $A_\theta^*(f)$, the current θ is decreased as $\theta := \theta/10$. The process continues until θ becomes very small.²³

6 Experiments

We implemented two versions of Algorithm 3 (incl. capacity scaling), differing in the local consistency used to attempt to disprove satisfiability of CSP $A^*(f)$:

- *Virtual singleton arc consistency via super-reparametrizations (VSAC-SR)*²⁴ uses singleton arc consistency. Precisely, we alternate between AC and SAC propagators: whenever a tuple (i, k) is removed by SAC, we step back to enforcing AC until no more AC propagations are possible, and repeat.

²¹ Finding an optimal step size (i.e., exact line search) would require finding a global minimum of the univariate convex piecewise-affine function $\alpha \mapsto B(f + \alpha d)$. As this would be too expensive for large instances, we find only a sub-optimal step size: we find the first break (i.e., non-differentiable) point of the function with a lower objective. This step size is decreased to β if $\gamma > \beta$ so that no maximum increases. This is analogous to the *first-hit strategy* in [11, §3.1.4].

²² This is similar to the notion of $\text{Bool}_\theta(f)$ in [5, §11.1], tolerance δ in [12, §4.2], and $\text{mi}_\epsilon[f]$ in [20, §6.2.4].

²³ Precisely, we initialized $\theta = \max_{k,l} g_{ij}(k, l) - \min_{k,l} g_{ij}(k, l) + \max_k g_{i'}(k) - \min_k g_{i'}(k)$ where $\{i, j\} \in E$ and $i' \in V$ is the edge and variable with the lowest index (based on indexing in the input instance). The terminating condition was $\theta \leq 10^{-6}$.

²⁴ In analogy to [5, 19], let us call a WCSP instance f *virtual X -consistent* (e.g., virtual AC or virtual RPC) if $A^*(f)$ has a non-empty X -consistency closure. Then, a *virtual X -consistency algorithm* naturally refers to an algorithm to transform a given WCSP instance to a virtual X -consistent WCSP instance. In the VAC algorithm, this transformation is equivalence-preserving, i.e., a reparametrization. But in our case, it is a super-reparametrization, which is why we call our algorithm VSAC-SR.

- *Virtual cycle consistency via super-reparametrizations (VCC-SR)* is the same as VSAC-SR except that SAC is replaced by CC.²⁵ Though our implementation is different than [17] (we compose deactivating directions rather than alternate between the cycle-repair procedure and the Augmenting DAG algorithm), it has the same fixed points.

The procedures for generating deactivating directions for AC, SAC and CC were implemented as described in Examples 9, 11, and 10. In SAC and CC it is useful to step back to AC whenever possible since, as described in §4.1, deactivating directions of AC correspond to reparametrizations and thus avoid increasing the values of individual assignments, which is beneficial as discussed in §3.2.1.

We compared the bounds calculated by VSAC-SR and VCC-SR with the bounds provided by EDAC [9], VAC [5], pseudo-triangles (option `-t=8000` in `toulbar2`, adds up to 8 GB of ternary weight functions), PIC, EDPIC, maxRPC, and EDmaxRPC [19] which are implemented in `toulbar2` [26].

We did the comparison on the Cost Function Library benchmark [8]. Due to limited computation resources, we used only the smallest 16500 instances (out of 18132). Of these, we omitted instances containing weight functions of arity 3 or higher. Moreover, to avoid easy instances, we omitted instances that were solved by VAC without search (i.e., `toulbar2` with options `-A -bt=0` found an optimal solution). Overall, 5371 instances were left for our comparison.

For each instance and each method, we only calculated the upper bound and did not do any search. For each instance and method, we computed the normalized bound $\frac{B_w - B_m}{B_w - B_b}$ where B_m is the bound computed by the method for the instance and B_w resp. B_b is the worst resp. best bound for the instance among all the methods. Thus, the best bound²⁶ transforms to 1 and the worst bound to 0, i.e., greater is better.

For 26 instances, at least one method was not able to finish in the prespecified 1 hour limit. These timed-out methods were omitted from the calculation of the normalized bounds for these instances. From the point of view of the method, the instance was not incorporated into the average of the normalized bounds of this particular method. We note that implementations of VSAC-SR and VCC-SR provide a bound when terminated at any time, whereas the implementations of the other methods provide a bound only when they are left to finish.²⁷

The results in Table 1 show that no method is best for all instance groups, instead each method is suitable for a different group. However, VSAC-SR performed best for most groups and otherwise was not much worse than the other strong consistency methods. VSAC-SR seems particularly good at `spinglass_maxcut` [25], `planning` [7] and `qplib` [13] instances. Taking the overall unweighted average of group averages (giving the same importance to each group), VSAC-SR achieved the greatest average value. We also evaluated the ratio to worst bound, B_m/B_w , for instances with $B_w \neq 0$; the results were qualitatively the same: VSAC-SR again achieved the best overall average of 3.93 (or 4.15 if only groups with ≥ 5 instances are considered) compared to second-best pseudo-triangles with 2.71 (or 2.84).

²⁵ We chose the cycles in VCC-SR as follows: if $2|E|/|V| \leq 5$ (i.e., average degree of the nodes is at most 5), then all cycles of length 3 and 4 present in the graph (V, E) are used. If $2|E|/|V| \leq 10$, then all cycles of length 3 present in the graph are used. If $2|E|/|V| > 10$ or the above method did not result in any cycles, we use all fundamental cycles w.r.t. a spanning tree of the graph (V, E) . No additional edges are added to the graph. Note, [17] experimented with grid graphs (where 4-cycles and 6-cycles of the grid were used) and complete graphs (where 3-cycles were used).

²⁶ To avoid numerical precision issues, bounds B_m within $B_b \pm 10^{-4}B_b$ or $B_b \pm 0.01$ are also normalized to 1. If $B_w = B_b$, then the normalized bounds for all methods are equal to 1 on this instance.

²⁷ Time-out happened 5, 2, 3, 6, and 24 times for pseudo-triangles, PIC, EDPIC, maxRPC, and EDmaxRPC, respectively. This did not affect the results much as there were 5731 instances in total.

The runtimes (on a laptop with i7-4710MQ processor at 2.5 GHz and 16GB RAM) are reported in Table 2. Again, the results are group-dependent and one can observe that the methods explore different trade-offs between bound quality and runtime. However, the strong consistencies are comparable in terms of runtime on average, except for pseudo-triangles which are faster but need significantly more memory.

Since both VSAC-SR and VCC-SR start by enforcing VAC (i.e., making $A^*(f)$ arc consistent by reparametrizations), before running these methods we used `toulbar2` to reparametrize the input WCSP instance to a VAC state (because a specialized algorithm is faster than the more general Algorithm 3). Besides this, we did no more attempts to fine-tune our implementation for efficiency. Thus, the set $A^*(f)$ was always calculated by iterating through all tuples. SAC was checked on all active tuples without warm-starting or using any faster SAC algorithm than SAC1 [2, 10]. Perhaps most importantly, we did not implement inter-iteration warm-starting as in [31, 11], i.e., after updating the weights on line 6 of Algorithm 3, some deactivating directions in the sequence which were not used to compose the improving direction may be preserved for the next iteration instead of being computed from scratch. Except for computing deactivating vectors, the code was the same for VSAC-SR and VCC-SR. We implemented everything in Java.

7 Concluding Remarks

We have proposed a method to compute upper bounds on the (maximization version of) WCSP. The WCSP is formulated as a linear program with an exponential number of constraints, whose feasible solutions are super-reparametrizations of the input WCSP instance (i.e., WCSP instances with the same structure and greater or equal objective values). Whenever the CSP formed by the active (i.e., maximal in their weight functions) tuples of a feasible WCSP instance is unsatisfiable, there exists an improving direction (in fact, a certificate of unsatisfiability of this CSP) for the linear program. As this approach provides only a subset of all possible improving directions, it can be seen as a local search. We showed how these improving directions can be generated by constraint propagation (or, more generally, by other methods to prove unsatisfiability of a CSP).

Special cases of our approach are the VAC / Augmenting DAG algorithm [5, 18, 32] which uses arc consistency and the algorithm in [17] which uses cycle consistency. We have newly implemented the approach for singleton arc consistency, resulting in VSAC-SR algorithm. When compared to existing soft local consistency methods on a public dataset, VSAC-SR provides comparable or better bounds for many instances.

The approach can be straightforwardly extended to WCSPs with different domain sizes, weight functions of any arities, and some weights equal to minus infinity (i.e., some constraints being hard). Note in particular that SAC is not restricted to binary CSPs. Of course, further experiments would be needed to evaluate the quality of the bounds in this case.

Our approach in general requires to store all the weights of the super-reparametrized WCSP instance. This may be a drawback when the domains are large and/or the weight functions are not given explicitly as a table of values but rather by an algorithm (oracle).

We expect our improved bounds to be useful to prune the search space during branch-and-bound search, when solving WCSP instances to optimality. However, we have done no experiments with this, so it is open whether during search the tighter bounds would outweigh the higher complexity of the algorithm. We leave this for the future research. Our approach can be also useful to solve more WCSP instances even without search (similarly as the VAC algorithm solves all supermodular WCSPs without search) or, given a suitable primal heuristic, to solve WCSP instances approximately.

■ **Table 1** Results on instances from Cost Function Library: Average normalized bounds.

Instance Group	Instances	EDAC	VAC	VSAC-SR	VCC-SR	Pseudo-tr.	PIC	EDPIC	maxRPC	EDmaxRPC
/biqmaclib/	157	0.02	0.11	0.90	0.22	0.92	0.83	0.81	0.79	0.81
/crafted/academics/	8	0.88	0.88	0.97	0.95	0.88	0.88	0.88	0.88	1.00
/crafted/auction/paths/	420	0.00	0.09	0.91	0.35	0.99	0.45	0.68	0.64	0.57
/crafted/auction/regions/	411	0.00	0.05	0.99	0.10	0.98	0.08	0.18	0.23	0.13
/crafted/auction/scheduling/	419	0.00	0.02	1.00	0.09	0.80	0.41	0.38	0.41	0.24
/crafted/coloring/	33	0.94	0.94	0.99	0.97	0.98	1.00	1.00	1.00	0.99
/crafted/feedback/	6	0.00	0.00	0.54	0.58	0.71	0.49	0.53	0.51	0.72
/crafted/kbtree/	1800	0.25	0.29	0.60	0.67	0.80	0.73	0.81	0.76	0.89
/crafted/maxclique/dimacs_maxclique/	49	0.06	0.24	0.98	0.39	0.87	0.39	0.50	0.51	0.55
/crafted/maxcut/spinglass_maxcut/unweighted/	5	0.00	0.00	1.00	0.42	0.15	0.15	0.15	0.15	0.15
/crafted/maxcut/spinglass_maxcut/weighted/	5	0.00	0.00	1.00	0.38	0.17	0.17	0.17	0.17	0.17
/crafted/modularity/	6	0.17	0.19	0.38	0.25	0.99	0.96	0.94	0.96	0.97
/crafted/planning/	65	0.00	0.54	0.94	0.72	0.32	0.07	0.09	0.07	0.17
/crafted/sumcoloring/	43	0.04	0.15	0.47	0.50	0.81	0.53	0.63	0.64	0.61
/crafted/warehouses/	49	0.35	0.99	1.00	0.99	0.35	0.42	0.42	0.42	0.42
/qaplib/	5	0.40	0.40	0.40	0.41	0.99	0.97	0.97	0.98	0.97
/qplib/	23	0.00	0.10	0.96	0.38	0.27	0.25	0.25	0.24	0.25
/random/maxcsp/completoose/	50	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
/random/maxcsp/completetight/	50	0.00	0.12	0.57	0.72	0.88	0.94	0.99	0.69	0.76
/random/maxcsp/denseloose/	50	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
/random/maxcsp/densetight/	50	0.02	0.14	0.52	1.00	0.68	0.48	0.49	0.52	0.60
/random/maxcsp/sparseloose/	90	0.96	0.96	1.00	0.96	0.96	0.96	0.96	0.96	0.96
/random/maxcsp/sparssetight/	50	0.01	0.12	0.54	1.00	0.64	0.40	0.40	0.43	0.51
/random/maxcut/random_maxcut/	400	0.00	0.00	0.77	0.13	0.95	0.98	0.98	0.97	0.99
/random/mincut/	500	0.09	1.00	1.00	1.00	0.10	0.10	0.10	0.10	0.10
/random/randomsat/	493	0.01	0.02	0.75	0.22	0.95	0.91	0.89	0.86	0.87
/random/wqueens/	6	0.00	0.52	0.96	0.94	0.48	0.12	0.29	0.13	0.72
/real/celar/	23	0.00	0.05	0.08	0.16	0.97	0.66	0.66	0.78	0.95
/real/maxclique/protein_maxclique/	1	0.00	0.00	1.00	0.03	0.93	0.04	0.04	0.08	0.04
/real/spot5/	1	0.00	0.08	1.00	0.49	1.00	0.74	0.66	0.41	0.74
/real/tagsnp/tagsnp_r0.5/	23	0.04	0.86	0.95	0.86	0.31	0.31	0.33	0.29	0.46
/real/tagsnp/tagsnp_r0.8/	80	0.13	0.66	0.91	0.68	0.29	0.39	0.38	0.33	0.47
Average over all groups	5371	0.20	0.36	0.82	0.58	0.72	0.56	0.58	0.56	0.62
Average over groups with ≥ 5 instances	5369	0.21	0.38	0.80	0.60	0.71	0.57	0.59	0.58	0.63

■ **Table 2** Results on instances from Cost Function Library: Average CPU time in seconds.

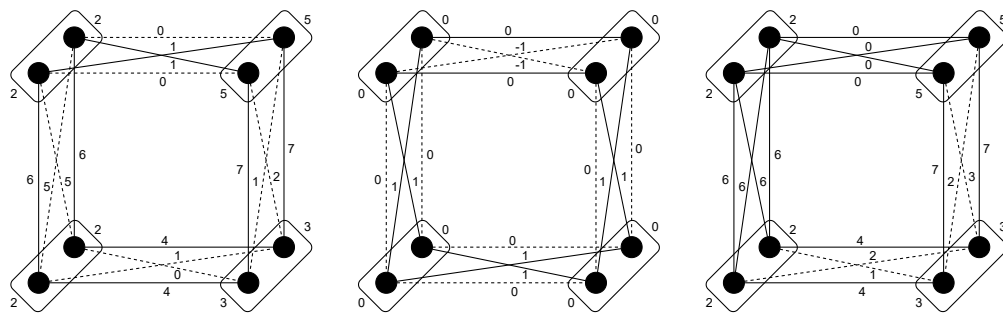
Instance Group	Instances	EDAC	VAC	VSAC-SR	VCC-SR	Pseudo-tr.	PIC	EDPIC	maxRPC	EDmaxRPC
/biqmaclib/	157	0.11	0.12	180.07	34.60	83.25	1240.00	1241.29	1242.16	1271.86
/crafted/academics/	8	0.11	0.11	28.61	1.04	29.08	121.44	120.86	108.08	104.47
/crafted/auction/paths/	420	0.04	0.04	1.96	0.83	1.92	0.19	0.23	0.48	0.64
/crafted/auction/regions/	411	0.20	0.32	32.14	9.45	673.42	49.85	51.37	102.61	110.48
/crafted/auction/scheduling/	419	0.10	0.12	16.22	2.03	49.85	26.90	26.89	32.06	32.30
/crafted/coloring/	33	0.09	0.10	4.99	1.40	0.20	545.50	545.50	545.51	545.50
/crafted/feedback/	6	0.70	0.70	3588.39	3600.11	11.64	1860.89	1874.08	1875.93	1873.07
/crafted/kbtree/	1800	0.02	0.02	3.13	11.25	0.10	0.04	0.05	0.06	0.07
/crafted/maxclique/dimacs_maxclique/	49	0.71	1.32	279.08	126.90	955.60	1345.67	1342.14	1429.73	1428.12
/crafted/maxcut/spinglass_maxcut/unweighted/	5	0.02	0.02	0.82	0.44	0.02	0.01	0.01	0.01	0.01
/crafted/maxcut/spinglass_maxcut/weighted/	5	0.02	0.02	1.09	0.53	0.02	0.01	0.01	0.01	0.01
/crafted/modularity/	6	0.19	0.29	1023.48	127.39	66.25	706.30	783.02	741.91	1442.57
/crafted/planning/	65	0.16	0.29	638.85	60.62	7.41	0.93	0.96	2.33	4.73
/crafted/sumcoloring/	43	1.29	1.94	727.49	963.61	255.72	1508.37	1508.36	1509.34	1512.68
/crafted/warehouses/	49	4.10	9.48	735.80	735.83	4.09	29.48	29.54	28.80	29.82
/qaplib/	5	0.08	0.09	119.05	278.53	7.38	1448.63	1444.95	1450.09	1449.22
/qplib/	23	0.13	0.14	255.85	43.11	195.32	626.25	626.24	626.27	626.36
/random/maxcsp/completoose/	50	0.06	0.06	1.31	0.16	0.48	0.09	0.10	0.19	0.18
/random/maxcsp/completetight/	50	0.02	0.03	6.35	12.68	0.47	0.21	0.25	0.31	0.33
/random/maxcsp/denseloose/	50	0.02	0.02	166.78	0.06	0.11	0.03	0.03	0.03	0.03
/random/maxcsp/densetight/	50	0.02	0.02	4.20	17.38	0.10	0.06	0.07	0.07	0.08
/random/maxcsp/sparseloose/	90	0.03	0.03	611.38	0.05	0.06	0.04	0.04	0.04	0.04
/random/maxcsp/sparssetight/	50	0.02	0.02	11.00	9.74	0.06	0.04	0.05	0.05	0.05
/random/maxcut/random_maxcut/	400	0.01	0.01	0.73	0.15	0.04	0.03	0.03	0.05	0.07
/random/mincut/	500	1.09	2.43	14.40	86.22	1.12	0.88	0.87	0.87	0.87
/random/randomsat/	493	0.02	0.02	3.42	0.17	0.13	0.07	0.10	0.16	0.31
/random/wqueens/	6	1.33	1.49	992.85	502.42	644.87	1800.15	1800.20	1800.18	1800.60
/real/celar/	23	0.27	0.28	1798.51	2972.69	66.56	300.76	219.91	495.26	1066.87
/real/maxclique/protein_maxclique/	1	0.26	0.44	25.24	6.77	1196.62	114.62	114.99	215.30	220.81
/real/spot5/	1	0.01	0.01	0.62	0.08	0.11	0.03	0.03	0.04	0.04
/real/tagsnp/tagsnp_r0.5/	23	4.83	378.77	3338.53	2897.83	239.38	3155.96	3148.66	3172.58	3295.19
/real/tagsnp/tagsnp_r0.8/	80	1.52	22.82	1239.73	858.83	90.05	195.12	206.76	359.55	409.88
Average over all groups	5371	0.55	13.17	495.38	417.59	143.17	471.21	471.49	491.88	538.35
Average over groups with ≥ 5 instances	5369	0.58	14.04	527.54	445.20	112.82	498.80	499.08	517.49	566.88

References

- 1 Dhruv Batra, Sebastian Nowozin, and Pushmeet Kohli. Tighter relaxations for MAP-MRF inference: A local primal-dual gap based separation algorithm. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 146–154, 2011.
- 2 Christian Bessiere, Stephane Cardon, Romuald Debruyne, and Christophe Lecoutre. Efficient algorithms for singleton arc consistency. *Constraints*, 16(1):25–53, 2011.
- 3 Christian Bessiere and Romuald Debruyne. Theoretical analysis of singleton arc consistency. In *Workshop on Modelling and Solving Problems with Constraints*, pages 20–29, 2004.
- 4 Martin C. Cooper. Cyclic consistency: a local reduction operation for binary valued constraints. *Artificial Intelligence*, 155(1-2):69–92, 2004.
- 5 Martin C. Cooper, Simon de Givry, Martí Sanchez, Thomas Schiex, Matthias Zytnicki, and Tomáš Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8):449–478, 2010.
- 6 Martin C. Cooper, Simon de Givry, and Thomas Schiex. Optimal soft arc consistency. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, volume 7, pages 68–73, 2007.
- 7 Martin C. Cooper, Marie de Roquemaurel, and Pierre Régnier. A weighted CSP approach to cost-optimal planning. *AI Communications*, 24(1):1–29, 2011.
- 8 <https://forgemia.inra.fr/thomas.schiex/cost-function-library>, commit 356bbb85.
- 9 Simon De Givry, Federico Heras, Matthias Zytnicki, and Javier Larrosa. Existential arc consistency: Getting closer to full arc consistency in weighted CSPs. In *IJCAI*, volume 5, pages 84–89, 2005.
- 10 Romuald Debruyne and Christian Bessiere. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI’97*, pages 412–417, 1997.
- 11 Tomáš Dlask. Minimizing convex piecewise-affine functions by local consistency techniques. *Master’s Thesis*, 2018.
- 12 Tomáš Dlask and Tomáš Werner. Bounding linear programs by constraint propagation: Application to Max-SAT. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 2020.
- 13 Fabio Furini, Emiliano Traversi, Pietro Belotti, Antonio Frangioni, Ambros Gleixner, Nick Gould, Leo Liberti, Andrea Lodi, Ruth Misener, Hans Mittelmann, et al. QPLIB: a library of quadratic programming instances. *Mathematical Programming Computation*, 11(2):237–265, 2019.
- 14 Amir Globerson and Tommi S Jaakkola. Fixing max-product: Convergent message passing algorithms for MAP LP-relaxations. In *Advances in Neural Information Processing Systems*, pages 553–560, 2008.
- 15 Eric Grégoire, Bertrand Mazure, and Cédric Piette. On finding minimally unsatisfiable cores of CSPs. *International Journal on Artificial Intelligence Tools*, 17(04):745–763, 2008.
- 16 Vladimir Kolmogorov. Convergent tree-reweighted message passing for energy minimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(10):1568–1583, 2006.
- 17 Nikos Komodakis and Nikos Paragios. Beyond loose LP-relaxations: Optimizing MRFs by repairing cycles. In *European conference on computer vision*, pages 806–820. Springer, 2008.
- 18 V. K. Koval and M. I. Schlesinger. Dvumernoe programmirovaniye v zadachakh analiza izobrazheniy (Two-dimensional programming in image analysis problems). *Automatics and Telemekhanics*, 8:149–168, 1976. In Russian.
- 19 Hiep Nguyen, Christian Bessiere, Simon de Givry, and Thomas Schiex. Triangle-based consistencies for cost function networks. *Constraints*, 22(2):230–264, 2017.
- 20 Bogdan Savchynskyy. Discrete graphical models – an optimization perspective. *Foundations and Trends in Computer Graphics and Vision*, 11(3-4):160–429, 2019.
- 21 M. Schlesinger. Sintaksicheskii analiz dvumernykh zritelnykh signalov v usloviyakh pomekh (syntactic analysis of two-dimensional visual signals in noisy conditions). *Kibernetika*, 4(113-130):2, 1976.

- 22 H. D. Sherali and W. P. Adams. A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems. *SIAM Journal of Discrete Mathematics*, 3(3):411–430, 1990.
- 23 David Sontag and Tommi Jaakkola. Tree block coordinate descent for MAP in graphical models. In *Artificial Intelligence and Statistics*, pages 544–551, 2009.
- 24 David Sontag, Talya Meltzer, Amir Globerson, Tommi Jaakkola, and Yair Weiss. Tightening LP relaxations for MAP using message passing, 2008.
- 25 <https://software.cs.uni-koeln.de/spinglass>.
- 26 <https://miat.inrae.fr/toulbar2>.
- 27 Siddharth Tourani, Alexander Shekhovtsov, Carsten Rother, and Bogdan Savchynskyy. MPLP++: Fast, parallel dual block-coordinate ascent for dense graphical models. In *Proceedings of the European Conference on Computer Vision*, pages 251–267, 2018.
- 28 Siddharth Tourani, Alexander Shekhovtsov, Carsten Rother, and Bogdan Savchynskyy. Taxonomy of dual block-coordinate ascent methods for discrete energy minimization. In *International Conference on Artificial Intelligence and Statistics*, pages 2775–2785. PMLR, 2020.
- 29 Stanislav Živný. *The Complexity of Valued Constraint Satisfaction Problems*. Cognitive Technologies. Springer, 2012.
- 30 Martin J. Wainwright and Michael I. Jordan. Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, 1(1-2):1–305, 2008.
- 31 Tomáš Werner. A linear programming approach to max-sum problem: A review. Technical Report CTU-CMP-2005-25, Center for Machine Perception, Czech Technical University, December 2005.
- 32 Tomáš Werner. A linear programming approach to max-sum problem: A review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(7):1165–1179, July 2007.
- 33 Tomáš Werner. Revisiting the linear programming relaxation approach to Gibbs energy minimization and weighted constraint satisfaction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(8):1474–1488, August 2010.
- 34 Tomáš Werner. Marginal consistency: Upper-bounding partition functions over commutative semirings. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(7):1455–1468, July 2015.

A Appendix



(a) WCSP f , $A^*(f)$ unsatisfiable. (b) Certificate h of unsatisfiability (c) WCSP $f+h$, $B(f+h) < B(f)$.
of $A^*(f)$ for f .

Figure 1 Example of a single iteration of the scheme. Variables (elements of V) are depicted as rounded rectangles, tuples (elements of T) as circles and line segments, and weights f_t are written next to the circles and line segments. Black circles and solid lines depict active tuples, dashed lines depict inactive tuples.

Proof of Proposition 12. First, if $d'_t \leq -1$ for all $t \in S$, then $d'' = d'$ satisfies the required condition immediately. Otherwise, $\delta > 0$ since $d_t < 0$ for all $t \in S$ by definition and $-1 - d'_t < 0$ due to $d'_t > -1$ in definition of δ . We will show that d'' satisfies the conditions in Definition 5.

For $t \in S$ with $d'_t \leq -1$, $d''_t = d'_t + \delta d_t < d'_t \leq -1$ because $\delta d_t \leq 0$. If $t \in S$ and $d'_t > -1$, then $\delta \geq (-1 - d'_t)/d_t$, so $d''_t = d'_t + \delta d_t \leq -1$. Summarizing, we have $d''_t < 0$ for all $t \in S$.

For $t \in S'$, $d'_t < 0$ and $d_t = 0$ holds by definition due to $S' \subseteq A - S$, thus $d''_t = d'_t + \delta d_t = d'_t < 0$ which together with the previous paragraph yields condition (a).

Due to $A - S \supseteq (A - S) - S' = A - (S \cup S')$, for any $t \in A - (S \cup S')$ we have $d_t = 0$ and $d'_t = 0$, which implies $d''_t = d' + \delta d = 0$, thus verifying condition (b).

To show (c): for any $x \in D^V$, $F(x|d'') = F(x|d') + \delta F(x|d) \geq 0$ by $\delta \geq 0$. ◀

Proof of Theorem 13. The fact that $M \supseteq \bigcup_{r \in R} S_r$ is obvious due to $S_{\max R} \subseteq M$ by initialization on line 1 and $S_r \subseteq M$ for any $r \in R$, $r < \max R$ because in such case the update on line 6 is performed.

It remains to show that d' is M -deactivating, which we will do by induction. We claim that vector d' is always M -deactivating direction for A_r on line 2 and M -deactivating direction for A_{r+1} on line 4.

Initially we have $d' = d''$, so d' is S_r -deactivating (i.e., M -deactivating since $M = S_r$ before the loop is entered) for A_r . Also, when vector d' is first queried on line 4, r decreased by 1 due to update on line 3, so d' is M -deactivating for A_{r+1} . The required property thus holds when the condition on line 4 is first queried with $r = \max R - 1$.

We proceed with the inductive step. If the condition on line 4 is not satisfied, then necessarily $d'_t = 0$ for all $t \in S_r$. So, if d' is M -deactivating for A_{r+1} , then it is also M -deactivating for $A_r = A_{r+1} \cup S_r$, as can be seen from Definition 5.

If the condition on line 4 is satisfied, d' is M -deactivating for A_{r+1} before the update on lines 5-6. Since $A_{r+1} = A_r - S_r$ and d'' is S_r -deactivating for A_r , Proposition 12 can be applied to d'' and d' to obtain an $(M \cup S_r)$ -deactivating direction for A_r . After updating M on line 6, it becomes M -deactivating for A_r .

Eventually, when $r = 0$, d' is M -deactivating for $A_0 = A$ by line 1 in Algorithm 1. ◀

Proof of Theorem 15. We have $\beta > 0$ because $d_{t'} > 0$ implies t' is an inactive tuple, so $\max_{t \in U(t')} f_t > f_{t'}$. We have $\gamma > 0$ because in $f_t - f_{t'}$ tuple t is always active and t' is inactive, hence $f_t > f_{t'}$.

To prove (a), let $A^*(f) \cap U \not\subseteq S$, so there is $t^* \in U$ such that $t^* \in A^*(f)$ and $t^* \notin S$. Hence, by Definition 5, $d_{t^*} = 0$ and value $\max_{t \in U} f'_t$ does not decrease for any α since $f'_{t^*} = f_{t^*} + \alpha d_{t^*} = f_{t^*}$. To show that the maximum does not increase, consider a tuple $t' \in U$ such that $d_{t'} > 0$ (due to $\alpha \geq 0$, tuples with $d_{t'} \leq 0$ can not increase the maximum). It follows that $\alpha \leq \beta \leq (\max_{t \in U} f_t - f_{t'})/d_{t'}$, so $f'_{t'} = f_{t'} + \alpha d_{t'} \leq \max_{t \in U} f_t$.

To prove (b), let $A^*(f) \cap U \subseteq S$. For all $t \in U \cap S$, we have $f'_t = f_t + \alpha d_t < f_t$ by $d_t < 0$ and $\alpha > 0$, i.e., $\max_{t \in U \cap S} f'_t < \max_{t \in U \cap S} f_t$. We proceed to show that $f'_t \leq \max_{t' \in U \cap S} f'_{t'}$ for every $t' \in U - S$. Let $t^* \in U \cap S$ satisfy $f'_{t^*} = \max_{t \in U \cap S} f'_t$. If $d_{t'} > d_{t^*}$, $\alpha \leq \gamma \leq (f_{t^*} - f_{t'})/(d_{t'} - d_{t^*})$ implies $f'_{t^*} = f_{t^*} + \alpha d_{t^*} \geq f_{t'} + \alpha d_{t'} = f'_{t'}$. If $d_{t'} \leq d_{t^*}$, then also $\alpha d_{t'} \leq \alpha d_{t^*}$ and $f'_{t'} = f_{t'} + \alpha d_{t'} \leq f_{t^*} + \alpha d_{t^*} = f'_{t^*}$ holds for any $\alpha \geq 0$ since $f_{t'} < f_{t^*}$. As a result, $\max_{t' \in U - S} f'_{t'} \leq \max_{t \in U \cap S} f'_t < \max_{t \in U \cap S} f_t = \max_{t \in U} f_t$.

To prove (c), let $A^*(f) \cap U \not\subseteq S$. Following (a), we have $\max_{t \in U} f_t = \max_{t \in U} f'_t$. If $t \in (A^*(f) - S) \cap U$, then $d_t = 0$ and such tuples remain active by $f'_t = f_t$. Tuples $t \in S \cap U$ become inactive since $f'_t = f_t + d_t \alpha < f_t = \max_{t' \in U} f_{t'}$ by $d_t < 0$ and $\alpha > 0$. Tuples $t \notin A^*(f)$ either satisfy $d_t \leq 0$ and can not become active or satisfy $d_t > 0$ and by $\alpha < \beta \leq (\max_{t' \in U} f_{t'} - f_t)/d_t$, $f'_t = f_t + d_t \alpha < \max_{t' \in U} f_{t'}$, so $t \notin A^*(f')$. ◀

Parallel Model Counting with CUDA: Algorithm Engineering for Efficient Hardware Utilization

Johannes K. Fichte ✉ 

TU Dresden, Germany

Markus Hecher ✉ 

TU Wien, Austria

Universität Potsdam, Germany

Valentin Roland ✉

TU Dresden, Germany

Abstract

Propositional model counting (MC) and its extensions as well as applications in the area of probabilistic reasoning have received renewed attention in recent years. As a result, also the need for quickly solving counting-based problems with automated solvers is critical for certain areas. In this paper, we present experiments evaluating various techniques in order to improve the performance of parallel model counting on general purpose graphics processing units (GPGPUs). Thereby, we mainly consider engineering efficient algorithms for model counting on GPGPUs that utilize the treewidth of a propositional formula by means of dynamic programming. The combination of our techniques results in the solver GPUSAT3, which is based on the programming framework CUDA that –compared to other frameworks– shows superior extensibility and driver support. When combining all findings of this work, we show that GPUSAT3 not only solves more instances of the recent Model Counting Competition 2020 (MCC 2020) than existing GPGPU-based systems, but also solves those significantly faster. A portfolio with one of the best solvers of MCC 2020 and GPUSAT3 solves 19% more instances than the former alone in less than half of the runtime.

2012 ACM Subject Classification Theory of computation → Logic; Computing methodologies → Massively parallel algorithms; Mathematics of computing → Graph algorithms

Keywords and phrases Propositional Satisfiability, GPGPU, Model Counting, Treewidth, Tree Decomposition

Digital Object Identifier 10.4230/LIPIcs.CP.2021.24

Supplementary Material *Software (Source Code, archived):* <https://zenodo.org/record/5539470>
Software (Source Code): <https://github.com/daajoe/GPUSAT/releases/tag/v3.000-pre>

Funding Johannes K. Fichte: Google Fellowship at the Simons Institute, UC Berkeley.

Markus Hecher: FWF Grants Y698 and P32830 and Grant WWTF ICT19-065.

Acknowledgements Authors are given in alphabetical order. Main work has been carried out while the first two authors were visiting the Simons Institute for the Theory of Computing. The authors gratefully acknowledge the valuable comments made by the anonymous reviewers. In particular, the authors thank Roland Yap for his efforts and feedback on the final version of this paper.

1 Introduction

Counting problems have perceived increasing interest in recent years. One of these problems that is well-studied is MC, which aims at counting the number of satisfying assignments of a given propositional formula. In fact, MC is canonical [3, 46] for the complexity class #P and there are a list of applications and variants thereof. Among those variants, extensions of the problem have been studied that involve, e.g., projecting satisfying assignments to certain variables or weighting variables, which enables applications like quantitative reasoning via Bayesian networks and other structures, e.g., [48, 9, 14]. Interestingly, there are also intensive studies focusing on approximation variants of MC, e.g., [7, 18, 6], whose goal is to approximate the number of satisfying assignments within a certain approximation factor.



© Johannes K. Fichte, Markus Hecher, and Valentin Roland;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 24; pp. 24:1–24:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

There are a list of solvers stemming from different technologies and approaches. These solvers have been pushed towards their limits with the help of a dedicated competition [22] for model counting. Among the different approaches, powerful solvers emerged based on caching, knowledge compilation, and parameterized complexity, e.g., [11, 51, 42, 47, 12, 44, 39, 49, 15]. Notably, for model counting, also techniques for parallel solving proved successful [5, 40]. Among those parallel solvers there are also solvers [28, 29] that utilize modern consumer general purpose graphics processing units (GPGPUs). Those existing GPGPU-based implementations are based on dynamic programming on a tree decomposition of different graph representations of a propositional formula. One particular graph representation [47], namely the so-called primal graph representation¹, proved successful since it is employed in the latest GPGPU-based implementation for MC, referred to by GPUSAT2 [29], but also in other solvers [8, 34, 27, 15, 16]. In this work, we take up this idea and systematically study improvements from the perspective of algorithm engineering in order to significantly speed up counting models of a propositional formula on GPGPUs. Our approach is an implementation that follows the existing implementation GPUSAT2, but we use the GPGPU framework CUDA instead of OPENCL since CUDA enables a more detailed and systematic performance analysis and hardware monitoring capabilities.

Contributions. We present a novel system GPUSAT3² that comprises the following new techniques and contributions. (i) We introduce a new clause representation, called *compact clause form (CCF)*, that allows us to check whether a partial assignment satisfies a clause by only two binary bit operations that can be efficiently implemented in hardware. (ii) Then, we enhance dynamic programming, which is designed to combine results (tables) of “local” computations, by *global caching*. Thereby we maintain a global cache on the GPGPU that is shared among different local tables in order to prevent copy overhead between the GPGPU memory and main memory (RAM) whenever possible. (iii) Further, we show the benefit of CUDA framework tuning, where we focus on quantifying the effect of *pinned memory*, a technique designed for reducing transfer overhead between GPGPU memory and RAM. (iv) Finally, we perform a study over existing *libraries for computing tree decompositions* in order to quantify the effect of those decomposers on the actual solving performance. These techniques and variants are systematically analyzed and presented individually. Then, the performance results involving the full system GPUSAT3 is given at the end. For the sake of presentation, *related work* is discussed in-place where suitable and applicable.

2 Preliminaries

Let α be a *bit vector* $\langle b_0, \dots, b_{n-1} \rangle$, which is a sequence consisting of n many *bits* that are integers between 0 and 1. Then, we refer to the i -th bit for position $0 \leq i < n$ by $\alpha_i := b_i$. We use the bit-wise XOR operator \oplus and the bit-wise AND operator $\&$ in the usual meaning. Further, we define the integer *value* of α by $\text{val}(\alpha) := \sum_{0 \leq i \leq n-1} 2^i \cdot \alpha_i$.

Propositional Satisfiability (SAT)

A literal is a propositional variable x or its negation $\neg x$. A *clause* is a finite set of literals, interpreted as the disjunction of these literals. A *(CNF) formula* is a finite set of clauses, interpreted as a conjunction of the clauses. Let F be a formula. Then, we refer to a

¹ The primal graph of a propositional formula distinguishes as vertices the variables of the formula and there is an edge between two variables, whenever those variables appear together in at least one clause.

² GPUSAT3 including benchmark data [26] is open source; available at github.com/vroland/GPUSAT.

set $S \subseteq F$ by a *sub-formula* (of F). For a clause $c \in F$, let $\text{var}(c)$ consist of all variables occurring in c and $\text{var}(F) := \bigcup_{c \in F} \text{var}(c)$. Without loss of generality, we assume for every $c \in F$ that $|c| = |\text{var}(c)|$, i.e., no variable appears twice in a clause. An *assignment* (over $V \subseteq \text{var}(F)$) is a mapping $A : V \rightarrow \{0, 1\}$. The formula $F[A]$ under A is obtained by removing all clauses from F that contain a literal set to 1 by A and removing from remaining clauses all literals set to 0 by A . An assignment A is *satisfying* if $F[A] = \emptyset$. The problem MC asks to count the number of satisfying assignments over $\text{var}(F)$ of a formula F .

► **Example 1.** Consider the formula $F := \{c_1, c_2, c_3\}$ with $c_1 := a \vee b \vee \neg c$, $c_2 := \neg b \vee \neg a$, and $c_3 := a \vee \neg d$. Then, $A_1 := \{a \mapsto 1, b \mapsto 0\}$ and $A_2 := \{a \mapsto 0, c \mapsto 0, d \mapsto 0\}$ are satisfying, i.e., $F[A_1] = F[A_2] = \emptyset$. In total, there are 7 satisfying assignments over $\{a, b, c, d\}$ of F .

Tree Decompositions (TDs), Treewidth, and Dynamic Programming

A *tree decomposition* (TD) of a given graph G is a pair $\mathcal{T} = (T, \chi)$ where T is a rooted tree and χ is a mapping which assigns to each node $t \in V(T)$ a set $\chi(t) \subseteq V(G)$, called *bag*, such that: (i) $V(G) = \bigcup_{t \in V(T)} \chi(t)$ and $E(G) \subseteq \{\{u, v\} \mid t \in V(T), \{u, v\} \subseteq \chi(t)\}$; and (ii) for each $r, s, t \in V(T)$, such that s lies on the path from r to t , we have $\chi(r) \cap \chi(t) \subseteq \chi(s)$. We let $\text{width}(\mathcal{T}) := \max_{t \in V(T)} |\chi(t)| - 1$. The *treewidth* $\text{tw}(G)$ of G is the minimum $\text{width}(\mathcal{T})$ over all TDs \mathcal{T} of G . The *primal graph* G_F [47] of a formula F has as vertices its variables and two variables are joined by an edge if they occur together in a clause of F . For brevity, we refer by *treewidth of a formula* to the treewidth of its primal graph. For a given node $t \in T$ of the primal graph G_F , we let $F_t := \{c \mid c \in F, \text{var}(c) \subseteq \chi(t)\}$ be the clauses entirely covered by $\chi(t)$. The formula $F_{\leq s}$ denotes the union over all F_t for all descendant nodes $t \in V(T)$ of s . In other words, $F_{\leq s}$ is the sub-formula of F containing all clauses entirely covered by a bag $\chi(s)$ for s and any of its descendant nodes.

► **Example 2.** Recall F from Example 1. From the primal graph P_F of F a TD \mathcal{T} of P_F with nodes t_1, t_2, t_3 can be constructed, where t_3 with $\chi(t_3) = \{a\}$ joins t_1, t_2 with $\chi(t_1) = \{a, b, c\}$ and $\chi(t_2) = \{a, d\}$. Intuitively, \mathcal{T} allows to evaluate F in parts. So, when evaluating $F = F_{\leq t_3}$, we split into $F_{\leq t_1}$ and $F_{\leq t_2}$, which refer to $\{c_1, c_2\}$ and $\{c_3\}$, respectively.

A solver based on *dynamic programming* for formulas evaluates the input formula F in parts along a given TD of the primal graph G_F . For each node t of the TD, results are usually stored in a *table* τ_t . The approach works in four steps as follows:

1. Construct the primal graph G_F of the input formula F .
2. Heuristically compute a tree decomposition $\mathcal{T} = (T, \chi)$ of the primal graph G_F .
3. Traverse the nodes in $V(T)$ in post-order. Thereby, at every node t , run an algorithm for computing table τ_t . This algorithm takes as input the bag $\chi(t)$, the sub-formula F_t , and previously computed tables for the child nodes of t . Such a table τ_t comprises rows of the form $\langle A, c \rangle$, where $A : \chi(t) \rightarrow \{0, 1\}$ is an assignment and c is an integer used for counting. Each row $\langle A, c \rangle$ indicates that there are c many satisfying assignments over $\text{var}(F_{\leq t})$ of $F_{\leq t}$ that extend A . These pairs are carefully maintained for all the different types of nodes; for details we refer to the literature [47, 29].
4. Print the model count by interpreting the result τ_n for the root n of T .

The worst-case runtime of such an algorithm for model counting is in $\mathcal{O}(2^k k d N)$, where k denotes the width of the primal graph, N refers to the number of nodes in the tree decomposition, and d denotes the maximum number of occurrences in the clauses of the input formula F over all variables of F [47].

General Purpose Graphics Processing Units (GPGPUs)

General purpose computing on graphics processing units (*GPGPU*) is the practice of exploiting the massively parallel computing capabilities of graphics cards for non-graphical and scientific applications. Historically, graphics hardware and drivers have been built with only graphics rendering pipelines in mind [52]. But with a growing demand for accelerating data-parallel programs, GPGPU frameworks like CUDA and OPENCL emerged. These offer APIs for writing GPGPU programs (functions) without relying on graphics primitives. Although frameworks aim to make programming applications for CPU and GPGPU more seamless, their memory spaces and instructions are distinct: While the CPU can access the *host memory (RAM)* and executes the *host program*, the GPGPU can only access *GPGPU memory (GPGPU RAM)* and execute functions called *kernels*. The functionality of CUDA and OPENCL largely overlaps. However, OPENCL is standardized and supports running CPUs as well. In contrast, CUDA is a proprietary platform for NVIDIA GPGPUs only, but CUDA is often perceived as a more mature ecosystem. This comprises tools like profilers and runtime checkers, learning resources, but also driver support.

Dynamic Programming With GPGPUs. Dynamic programming as described above is implemented in the solver GPUSAT2 [29], which heavily uses OPENCL and introduces a framework for efficiently solving MC in parallel on GPGPUs. For GPUSAT2, besides a simple implementation of storing tables via a fixed pre-allocated memory block, called ARRAY data structure, the authors have proposed an advanced data structure, referred to by TREE, which is a binary search tree that can be manipulated in parallel by the GPGPU. This solver also provides a heuristic that is used internally in order to decide whether to use ARRAY or TREE based on a tree decomposition width threshold.

Benchmarks

In order to systematically analyze performance, we use the following instances and hardware.

Benchmark Instances. We use the 200 instances of track 1 (private and public) of the 1st International Competition on Model Counting (MCC 2020) [22] as the MCC2020-TRACK1 benchmark. For our final evaluation, we additionally incorporate the instances of track 2 of MCC 2020 with variable weights stripped, making them unweighted. The 400 instances of this combined benchmark set are referred to as MCC2020-TRACK1+2.

Benchmark Hardware. We run benchmarks on three different machines. **Server:** 2x Intel Xeon Silver 4112@2.60GHz 128GB RAM, NVIDIA GeForce GTX 1180 8GB GPGPU RAM, Ubuntu 18.04 LTS, CUDA 9.1.85. **Desktop:** Intel Core i7-9700@3.00GHz 16GB RAM, NVIDIA Quadro RTX 4000 8GB GPGPU RAM, Ubuntu 18.04 LTS, CUDA 11.0. **Cluster:** Cluster of 44 nodes; 2x Intel Xeon E5-2680v3@2.50GHz, 256GB RAM, RHEL 7.9.

SERVER is used for running full benchmarks with various decomposers, as it provides an environment that is not shared with other users and enough memory resources. For detailed profiling, DESKTOP is employed due to the availability of a more current driver and local access to the machine. CLUSTER is chosen for comparing different tree decomposition libraries since it provides the resources needed to finish a massive amount of runs quickly.

3 Algorithm Engineering and Hardware Utilization

While the existing GPGPU-based system GPUSAT2 delivers decent performance compared with state-of-the-art model counters, a number of possible improvements as well as hardware-specific potentials are left unexplored. In this section, we outline several algorithmic as well as implementation-specific improvements for Step 3 of dynamic programming on tree decompositions as defined in the preliminaries. We then systematically evaluate the impact of these improvements, which finally leads to a *new system GPUSAT3*. Since GPUSAT3 is based on CUDA, we can leverage tools and suitable workflows for a systematic analysis.

Next, we introduce a compact representation of clauses as bit vectors that allows us to efficiently check for satisfiability on GPGPU hardware. Then, we describe a global caching scheme for result tables such that GPUSAT3 avoids superfluous transfers between host memory (RAM) and *GPGPU memory (VRAM)*. Lastly, we show that using the so-called pinned memory of CUDA, while introducing some overhead, benefits performance by increasing data transfer speed between host and GPGPU.

Compact Clause Form (CCF)

Clearly we cannot hope that GPUSAT3 solves instances of arbitrarily large treewidth. Since GPUSAT3 is based on dynamic programming aiming for utilizing reasonably small treewidth, we therefore restrict ourselves to instances below a reasonable threshold like treewidth 64. This allows us to build a clause data structure that is more optimized for satisfiability testing on GPGPUs. Assume a formula F , a TD $\mathcal{T} = (T, \chi)$ of G_F , and a node t of T .

Interestingly, every clause $c \in F_t$ can then be represented with two bit vectors, namely an occurrence vector occ and a sign vector sign , which both combined correspond to the *compact clause form (CCF)* of c . To the end of defining this compact representation, let $\text{idx} : \chi(t) \rightarrow \{0, \dots, |\chi(t)|-1\}$ be a bijective function that assigns each variable $v \in \chi(t)$ a *positional index* from 0 to the number $|\chi(t)|-1$ of variables minus one, thereby adhering to some fixed total ordering of variables in $\chi(t)$. Since idx is bijective, we denote the inverse of idx by idx^{-1} . Then, the *occurrence vector* $\text{occ}(c)$ for c is a sequence consisting of $|\chi(t)|$ many bits such that whenever $v \in \text{var}(c)$, the corresponding bit $\text{occ}(c)_{\text{idx}(v)}$ is set to 1 (and to 0 otherwise). The *sign vector* $\text{sign}(c)$ for c is of the same form as the occurrence vector such that $\text{sign}(c)_{\text{idx}(v)}$ is set to 1 whenever $\neg v \in c$. Otherwise, bit $\text{sign}(c)_{\text{idx}(v)}$ is set to 0.

In order to test if an assignment satisfies a set of clauses in CCF, assignments must be in a compact representation as well. Let A be an assignment over $\chi(t)$. Then, we compactly represent A as an *assignment vector* \vec{A} such that the i -th bit \vec{A}_i of \vec{A} for $0 \leq i < |\chi(t)|$ corresponds to the truth value of the variable at position i in A , i.e., $\vec{A}_i = A(\text{idx}^{-1}(i))$.

► **Proposition 3** (Correctness of CCF). *Assume a formula F , a TD $\mathcal{T} = (T, \chi)$ of G_F as well as a node t of T . Let further $c \in F_t$ be a clause and A be any assignment over variables $\chi(t)$.*

Then, A satisfies c if and only if $\text{val}((\vec{A} \oplus \text{sign}(c)) \& \text{occ}(c)) \geq 1$, where \oplus and $\&$ denotes the bit-wise XOR and AND operator, respectively.

► **Example 4.** Consider the clauses from Ex. 1: $c_1 = a \vee b \vee \neg c$, $c_2 = \neg b \vee \neg a$, and $c_3 = a \vee \neg d$. Observe that for the given total ordering $\langle a, b, c, d \rangle$, there is only one unique positional index function idx , defined by $\text{idx}(a) := 0$, $\text{idx}(b) := 1$, $\text{idx}(c) := 2$, and $\text{idx}(d) := 3$. Then, the corresponding bit vectors are $\text{occ}(c_1) = 1110$, $\text{occ}(c_2) = 1100$, $\text{occ}(c_3) = 1001$ and sign vectors are $\text{sign}(c_1) = 0010$, $\text{sign}(c_2) = 1100$, $\text{sign}(c_3) = 0001$.

Now, let us check the assignment $A = \{a \mapsto 0, b \mapsto 1, c \mapsto 0, d \mapsto 1\}$. In bit vector representation, this corresponds to $\vec{A} = 0101$. For c_1 , we have that $(\vec{A} \oplus \text{sign}(c_1)) \& \text{occ}(c_1) = (0101 \oplus 0010) \& 1110 = 0110$. Since $\text{val}(0110) = 6 \geq 1$, A satisfies c_1 . Conversely, $(\vec{A} \oplus \text{sign}(c_3)) \& \text{occ}(c_3) = (0101 \oplus 0001) \& 1001 = 0000$ indicates that A does not satisfy c_3 .

Observe that if we restrict ourselves to instances of treewidth below 64, the length of all involved vectors discussed above is bounded by 64 as well. So, testing if a truth assignment satisfies a clause can be efficiently implemented using 64-bit integers and bit-wise logic operators on top. More concretely, a satisfiability check can be implemented on any 64-bit hardware by only using one bit-wise XOR and one bit-wise AND operation for each clause in F_t . Interestingly, both occurrence and sign vectors can be computed once per TD node t and clause in F_t before actually invoking the GPGPU.

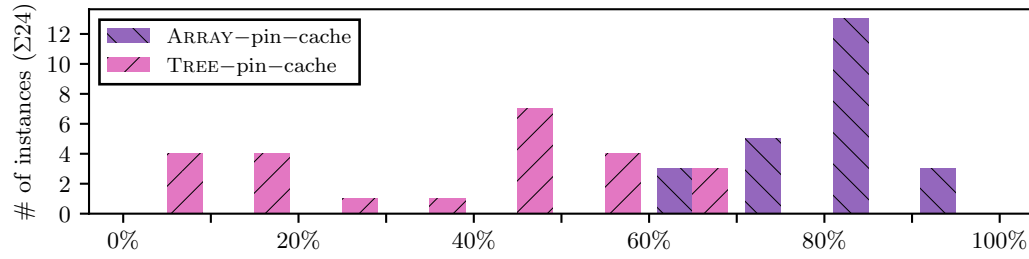
By carefully choosing the variable ordering (`idx`), we can ensure that the unique id of each parallel GPGPU computation unit performing such checks is a prefix of the assignment vector A . Consequently, the assignments tested in a single such computation unit only differs by a few of their least significant bits, allowing the assignment vector to be efficiently constructed by combining the unique id with a counter variable.

Achieving a form of parallelism using bit-wise instructions was used in the context of SAT solving [35]. There, a single instruction operates on multi-bit variable values representing multiple assignments. In our #SAT solver, multiple instructions operate on multiple assignments where each thread works on exactly one assignment. We obtain the assignments immediately from the thread id. Our compact representation of clauses minimizes thread divergence by taking a constant number of instructions for varying number of literals in a clause. In fact, small thread divergence is important for effective performance when running massive parallel execution on the GPGPU and low overhead of the used caches.

Reducing GPGPU Copy Overhead via Global Caching

In the preliminaries, we outline a model counting algorithm based on dynamic programming. The implementation in GPUSAT3 contains some steps that are performed on the GPGPU and others on the host. Thus, for each node in the tree decomposition, one or more *kernels* are executed which compute a table associated with the current node. Intuitively, if a kernel invocation uses a table produced by a previous kernel, this data can remain in VRAM and does not have to be copied to the host. If ideally VRAM was unlimited, no intermediate memory transfers to the host memory (RAM) would be needed, except for the final result. However, when tables become too large to store in VRAM alongside the next table, tables are divided into multiple *table chunks* per node in order to make solving such nodes feasible. Table chunks that are not currently needed are moved to main (host) memory, which is typically larger than the available VRAM.

Copying tables from and back to the VRAM can take a significant portion of the overall execution time. This leads to the idea of *global caching*: GPUSAT3 tries to keep whole tables or table chunks in a cache that is managed globally, in the sense that it potentially contains tables for several tree decomposition nodes at the same time. Thus, in the case that some or all child tables chunks already reside in VRAM when solving a node, i.e., they are *cached*, GPUSAT3 does not need to transfer them from host memory for solving. After a kernel execution, the result is left in the cache if the table is needed later in the solving process. More precisely, GPUSAT3 prefers to not transfer table chunks to the RAM until solving is completed; instead tracking a repository of chunks in VRAM.



■ **Figure 1** Histogram of percentages of GPGPU runtime spent on data transfers (`memcpy`) grouped in steps of 10% without global caching, using either TREE or ARRAY data structure.

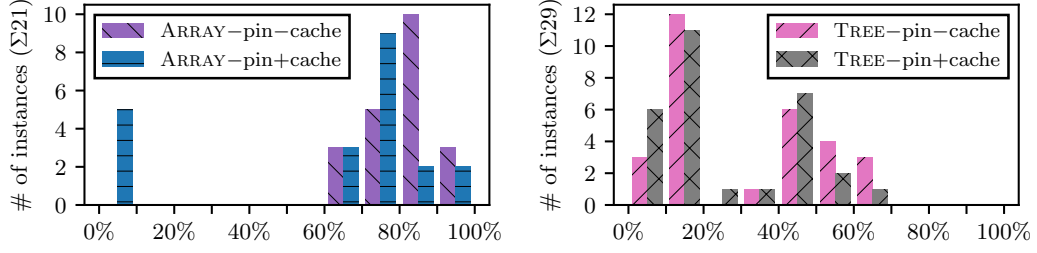
A chunk is deleted from the cache if (a) it is either no longer needed for solving subsequent nodes or (b) the VRAM is used otherwise. The latter (b) is the case if a new table needs more VRAM than available. In this scenario, all cache entries are evicted, since as much VRAM as possible is needed for solving the current node. This effectively degrades the cache to a local one, as only the currently needed table chunks can be kept. However, if tables are in relation to the available VRAM sufficiently small, the algorithm can benefit from keeping tables from both branches of a *join* node in the VRAM during the traversal. Ideally, this prevents intermediate transfers from the VRAM to host memory, except for the final results.

Evaluation. To investigate the need for global caching, we analyze the following hypothesis.

► **Hypothesis 1.** *GPUSAT3 spends large portions of runtime on GPGPU data transfers.*

To evaluate Hypothesis 1, we use the NVPROF profiler to determine how much GPGPU runtime is spent for copying data to and from the GPGPU. We relate this time to the total time spent in GPGPU functions during the solver run, as recorded by NVPROF. This represents the proportion of GPGPU runtime spent on data transfers instead of kernel executions. For small instances, this ratio may not be representative, as their data structures are typically very small. Constant costs like runtime initialization and memory allocation could further distort the results in such cases. Consequently, we only consider instances with a total solver runtime of at least 5s in one of the configurations. In each following comparison, only instances successfully solved by both compared configurations are included.

First, we compare this ratio for GPUSAT3 with caching and pinned memory disabled for both the ARRAY and TREE data structures, executed on DESKTOP with an arbitrary but fixed decomposition seed. Pinned memory is a technique to speed up data transfers between GPGPU and host, which is disabled here and will be explained in more detail in the next subsection. MCC2020-TRACK1 with a timeout of 600s is chosen as it contains many instances with sufficient treewidth to necessitate transfers between GPGPU and host memory. This allows us to get a baseline for the cost of data transfers without any optimizations. In Fig. 1, we show the distribution of GPGPU runtime in data transfers among the applicable instances of MCC2020-TRACK1 as a histogram. Clear differences are visible between the TREE and ARRAY data structures: With TREE, two clusters are visible at $\leq 20\%$ and $50\% - 70\%$. No instance uses more than 70% of GPGPU runtime in `memcpy`. With ARRAY, at least 60% is used, with the highest number of instances spending $80\% - 90\%$. Based on this experiment, we can confirm that a large portion of runtime is taken up by data transfers, assuming most work is done on the GPGPU. The TREE structure results in smaller relative



■ **Figure 2** Histograms of percentages of GPGPU runtime spent in `memcpy` with and without caching. Only instances of MCC2020-TRACK1 which are solved in both configurations with a runtime of at least 5s in one are considered. Results are given for the ARRAY and TREE data structure, (left) and (right), respectively.

transfer times than using ARRAY. This could be due to their smaller average size [29], longer kernel runtimes or a combination of both. Nevertheless, a large portion of instances spend a proportion of at least 40%, as well.

To alleviate this issue, we propose global caching as described above. In order to evaluate the effectiveness of our global cache, which avoids data transfers and reuses (cached) tables within the GPGPU memory, we consider the following hypothesis.

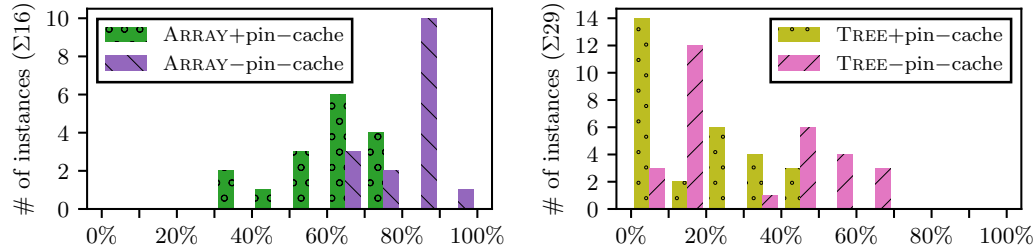
► **Hypothesis 2.** *Caching reduces the proportion of time GPUSAT3 spends on data transfers.*

We assess Hypothesis 2 with the same method as previously used for Hypothesis 1, where we run GPUSAT3 for both data structures, with and without caching. The results are presented in Fig. 2, which confirms this hypothesis: Considering the ARRAY configuration, the number of instances spending 80%–90% in `memcpy` is greatly reduced, many presumably shifting to the 70%–80% bucket. Interestingly, while without caching no instance spent less than 60% of its runtime on data transfers, a number of instances achieves using less than 10% with caching. In these cases, global caching avoids most transfers altogether. This is in-line with our expectations: While the usefulness of the global cache is degraded when all GPGPU memory is needed for solving, instances which mostly produce small tables can benefit significantly. With the TREE data structure, we see a similar trend of instances spending less time in `memcpy`. However, the effect is not as strong as with ARRAY. Again, we believe this is due to the TREE already using GPGPU memory more efficiently, leading to smaller transfers which take less time relative to kernel executions.

The Effect of Cuda Pinned Memory

To speed up data transfer between host memory and GPGPU memory, the CUDA driver offers an API that allows the use of *pinned memory* pages (also known as page-locked memory) when allocating host memory [43, 10]. Pinned memory pages reside in a fixed physical location of the host memory and cannot be moved, e.g., swapped out, by the *operating system (OS)*. This guarantee allows the CUDA driver to perform data transfers to and from these regions through its *direct memory access (DMA)* engine. Through DMA hardware, neither the CPU, nor the OS are involved in transferring data. So, no checks for the validity of memory pages through the OS kernel are needed, since physical page locations are fixed.

Pinned memory has already been utilized in the literature. Quirem et al. [45] implement a GPGPU-accelerated version of an algorithm used in the HMMER framework [30] for identifying homologous protein sequences using CUDA. The authors report a 20% speed



■ **Figure 3** Histograms of percentages of GPGPU runtime spent on data transfers (`memcpy`) grouped in steps of 10%, with and without pinned memory. Only instances of MCC2020-TRACK1 which are solved in both configurations with a runtime of at least 5s in one are considered. Results are given for both the ARRAY (left) data structure as well as the TREE (right) data structure.

up by using pinned memory. Similarly, Fatica [20] demonstrates significant performance improvements for LINPACK with pinned memory. However, allocating pinned memory incurs additional overhead compared to regular allocations of main memory [32]. Additionally, if a large portion of the physical system memory is pinned, overall performance is degraded. Moreover, when using pinned memory, the CUDA driver enforces a shared virtual address space for allocations in host memory and GPGPU memory, which is referred to as CUDA Unified Memory. Aside from pinned memory, this feature is used for handling data transfers between CPU and GPGPU memory implicitly by the driver, as a convenience for the programmer. Consequently, the overhead of unified memory applies as well. Jarzabek et al. [37] investigated the performance of unified memory, overall finding it to have only a small impact. Nevertheless, especially many small allocations increased the performance overhead. We mitigate the performance degradation of repeatedly allocating and freeing pinned memory by employing a so-called sub-allocator. This *sub-allocator* is responsible for caching pinned memory allocations, handing out memory allocations from an existing pool and only allocating additional memory when needed [36].

Evaluation

We evaluate the potential for performance improvements through achieving faster data transfers which is counteracted by the additional overhead of pinned memory allocation.

The Potential of Pinned Memory. As a first step, we consider the potential speedup of pinned memory without considering the cost for its allocation.

► **Hypothesis 3.** *With pinned memory, GPUSAT3 reduces the proportion of time spent on copying data to and from the GPGPU.*

In order to analyze this hypothesis, we measure the proportion of time the GPGPU spends for memory transfers with and without pinned memory for both the ARRAY and TREE data structures. The experiment is conducted on DESKTOP with a maximal runtime of 600s for each instance of MCC2020-TRACK1. Chunk caching, i.e., leaving solution data in GPGPU memory if possible, is disabled to measure the impact of pinned memory in isolation. The results are shown in Fig. 3 as histograms of the ratio of GPGPU runtime used for copying memory to the total GPGPU runtime, given in percent. We apply the same criteria for selecting instances for the comparison as in Sect. 3. For the ARRAY data structure (left), we see that without pinned memory, most memory transfers take up as much as 80%

to 90% of the GPGPU runtime on some instances. All instances spend at least 60% for copying memory by means of `memcpy`. With pinned memory, the majority consumes around 50% to 60% GPGPU runtime for copying, some less, with at most $\approx 80\%$. When using the TREE data structure (right) and no pinned memory, the proportion is generally lower and corresponds to the baseline distribution established in Fig. 1. With pinned memory, the distribution shifts to spend significantly less time in `memcpy`. Additionally, many instances have a copy to execution time ratio below 10%, while no instance has more than 50%. In conclusion, we see that pinned memory reduces the time used for data transfers significantly regardless of data structure. The smaller size of the TREE data structure compared to ARRAY results in a smaller proportion of GPGPU runtime spent copying.

The Benefit of Pinned Memory. For the benefit of faster memory transfers to result in improved solver performance, it needs to outweigh the overhead of pinned memory allocation.

► **Hypothesis 4.** *Employing pinned memory decreases the runtime of the solver GPUSAT3.*

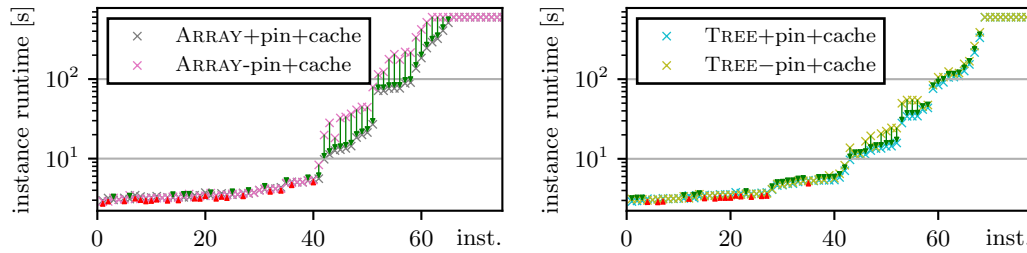
To test this hypothesis, we consider the runtime for the instances of MCC2020-TRACK1 with and without pinned memory. We run GPUSAT3 for both the ARRAY and the TREE data structure on MCC2020-TRACK1 for up to 600s on SERVER. Global caching is enabled to determine if pinned memory further improves solver runtime on top of caching. In Fig. 4, we compare the differences in runtime on a per-instance basis with an arbitrary but fixed decomposition seed. For instances where the runtime differs by more than $\geq 1\%$ of the unpinned runtime, the difference is marked by an arrow as described in the figure caption.

With both the ARRAY and TREE data structures, especially long-running instances benefit from pinned memory, while the allocation overhead amounts to an overall longer runtime on small instances. The configuration using the ARRAY data structure (left) benefits more from pinned memory on instances with long runtimes compared to the TREE (right) configuration. We attribute this to the capability of the TREE structure to grow as needed, leading to smaller transfers than with the ARRAY data structure on average. Conversely, the size of ARRAYS grows with the number of variables in a node regardless of its solution count. Moreover, fewer small instances are negatively impacted by pinned memory when using TREE, presumably because of smaller allocations incurring less overhead.

Overall, we have shown that pinned memory can speed up the solving process for instances where large memory transfers are needed. For small instances and depending on the data structure, its additional allocation overhead can outweigh the faster transfer times however. By comparing the improvement in data transfer times seen in Fig. 3 with overall solver performance in Fig. 4, we see that this mostly translates to improvements in runtime for larger instances, where the sub-allocator can serve most allocations. Thus, the addition of pinned memory is beneficial in most cases, although using the TREE data structure often mitigates the need for large memory transfers, lessening the impact of pinned memory.

4 The Influence of Decomposition Libraries

In this section, we focus on Step 2 of the dynamic programming approach, as defined in the preliminaries. Thereby, we compare several available implementations for finding tree decompositions and their effect on solver runtime in order to efficiently employ these libraries.



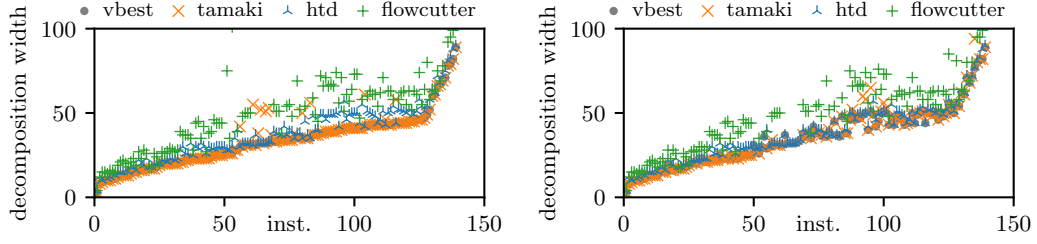
■ **Figure 4** Comparison of runtime over the instances of the MCC2020-TRACK1 data set with and without pinned memory. Downward (green) arrows denote an improvement in runtime with pinned memory, upward (red) arrows indicate a longer runtime with pinned memory. Results for the ARRAY and TREE data structure are given (left) and (right) respectively.

Finding Tree Decompositions

Quickly finding a tree decomposition with a small width is crucial for the performance of GPUSAT3. Utilizing a tree decomposition of smaller width not only improves worst-case runtime of our algorithm, but also exponentially decreases memory requirements. This is paramount for the practical efficiency: Once certain tables do not fit into the GPGPU memory, larger chunks of data have to be swapped to the host memory (RAM), thereby increasing processing time. Inconveniently, finding the treewidth of a graph represents a NP-hard problem itself [2]. An algorithm for obtaining tree decompositions of small, bounded width in linear time has been developed, but its runtime complexity contains constant factors too large for practical use [4]. To the best of our knowledge, there are no practically feasible, fast algorithms with such low time complexity. Thus, the time spent on finding a tree decomposition of low treewidth and running the dynamic programming algorithm must be balanced. To find a suitable decomposer for computing tree decompositions we compared the 3 top-ranked submissions to the heuristic competition of track A of the “Parameterized Algorithms and Computational Experiments Challenge” in 2017 (PACE17) [13]: *tamaki* by Keitaro Makii, Hiromu Ohtsuka, Takuto Sato, Hisao Tamaki (Meiji University), github.com/TCS-Meiji/PACE2017-TrackA, *flowcutter* [50] by Ben Strasser (Karlsruhe Institute of Technology), and *htd* [1] by Michael Abseher, Nysret Musliu, Stefan Woltran (TU Wien). As a first step, we compare the obtained widths and speed of the three decomposers above.

► **Hypothesis 5.** *Given a long processing time, the rank by best decomposition width reflects the placement in PACE17 in MCC2020-TRACK1: 1. tamaki, 2. flowcutter, and 3. htd.*

In Fig. 5 (left), we compare the lowest width found by the implementations in 600s for the MCC2020-TRACK1 instances on CLUSTER. For each instance and each decomposer, we obtain 10 decompositions with varying seeds. In the following analysis, the best result of these runs is considered. Although *flowcutter* ranked better than *htd* in PACE17, it consistently produces higher widths than *htd* and *tamaki* in our benchmark. Note however, that we use the more recent *htd* 1.2. The decomposer *tamaki* generates lower widths than *htd* for most instances, with some outliers. For small widths up to ≈ 30 , the difference between *htd* and *tamaki* is noticeable but small for most instances. With wider widths, *flowcutter* performs much worse than *htd* and *tamaki*. Overall, the relative performance of *htd* and *tamaki* matches our expectations of Hypothesis 5. However, *flowcutter* performs significantly worse than its competitors, which might be due to the size and structure of our instances.



■ **Figure 5** Best decomposition width for MCC2020-TRACK1 instances of different implementations; ordered by asc. best width. The plots show the widths obtained after 600s (left) and 15s (right).

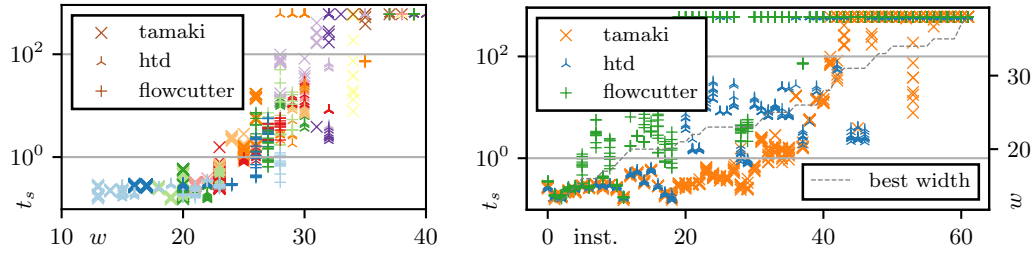
As `htd` was chosen in GPUSAT2 for being fast, we suspect the above results to change when restricting the implementations to shorter runtimes. Thus, we repeat the above analysis in Fig. 5 (right), but consider the respective best result found after only 15 seconds. GPUSAT3 with `htd` usually takes less than one second for computing the tree decomposition for most solvable instances of MCC2020-TRACK1. Nonetheless, we believe that 15s would be a realistic time budget for implementations that cannot be tightly integrated into the solver as a library. Compared with Fig. 5 (left), we see the advantage of `tamaki` over `htd` shrinking. For some instances, `htd` produces smaller decompositions than `tamaki`, which generates either a very wide or no decomposition at all. Decomposer `flowcutter` still produces the widest decompositions for most instances. Consequently, `tamaki` generates the best results for most instances, even with constrained time. However, there are cases where only `htd` produces a decomposition of usable width. Since the advantage of `tamaki` is small at low runtimes and `htd` is available as a C++ library, we keep it as the primary implementation in GPUSAT3 for convenience. In the future, a portfolio of implementations with a tuned heuristic of the time spent searching for a decomposition could yield better results [17].

The Performance Impact of Decompositions

Recall that we have already investigated the benefit of finding tree decompositions of small width, based on worst-case time and space bounds. To justify this claim, we now explore the performance impact of the chosen tree decomposition during solving in practice.

► **Hypothesis 6.** *The solving time of GPUSAT3 strongly correlates with the decomposition width and only to a lesser extent depends on the instance.*

We investigate the connection of decomposition width and solving time by creating a set of different tree decompositions for select instances. To find suitable instances, we first compute 10 tree decompositions per instance of MCC2020-TRACK1 with each of the implementations: `tamaki`, `flowcutter` and `htd`. Each run is allowed 600s of wall clock time on CLUSTER. From MCC2020-TRACK1, we select a subset of instances where we find at least one decomposition with a width between 25 and 35. These bounds are chosen because instances with a width larger than 35 are mostly unsolvable on our hardware. Conversely, if the decomposition width is too low, the solving process is usually very fast and dominated by set-up time. Thus, measuring runtime differences is not meaningful outside of this range. By applying this criterion, we obtain a set of 62 instances for our evaluation. Then, we measure the solving time for each decomposition of the selected instances with GPUSAT3 on SERVER. The version of GPUSAT3 used in this experiment already includes the improvements introduced in the course of this paper. For each instance, 10 decompositions are obtained by each

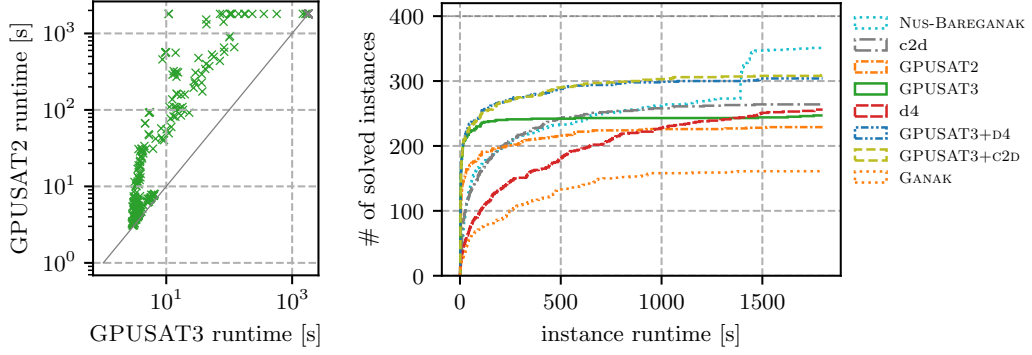


■ **Figure 6** GPUSAT3 solving time t_s in seconds by decomposition width w (left) and by instance (right). On the left, runs of the same instance are colored in the same color, but different instances may have the same color. On the right, instances are ordered by their respective lowest decomposition width, which is marked as a dashed grey line.

decomposer, amounting to 1860 decompositions in total. Since we are interested in the behaviour of the GPGPU solving algorithm rather than overall runtime compared to other solvers, we use the time spent in the solving step as reported by GPUSAT3. This time is referred to by *solving time* and excludes parsing and preprocessing steps.

In Fig. 6 (left), we show the results of this experiment as a plot of solving time and decomposition width, colored by instance. We observe a general trend of increasing solving time with larger decomposition width. Moreover, the plot can be roughly divided in three sections: Up to a width of ≈ 22 , the solving time consistently stays below one second, without major variations for different widths. For widths between ≈ 23 and ≈ 38 , solving time correlates with decomposition width. However, large differences in solving time occur among decompositions of equal widths, sometimes by multiple orders of magnitude. For larger decompositions, all runs either time out or exhaust the resources of SERVER, which is also shown as a timeout. This observation supports our focus on instances with decompositions of widths between 25 and 35 for this experiment: The runtime for small decompositions is dominated by set-up costs and parallel GPGPU resources are not saturated, thus it does not vary significantly. For decompositions of high width, resource and time limits are quickly exceeded. While this experiment clearly demonstrates a correlation between decomposition width and solving time as indicated by theoretical bounds, there is a large spread of runtimes for decompositions of the same width. For example, runtimes for the width 28 span from sub-1s to almost 100s. When looking at the distribution of instances, which are indicated by color, we see that runs of the same instance often form clusters. Through the different marker symbols, we see that distinct clusters of the same color mostly originate from different composers. Conversely, most clusters only contain runs of one decomposer and instance. This indicates that the variance in runtime is low for the same instance and decomposer. However, this view does not immediately reveal solving times of all runs of the same instance.

Thus, we visualize this perspective in Fig. 6 (right). The plot shows solving times for every generated decomposition of each instance as specified in the figure caption. Similarly to the by-width perspective Fig. 6 (left), runtimes for decompositions generated by the same decomposer often appear clustered. However, this effect appears less pronounced for decompositions generated by *flowcutter*. For many instances, *tamaki* appears to generate the best-performing decompositions. This is in line with our findings above. Some instances, especially those of smaller width, show very similar runtimes for all decompositions. However, in most cases, runtimes are clearly separated for the decomposers. As instances are sorted by their best generated decomposition width, a trend of increasing runtime with width is visible.



■ **Figure 7** Performance of GPUSAT3 compared GPUSAT2 and other state-of-the-art systems. In the cactus plot (right) instances are sorted for each solver individually, according to ascending runtime. The scatter plot (left) compares instance runtimes of GPUSAT3 with GPUSAT2.

This indicates that the width of a decomposition is a better predictor for solver runtime than the given instance, as stated in Hypothesis 6. Nonetheless, runtimes vary among decompositions of the same width, so treewidth is not the sole estimate for instance hardness of GPUSAT3. Thus, detailed structural studies are left for further research, cf. [41].

5 Overall Results

Next, we evaluate the combined result of the presented techniques above by comparing the performance of GPUSAT3 to GPUSAT2, D4 [39], C2D [11], NUS-BAREGANAK [22], and GANAK [49]. The systems D4, C2D, and NUS-BAREGANAK are among the best solvers of the Model Counting Competition 2020 [22] (MCC 2020). At time of submission, the full instance set of the 2021 competition was not publicly available [21]. D4 and C2D are based on knowledge compilation, while NUS-BAREGANAK is a portfolio of solvers: First, they run the B+E preprocessor [38], followed by GANAK. If GANAK does not produce a result in a chosen timeout, APPROXMC [7] is used. As no external preprocessing is used with the other solvers, pure GANAK is included for reference. We run all solvers for up to 1800s for each instance of MCC2020-TRACK1+2 on SERVER. All solvers are used in their default configuration: C2D and D4 ran with flags to enable counting, otherwise no additional arguments were supplied. The number of solved instances, ordered by instance runtime, is shown in Fig. 7 (right).

The total number of solved instances per solver is listed in Tab. 1. For low runtimes, GPUSAT3 establishes a clear lead over the other solvers. Given more time, GPUSAT2 approaches GPUSAT3 and D4, C2D surpass GPUSAT3. NUS-BAREGANAK delivers similar performance to C2D until APPROXMC is used, where it surpasses the other solvers. Note that the results of APPROXMC are within $\pm 30\%$ of the correct count with 80% probability [7].

In practice, not only the number of eventually solved instances is relevant, but also the time in which they are solved. Thus, we define a *baseline* set of benchmarks, which are the instances that are solved by all solvers except pure GANAK, thereby enabling meaningful comparisons of solving time. GANAK is excluded to maintain a meaningful baseline set size. Tab. 1 lists the accumulated runtime for each solver, for 50%, 90%, 95% and 100% of instances of both the baseline set and all of a solver’s respective solved instances. When comparing with respect to the baseline set, we obtain an 8x speedup in accumulated runtime of GPUSAT3 over GPUSAT2, 11x over C2D, and over 10x speedup over NUS-BAREGANAK.

■ **Table 1** Solved instances and accumulated runtimes for the fastest $n\%$ of solved instances for each surveyed solver. In the second row for each solver, accumulated runtime is compared with respect to a baseline set of 161 instances, which are solved by all except GANAK. * the portfolio includes an approximate solver, for more details see above.

Solver	# inst.	$\sum t$ 100%	$\sum t$ 95%	$\sum t$ 90%	$\sum t$ 50%
GPUSAT2	229	5:56:06	3:05:47	1:51:28	0:09:13
... on baseline	161	2:26:36	1:00:06	0:36:32	0:06:38
GPUSAT3	247	3:05:58	0:43:40	0:27:12	0:07:01
... on baseline	161	0:18:08	0:13:00	0:11:03	0:04:51
D4	256	1 day, 2:30:28	20:57:09	16:39:14	1:59:09
... on baseline	161	15:54:58	12:09:37	9:21:47	0:53:41
C2D	265	12:25:56	8:20:19	6:21:38	0:39:15
... on baseline	161	3:29:07	2:16:12	1:40:38	0:13:16
NUS-BAREGANAK	351*	1 day, 21:47:59	1 day, 14:21:25	1 day, 7:41:05	1:33:55
... on baseline	161	3:12:16	1:57:25	1:30:17	0:17:53
GANAK	161	11:26:48	9:05:15	7:28:31	0:53:24
GPUSAT3+D4	304	7:36:44	3:36:43	1:58:31	0:09:05
... on baseline	161	0:18:08	0:13:00	0:11:03	0:04:51
GPUSAT3+C2D	309	8:45:15	4:30:15	2:35:57	0:09:23
... on baseline	161	0:18:08	0:13:00	0:11:03	0:04:51

To combine the capability of D4 and C2D with the speed of GPUSAT3, we define the portfolios GPUSAT3+D4 and GPUSAT3+C2D to use GPUSAT3 for instances with a decomposition width of ≤ 35 and D4 resp. C2D otherwise. The time to calculate the width is negligible and therefore not included in the runtime of instances solved with the portfolio solvers. To generate the decomposition we use `htd` as used in GPUSAT3. As shown in Tab. 1, the portfolio solvers are very successful: Not only do they solve significantly more instances than D4 and C2D alone, but accomplish this in significantly less accumulated runtime. As expected, GPUSAT3+D4 and GPUSAT3+C2D achieve the same performance on the baseline set as GPUSAT3, which overall either solves instances extremely fast or fails. With the availability of advanced hardware with larger GPGPU memory, we expect that due to global caching, GPUSAT3 solves instances that currently reach a timeout. External preprocessing as used in NUS-BAREGANAK could further improve the results, cf. [29].

6 Conclusion and Future Work

Efficiently solving problems related to propositional model counting is critical for a range of applications such as probabilistic reasoning. To accelerate solving, the massively parallel computing capabilities of general purpose GPUs (GPGPUs) can be leveraged by a dynamic programming based algorithm. Our system GPUSAT3 builds on top of ideas from GPUSAT2, where we implement algorithmic improvements and techniques for better hardware utilization. We describe a new, hardware-friendly compact clause form, a global caching strategy, as well as pinned memory, and systematically evaluate impacts on solving performance. Additionally, we survey a range of libraries for generating tree decompositions

and show their performance impact as well. Compared to GPUSAT2, our overall results show that GPUSAT3 solves all instances faster, sometimes by an order of magnitude. While GPUSAT3 is designed for bounded treewidth, it complements D4 and C2D in a portfolio approach; significantly enhancing the overall performance compared to the individual solvers.

In the future, we plan on migrating portions of the solver code to GPGPU kernel code. Additionally, we are interested in the scalability of GPUSAT3 when using multiple GPGPUs. To the best of our knowledge, currently there is no way to limit the number of parallel compute cores a program can utilize, preventing such experiments with a single device. Alternatives include frequency and voltage scaling [31] and dynamic transformation to a CPU program [19], both options have probably different scaling characteristics than the addition of parallel compute cores. Furthermore, techniques used in other dynamic programming based solvers such as ADDMC [15] could be brought to the GPGPU. Conversely, integrating GPUSAT3 into other solvers for solving sub-problems of small treewidth might be beneficial. Finally, GPGPU-based approaches might be applicable to formalisms such as argumentation [23], logic programming [33], or description logics [24], despite strong theoretical limits [25].

References

- 1 Michael Abseher, Nysret Musliu, and Stefan Woltran. htd - A free, open-source framework for (customized) tree decompositions and beyond. In Domenico Salvagnin and Michele Lombardi, editors, *Proceedings of the 14th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2017), Padua, Italy, June 5-8, 2017*, volume 10335 of *Lecture Notes in Computer Science*, pages 376–386. Springer, 2017. doi:10.1007/978-3-319-59776-8_30.
- 2 Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987. doi:10.1137/0608024.
- 3 Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and complexity results for #SAT and bayesian inference. In *Proceedings of the 44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA*, pages 340–351. IEEE Computer Soc., 2003. doi:10.1109/SFCS.2003.1238208.
- 4 Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996. doi:10.1137/S0097539793251219.
- 5 Jan Burchard, Tobias Schubert, and Bernd Becker. Laissez-faire caching for parallel #SAT solving. In Marijn Heule and Sean A. Weaver, editors, *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT 2015), Austin, TX, USA, September 24-27, 2015*, volume 9340 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2015.
- 6 Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. Distribution-aware sampling and weighted model counting for SAT. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI 2014), July 27 -31, 2014, Québec City, Québec, Canada*, pages 1722–1730. AAAI Press, 2014. URL: <http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8364>.
- 7 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls. In Subbarao Kambhampati, editor, *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016), New York, NY, USA, 9-15 July 2016*, pages 3569–3576. IJCAI/AAAI Press, July 2016. URL: <http://www.ijcai.org/Abstract/16/503>.
- 8 Günther Charwat and Stefan Woltran. Expansion-based QBF solving on tree decompositions. *Fundamenta Informaticae*, 167(1-2):59–92, 2019. doi:10.3233/FI-2019-1810.

- 9 Arthur Choi, Guy Van den Broeck, and Adnan Darwiche. Tractable learning for structured probability spaces: A case study in learning preference distributions. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, Buenos Aires, Argentina, July 25-31, 2015, pages 2861–2868. AAAI Press, 2015. URL: <http://ijcai.org/Abstract/15/405>.
- 10 Shane Cook. *CUDA programming: A developer's guide to parallel computing with GPUs*. Applications of GPU Computing Series. Morgan Kaufmann, Boston, 2013. doi:10.1016/B978-0-12-415933-4.02001-9.
- 11 Adnan Darwiche. New advances in compiling CNF to decomposable negation normal form. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI 2004)*, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004, pages 318–322. IOS Press, 2004.
- 12 Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In Toby Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, Barcelona, Catalonia, Spain, July 16-22, 2011, pages 819–826. AAAI Press/IJCAI, 2011. doi:10.5591/978-1-57735-516-8/IJCAI11-143.
- 13 Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The PACE 2017 parameterized algorithms and computational experiments challenge: The second iteration. In Daniel Lokshantov and Naomi Nishimura, editors, *Proceedings of the 12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*, September 6-8, 2017, Vienna, Austria, volume 89 of *LIPIcs*, pages 30:1–30:12. Dagstuhl Publishing, 2017. doi:10.4230/LIPIcs.IPEC.2017.30.
- 14 Carmel Domshlak and Jörg Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research*, 30:565–620, 2007. doi:10.1613/jair.2289.
- 15 Jeffrey M. Dudek, Vu Phan, and Moshe Y. Vardi. ADDMC: weighted model counting with algebraic decision diagrams. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020)*, New York, NY, USA, February 7-12, 2020, pages 1468–1476. AAAI Press, 2020. URL: <https://aaai.org/ojs/index.php/AAAI/article/view/5505>.
- 16 Jeffrey M. Dudek, Vu H. N. Phan, and Moshe Y. Vardi. DPMC: weighted model counting by dynamic programming on project-join trees. In Helmut Simonis, editor, *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP 2020)*, Louvain-la-Neuve, Belgium, September 7-11, 2020, volume 12333 of *Lecture Notes in Computer Science*, pages 211–230. Springer, 2020. doi:10.1007/978-3-030-58475-7_13.
- 17 Jeffrey M. Dudek and Moshe Y. Vardi. Parallel weighted model counting with tensor networks. *CoRR*, abs/2006.15512, 2020. arXiv:2006.15512.
- 18 Leonardo Dueñas-Osorio, Kuldeep S. Meel, Roger Paredes, and Moshe Y. Vardi. Counting-based reliability estimation for power-transmission grids. In Satinder P. Singh and Shaul Markovitch, editors, *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI 2017)*, February 4-9, 2017, San Francisco, California, USA, pages 4488–4494. AAAI Press, 2017. URL: <http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14870>.
- 19 Naila Farooqui, Andrew Kerr, Gregory Frederick Diamos, Sudhakar Yalamanchili, and Karsten Schwan. A framework for dynamically instrumenting GPU compute applications within GPU ocelot. In *Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2011)*, Newport Beach, CA, USA, March 5, 2011, page 9. ACM, 2011. doi:10.1145/1964179.1964192.
- 20 Massimiliano Fatica. Accelerating linpack with CUDA on heterogenous clusters. In David R. Kaeli and Miriam Leeser, editors, *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2009)*, Washington, DC, USA, March 8, 2009, volume 383 of *ACM International Conference Proceeding Series*, pages 46–51. ACM, 2009. doi:10.1145/1513895.1513901.
- 21 Johannes K. Fichte and Markus Hecher. The model counting competition 2021. https://mcccompetition.org/past_iterations, 2021.

- 22 Johannes K. Fichte, Markus Hecher, and Florim Hamiti. The model counting competition 2020. *ACM Journal of Experimental Algorithmics*, 2021. In press.
- 23 Johannes K. Fichte, Markus Hecher, and Arne Meier. Counting complexity for reasoning in abstract argumentation. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI'19)*, Honolulu, Hawaii, USA, 2018.
- 24 Johannes K. Fichte, Markus Hecher, and Arne Meier. Knowledge-base degrees of inconsistency: Complexity and counting. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI'21)*, pages 6349–6357, 2021.
- 25 Johannes K. Fichte, Markus Hecher, and Andreas Pfandler. Lower Bounds for QBFs of Bounded Treewidth. In Naoki Kobayashi, editor, *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'20)*, pages 410–424. Assoc. Comput. Mach., New York, 2020.
- 26 Johannes K. Fichte, Markus Hecher, and Valentin Roland. GPUSAT3 benchmark data and source code. *Zenodo*, 2021. doi:10.5281/zenodo.5159903.
- 27 Johannes K. Fichte, Markus Hecher, Patrick Thier, and Stefan Woltran. Exploiting database management systems and treewidth for counting. In Ekaterina Komendantskaya and Y. Annie Liu, editors, *Proceedings of the 22nd International Symposium on Practical Aspects of Declarative Languages (PADL'20)*, volume 12007 of *Lecture Notes in Computer Science*, pages 151–167. Springer, 2020. doi:10.1007/978-3-030-39197-3_10.
- 28 Johannes K. Fichte, Markus Hecher, Stefan Woltran, and Markus Zisser. Weighted model counting on the GPU by exploiting small treewidth. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *Proceedings of the 26th Annual European Symposium on Algorithms (ESA 2018)*, August 20–22, 2018, Helsinki, Finland, volume 112 of *LIPIcs*, pages 28:1–28:16. Dagstuhl Publishing, 2018. doi:10.4230/LIPIcs.ESA.2018.28.
- 29 Johannes K. Fichte, Markus Hecher, and Markus Zisser. An improved GPU-based SAT model counter. In Thomas Schiex and Simon de Givry, editors, *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP 2019)*, Stamford, CT, USA, September 30 - October 4, 2019, volume 11802 of *Lecture Notes in Computer Science*, pages 491–509. Springer, 2019. doi:10.1007/978-3-030-30048-7_29.
- 30 Robert D. Finn, Jody Clements, and Sean R. Eddy. HMMER web server: interactive sequence similarity searching. *Nucleic Acids Research*, 39(Web-Server-Issue):29–37, 2011. doi:10.1093/nar/gkr367.
- 31 Rong Ge, Ryan Vogt, Jahangir Majumder, Arif Alam, Martin Burtscher, and Ziliang Zong. Effects of dynamic voltage and frequency scaling on a K20 GPU. In *Proceedings of the 42nd International Conference on Parallel Processing (ICPP 2013)*, Lyon, France, October 1–4, 2013, pages 826–833. IEEE Computer Society, 2013. doi:10.1109/ICPP.2013.98.
- 32 GNU Project. GNU libc manual, 3.5.2 locked memory details. URL: https://www.gnu.org/software/libc/manual/html_node/Locked-Memory-Details.html.
- 33 Markus Hecher. Treewidth-aware reductions of normal ASP to SAT - is normal ASP harder than SAT after all? In Diego Calvanese, Esra Erdem, and Michael Thielscher, editors, *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12–18, 2020*, pages 485–495, 2020. doi:10.24963/kr.2020/49.
- 34 Markus Hecher, Patrick Thier, and Stefan Woltran. Taming high treewidth with abstraction, nested dynamic programming, and database technology. In Luca Pulina and Martina Seidl, editors, *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT 2020)*, Alghero, Italy, July 3–10, 2020, volume 12178 of *Lecture Notes in Computer Science*, pages 343–360. Springer, 2020. doi:10.1007/978-3-030-51825-7_25.
- 35 Marijn Heule and Hans van Maaren. Parallel SAT solving using bit-level operations. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):99–116, 2008. doi:10.3233/sat190040.

- 36 Jared Hoberock, Nathan Bell, and Thrust Contributors. Thrust API documentation. URL: https://thrust.github.io/doc/classthrust_1_1mr_1_1disjoint__unsynchronized__pool__resource.html.
- 37 Lukasz Jarzabek and Pawel Czarnul. Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications. *Journal of Supercomputing*, 73(12):5378–5401, 2017. doi:10.1007/s11227-017-2091-x.
- 38 Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Improving model counting by leveraging definability. In Subbarao Kambhampati, editor, *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, New York, NY, USA, 9-15 July 2016, pages 751–757. The AAAI Press, July 2016. URL: <http://www.ijcai.org/Abstract/16/112>.
- 39 Jean-Marie Lagniez and Pierre Marquis. An improved decision-DDNF compiler. In Carles Sierra, editor, *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, Melbourne, Australia, August 19-25, 2017, pages 667–673. The AAAI Press, 2017. doi:10.24963/ijcai.2017/93.
- 40 Jean-Marie Lagniez, Pierre Marquis, and Nicolas Szczepanski. DMC: a distributed model counter. In Jérôme Lang, editor, *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*, July 13-19, 2018, Stockholm, Sweden, pages 1331–1338. The AAAI Press, 2018. doi:10.24963/ijcai.2018/185.
- 41 Silviu Maniu, Pierre Senellart, and Suraj Jog. An experimental study of the treewidth of real-world graph data. In Pablo Barceló and Marco Calautti, editors, *Proceedings of the 22nd International Conference on Database Theory (ICDT 2019)*, March 26-28, 2019, Lisbon, Portugal, volume 127 of *LIPIcs*, pages 12:1–12:18. Dagstuhl Publishing, 2019. doi:10.4230/LIPIcs.ICDT.2019.12.
- 42 Sheila A. Mui, Christian J. and McIlraith, J. Christopher Beck, and Eric I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. In Leila Kosseim and Diana Inkpen, editors, *Proceedings of the 25th Canadian Conference on Advances in Artificial Intelligence Artificial Intelligence (Canadian AI 2012)*, Toronto, ON, Canada, May 28-30, 2012, volume 7310 of *Lecture Notes in Computer Science*, pages 356–361. Springer, 2012. doi:10.1007/978-3-642-30353-1_36.
- 43 NVIDIA Corporation. Application note – CUDA 2.2 pinned memory APIs. URL: <http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/simpleZeroCopy/doc/CUDA2.2PinnedMemoryAPIs.pdf>.
- 44 Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, Buenos Aires, Argentina, July 25-31, 2015, pages 3141–3148. The AAAI Press, 2015. URL: <http://ijcai.org/Abstract/15/443>.
- 45 Saddam Quireem, Fahian Ahmed, and Byeong Kil Lee. CUDA acceleration of P7Viterbi algorithm in HMMER 3.0. In Sheng Zhong, Dejing Dou, and Yu Wang, editors, *Proceedings of the 30th IEEE International Performance Computing and Communications Conference (IPCCC 2011)*, Orlando, Florida, USA, November 17-19, 2011, pages 1–2. IEEE, 2011. doi:10.1109/PCCC.2011.6108104.
- 46 Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996. doi:10.1016/0004-3702(94)00092-1.
- 47 Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50–64, 2010. doi:10.1016/j.jda.2009.06.002.
- 48 Tian Sang, Paul Beame, and Henry A. Kautz. Performing bayesian inference by weighted model counting. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings of the 20th National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, July 9-13, 2005, Pittsburgh, Pennsylvania, USA, pages 475–482. AAAI Press / The MIT Press, 2005. URL: <http://www.aaai.org/Library/AAAI/2005/aaai05-075.php>.
- 49 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In Sarit Kraus, editor, *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, Macao, China, August 10-16, 2019, pages 1169–1176, 2019. doi:10.24963/ijcai.2019/163.

- 50 Ben Strasser. Computing tree decompositions with flowcutter: PACE 2017 submission. *CoRR*, abs/1709.08949, 2017. [arXiv:1709.08949](#).
- 51 Marc Thurley. sharpSAT - counting models with advanced component caching and implicit BCP. In Armin Biere and Carla P. Gomes, editors, *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT 2006)*, Seattle, WA, USA, August 12-15, 2006, volume 4121 of *Lecture Notes in Computer Science*, pages 424–429. Springer, 2006. doi:10.1007/11814948_38.
- 52 Richard Vuduc and Jee Choi. *A Brief History and Introduction to GPGPU*, pages 9–23. Springer US, Boston, MA, 2013. doi:10.1007/978-1-4614-8745-6_2.

Complications for Computational Experiments from Modern Processors

Johannes K. Fichte ✉ 

TU Dresden, Germany

Markus Hecher ✉ 

TU Wien, Austria

Universität Potsdam, Germany

Ciaran McCreesh ✉

University of Glasgow, UK

Anas Shahab ✉

TU Dresden, Germany

Abstract

In this paper, we revisit the approach to empirical experiments for combinatorial solvers. We provide a brief survey on tools that can help to make empirical work easier. We illustrate origins of uncertainty in modern hardware and show how strong the influence of certain aspects of modern hardware and its experimental setup can be in an actual experimental evaluation. More specifically, there can be situations where (i) two different researchers run a reasonable-looking experiment comparing the same solvers and come to different conclusions and (ii) one researcher runs the same experiment twice on the same hardware and reaches different conclusions based upon how the hardware is configured and used. We investigate these situations from a hardware perspective. Furthermore, we provide an overview on standard measures, detailed explanations on effects, potential errors, and biased suggestions for useful tools. Alongside the tools, we discuss their feasibility as experiments often run on clusters to which the experimentalist has only limited access. Our work sheds light on a number of benchmarking-related issues which could be considered to be folklore or even myths.

2012 ACM Subject Classification Computer systems organization → Multicore architectures; Hardware → Temperature monitoring; Hardware → Impact on the environment; Hardware → Platform power issues; Theory of computation → Design and analysis of algorithms

Keywords and phrases Experimenting, Combinatorial Solving, Empirical Work

Digital Object Identifier 10.4230/LIPIcs.CP.2021.25

Supplementary Material Benchmark results can be found on Zenodo <https://doi.org/10.5281/zenodo.5542156>.

Funding Johannes K. Fichte: Google Fellowship at the Simons Institute, UC Berkeley.

Markus Hecher: FWF Grants Y698 and P32830 and Grant WWTF ICT19-065.

Acknowledgements Authors are given in alphabetical order. The work has been carried out while the first three authors were visiting the Simons Institute for the Theory of Computing.

1 Introduction

“Why trust science?” is the title of a recent popular science book by Naomi Oreskes [74]. We can ask the same question of combinatorial sciences, algorithms, and evaluations: *Why trust an empirical experiment?* Roughly speaking, in science, we try to understand why things happen in the real world and investigate them with the help of scientific methods. One important aspect to make an empirical evaluation trustworthy is reproducibility. This topic has been the subject of much recent scrutiny, with some arguing there is a reproducibility



© Johannes K. Fichte, Markus Hecher, Ciaran McCreesh, and Anas Shahab;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 25; pp. 25:1–25:21

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

crisis in areas fields of computer science [33, 31] and even a replicability crisis in other scientific fields of research [81]. Luckily in combinatorial problem solving, replicability is often already indirectly addressed in public challenges, which many combinatorial solving communities organize in order to foster implementations and evaluations [47, 82, 80, 83, 86, 20, 21]. The challenges provide a place for empirical evaluations, feature shared benchmarks, and support long-term heritage [3, 17, 22, 54]. It is therefore often assumed that everything should be judged with respect to these benchmarks or latest solvers [5]. However, benchmarks featured in competitions are not necessarily robust [40, 49] and might bias towards existing solving approaches and heuristics. On that account, one can argue that non-competitive evaluations are quite helpful for papers that are orthogonal to classical improvements over one particular solving technique or algorithm [19, 30]. There, one can often see a strong focus on algorithm engineering and their evaluation [63], which might not always be desired from a theoretical perspective. In particular, it makes reproducibility far less obvious than one would expect from theory. While reproducibility initiatives are becoming fashionable [73, 59], aspects are often left out in practical algorithm engineering and when testing combinatorial implementations: (i) the test-setup is not given (*no protocol*) or error prone (*no failure analysis/considerations*), (ii) modern hardware is simplified to the von-Neumann model (Princeton architecture) [87] and considered deterministic, and (iii) underlying software is neglected.

In this work, we summarize a list of topics to consider that might be folklore to an experienced engineer, but are often only mentioned between the lines while being crucial to actual reproducibility (Section 2.1). We include a list of *system and environment parameters* that are impactful when carrying out empirical work (Section 2.3). We summarize useful tools and list *practical problems* that repeatedly occur when experimenting (Sections 3.1, 3.2, and 3.3). In the main part of our paper, we provide an initial *list of issues* caused by modern consumer hardware that can have a notable impact if setup and configurations are not carefully designed (Table 1). We show by example that *one can achieve different results* in the number of solved instances ranging from 5%–40% on the same hardware, depending on the setup (Experiment 1, Table 3). This could suggest that it is not always meaningful to only prefer solvers that beat the “best” solver, but to aim for *clean benchmark settings* and elaborate discussions that highlight both the solver’s advantages and disadvantages.

Related Works. There are various works that address aspects of reproducibility [3, 7, 8, 15, 16, 56, 79, 94, 98] and experimental design [39, 63, 71], including micro-benchmarking, which requires special attention in terms of statistical analysis [42], [71, Ch.8]. Previous works neglected effects of modern parallel hardware on experimenting and some aspects have only been addressed in the background by the community. We put attention on certain issues arising on modern machines, updating outdated assumptions on measures, and illustrating how certain problems can be omitted. In the sequel, we revisit some of these related works.

2 Evaluating Combinatorial Algorithms

Natural sciences have a long tradition in designing experiments (DOE). Practical experiments date back to the ancient Greek philosophers such as Thales and Anaximenes with empirically verifiable ideas. There, *methodology is key* and has a long tradition with formal approaches existing since the late 1920s [25, 26]. Methodology not only involves the experiment itself, but also observation, measurement, and the design of test aiming to reduce external influence. Already in 1995, Hooker [39] discussed challenges in competitive evaluations of heuristics. A variety of these challenges are still of relevance in today’s combinatorial solving community.

In particular, emphasis on competitions tells which algorithms/implementations are better, but not *why*; this remains a particularly big challenge in the SAT community [27, 91]. If a novel implementation wins, it is accepted; otherwise, it is considered as failure, resulting in a high incentive to find the best possible parameter settings. The challenge of designing an experiment (DOE) has meanwhile been addressed by a more experimental community in broad guides [63] or algorithm engineering works [71]. In contrast, theoretical computer scientists often neglect environmental considerations due to the assumption that modern hardware behaves similar to simplified mathematical machine models [90] or classical hardware models [93]. Unfortunately, this is no longer the case for modern architectures. There is a list of concepts and external influences that can interfere, some of which are discussed below.

2.1 Repeatability, Replicability, and Reproducibility

When conducting a study or experiment, a central goal is to reduce inconsistencies between theoretical descriptions and actual experiments. Three major principles play a central role: repeatability, replicability, and reproducibility. Unsurprisingly, these topics are also critically discussed in other scientific fields [65] and sometimes confused with each other.

Repeatability requires repeating a computation by the same researcher with the same equipment at reliably the same result. The main purpose is often to estimate random errors inherent in any observation. When evaluating combinatorial solvers, repeatability translates to running the same solver with the same configuration on a given instance multiple times, maybe even on different hardware. Some publicly accessible evaluation platforms for combinatorial competitions address repeatability to a certain extent, as for example, StarExec [84] and Optil.io [94]. Effective tools to measure and control the execution of combinatorial solvers are runsolver [79] and BenchExec [7, 8].

Replicability, sometimes also called method reproducibility, refers to the principle that if an experiment is replicated by independent researchers with access to the original artifacts and same methodology, that then outcomes are the same with high confidence. When evaluating combinatorial solvers, replicability translates to running the same solver with the same configuration and instance on a different system by independent researchers, which is sometimes also called *recomputability*. Relevant aspects relate to works such as the Heritage projects [3, 16, 15], which preserve access to old solvers and making sources accessible to a broad community, or Singularity, which aims for easy an setup on high-performance computing (HPC) systems with few prerequisites on the environment [56, 98]. Another initiative (Guix) aims for a dedicated Linux distribution that provides highly stable system dependency configurations [1, 96]. Already in 2013, the recomputation manifesto postulated that one can only build on previous work if it can properly be replicated as a first step [28]. In addition, it makes research more efficient, similarly to how high quality publications can benefit other researchers. In contrast, some researchers argue that replicability is not worth considering, since sharing all artifacts is a non-trivial activity, which in consequence wastes efforts of the researchers [18]. Still, replicability is getting solid attention within the experimental algorithmics community [73], since it supports quality assurance.

Reproducibility aims for being able to obtain the same outcome using artifacts, which independent researchers develop without help of the original authors. When evaluating combinatorial solvers reproducibility roughly refers to another group constructing a second solver that implements the same algorithmic ideas. For example, Knuth re-implemented SAT algorithms from several epochs [52]. More experimental directions are investigations into robustness of benchmark sets and their evaluation measures [49]. Reproducibility can also be interpreted fairly vaguely [32]. Interestingly, the literature on experimental setup [63] and algorithm engineering [71] already contains a variety of suggestions to obtain reproducibility.

Repeatability and to some extent also replicability are the focus of our paper. Our aim is to make researchers aware of potential problems caused by modern computer systems, illustrate how to detect and reduce them without spending hours of debugging or over-valuing small improvements. Before we go into details, we briefly discuss principles and tools that support both repeatability and replicability. Since recent works on reproducibility provide various helpful suggestions on replicability in terms of environment [75], we focus only on aspects that might degrade long-term repeatability and replicability, resulting in over-engineering or over-tooling. We argue in favour of reviving an old Unix philosophy: build simple, short, clear, modular, and extensible code [64] both for the actual solver as well as the evaluation. Always keep dependencies low and provide a statically linked binary along with your code [89, 60] or a simple virtual environment to reproduce dependencies if you use interpreted languages. Even if the source code does not compile with newer versions, binary compatibility is mostly maintained for decades. The primary focus of container-based solutions, such as Singularity [56], is current accessibility of scientific computing software that requires extensive libraries and complex environments. It is quite useful if the software is widely used, requires complicated setup on high performance computing environments, and is continuously maintained. Container-based solutions can also be useful for building source code on old operating systems [3]. However, they introduce additional dependencies, increase conceptual complexity, can have notable runtime overhead under certain conditions [100], require additional work for a proper setup (both hosts as well as containers), and increase chances that the software does not out run of the box in 3 years. A practical observation illustrates this quite well: already since 2010, a meta software (Vagrant) tries to wrap providers such as VirtualBox, Hyper-V, Docker, VMWare, or AWS. While virtualization can be tempting to use, chances are high that some of these providers upgrade functionality or disappear entirely resulting in useless migration efforts.

2.2 An Experiment

In the beginning of Section 2, we stated classical experimental viewpoints: fixed solver or fixed instance set. In contrast, we take a third perspective by fixing both the instance set and the solver and focus on differences in hardware configurations. Therefore, we turn our attention to a recent experiment on SAT solvers (time leap challenge) [22]. We repeat the experiment with the solver *CaDiCa1* on other hardware to investigate side effects of experimental setup and hardware. Also, we use *set-asp-gauss* as instances, which contains 200 publicly available SAT instances from a variety of domains with increasing practical hardness [40]¹. We take a timeout of 900 seconds, but would like to point out that recent SAT competitions restrict the total runtime over all instances to 5,000 seconds. We run experiments on the following environments: COMET LAKE (i7 GEN10): Intel i7-10710U 4.7 GHz, Linux 5.4.0-72-generic, Ubuntu 20.04; HASWELL (XEON GEN4): 2x Intel Xeon E5-2680v3 CPUs, Linux 3.10.0-1062, RHEL 7.7; ROME (ZEN2): 2x AMD EPYC 7702, Linux 3.10.0-1062, RHEL 7.7; and SKYLAKE (XEON GEN6): Xeon Silver 4112 CPU, Linux version 4.15.0-91, Mint 19. We explicitly include cheap mobile hardware by using a COMET LAKE (i7 GEN10) CPU, since not every group can afford expensive server hardware or spend valuable research time on setting up stable experiments on a cluster.

¹ The benchmark set is available for download at <https://www.cs.uni-potsdam.de/wv/projects/sets/set-industrial-09-12-gauss.tar.xz>

■ **Table 1** Number of solved instances out of 200 SAT instances running the solver **CaDiCal** on varying platforms. Column $s(15)$ contains the number of solved instances when timeout is 15 minutes; f and p refer to the CPU frequency in GHz and number of solvers running in parallel, respectively. The t column contains the total runtime in hours for all instances solved within 15 minutes.

Processor (CPU)	f	p	$s(15)$	$t[h]$
SKYLAKE	3.0	1	190	5.12
HASWELL	3.3	1	189	3.89
ROME	3.4	1	190	3.79
COMET LAKE	4.7	1	191	3.81
COMET LAKE	4.7	6	189	6.13
COMET LAKE	4.7	12	176	7.18

Table 1 illustrates the results of the experiment on varying hardware. Unsurprisingly, the modern hardware running at 4.7 GHz solves the most instances. Somewhat unexpected is that two potentially faster processors solve fewer instances. Namely, the Rome CPU which is faster than the Skylake CPU solves fewer instances and similarly the Haswell solves fewer instances than the Skylake. Since both processors are different generations one might expect that the AMD CPU is simply slower. While the 5% fewer solved instances might seem not much comparing the results to the ones of the time leap challenge, it would mean that a ten year old solver solves almost the same number of instances on a modern hardware as **CaDiCal** on very recent hardware. Below, we explain that this is clearly not the case and illustrate details of the experimental setup that contribute to the low number of solved instances. In contrast, when comparing the number of solved instances for the COMET LAKE configurations, it is obvious to an experienced reader that while the COMET LAKE CPU exposes 12 software cores, due to multithreading (MT) only 6 physical cores are available. Still, when using all physical cores, we have more than 30% higher runtime, which can be particularly problematic when comparing to settings that prefer total runtime as measure. To avoid this, we could simply not run solvers in parallel; however, this seems quite impractical and inefficient. In the following sections, we clear up which problems in the setup may have caused the differences and illustrate how to avoid such issues.

2.3 Uncertainty on Modern Hardware

Modern processors can do many calculations at the same time by using multiple cores on each processor and each core also has built-in a certain parallelism. While this can be exploited explicitly in terms of parallel programming frameworks, some features are already done by on-board circuits or firmware, which is a low level software layer between the CPU hardware and the operating system. Compile time optimizations such as *automated parallel execution optimization* and *cache performance optimization* [4] can then automatically employ specific features. For example, *loop optimization* tries to automatically rewrite loops in programs such that the loop can be executed in parallel on multiprocessor systems. Scheduling splits loops so that they can run concurrently on multiple processors. Vectorization optimizes for running many loop iterations on parallel hardware that supports single instruction multiple data (SIMD). Therefore, instead of processing a single element of a vector N times, m elements of a vector are processed simultaneously N/m times. In fact, modern CPUs have so-called *vector instruction sets* such as SSE, AVX, NEON, or SVE depending on the architecture, which makes them SIMD hardware. Loop vectorization can have a significant impact on

the runtime due to effects on pipeline synchronization or data-movement timing. Usually, dependency analysis tries to optimize these operations. But depending on the compiler (GCC, Intel, or LLVM) different runtimes of the resulting binary can be observed [92].

Processor specific features add to less pre-calculable behavior. *Turbo Boost*, which was introduced around 2008, allows to dynamically overclock the CPU if the operating system requests the highest performance state of the processor [66]. *Thermal design power (TDP)*, which was established around 2012, allows to scale the power (energy transfer rate) variably between 50W and 155W [70] to save energy depending on the system load. In particular, this is active on laptop systems that are not connected to an electrical outlet or if certain system sensors detect high temperature. *Turbo Boost 2.0* was introduced around 2011 and it uses time windows with different levels of power limits, so that a processor can boost its frequency beyond its thermal design power, which can thus only be maintained for a few seconds without destroying the CPU [2]. *Huge Pages*, which were increased to 1GB, can reduce the overhead of virtual memory translations by using larger virtual memory page sizes which increases the effective size of caches in the memory pipeline [24]. *Branch prediction*, whose early forms already date back to the 1980s in SPARC or MIPS [68], speculates on the condition that most likely occurs if a conditional operation is run. Modern CPUs have a quite sophisticated branch prediction system, which executes potential operations in parallel [48]. The CPU can then complete an operation ahead of time if it made a good guess and significantly speed up the computation. This often depends on how frequently the same operation is used. Otherwise, if the branch predictor guessed wrong, the CPU executes the other branch of operation with some delay, which can be longer than expected as modern processors tend to have quite long pipelines so that the misprediction delay is between 10 and 20 clock cycles. The situation gets more complicated when substantial architectural bugs are mitigated or patched, as this can notably slow down the total system performance [58, 53].

Clearly, we need practical empirical evaluations of algorithms and techniques and often-times it is not useful to just restrict an evaluation to existing benchmarks used in competitions, if they even exists. But just the “complications” or, more formally, source for an error in measurement mentioned above, could make the outcome of an experiment far less deterministic than one would expect. For that reason, we suggest a more rigorous process when evaluating implementations, including the understanding of measurements and effects of potential errors on the outcome as well as approaches to reduce unexpected and not entirely deterministic effects. In a way, the following sections provide a modern perspective on simple measures (runtime) that incorporate state-of-the-art in hardware and operating system technology.

3 Measurements and Hardware Effects

In the following, we discuss measurements used when evaluating runtime of empirical work. Along with the measures, we recap useful measuring and controlling tools. Since most of the tools are highly specific to the kernel in the used operating system, we restrict ourselves to recent versions of Linux and widely used distributions thereof.

3.1 Runtime

When evaluating algorithms, a central question is how long its implementation actually runs on the input data (runtime). There are five main measures that are interesting in this context: real-time, user-time, system-time, CPU-usage, and system load. The *real-time*, frequently just called *wallclock time*, measures the elapsed time between start and end of a considered program (method entry and exit). In contrast, *CPU-time* measures the actual amount of

time for which a CPU was used when executing a program. More precisely, the *user-time* measures how much CPU-time was utilized and *system-time* how much the operating system has used the CPU-time due to system calls by the considered program. Both measures neglect waiting times for input/output (I/O) operations or entering a low-power mode due to energy saving or thermal reasons. There are more detailed time measures on time spent in user/kernel space, idle, waiting for disk, handling interrupts, or waiting for external resources if the system runs on a hypervisor. *CPU-usage* considers the ratio of CPU-time to the CPU capacity as a percentage. It allows for estimating how busy a system is, to quantify how processors are shared between other programs. The *system load* indicates how many programs have been waiting for resources, e.g., a value of 0.05 means that 0.05 processes were waiting for resources. The system load is often given as *load average* which states the last average of a fixed period of time; by default, system tools report three time periods (1, 5, and 15 minutes). If the load average goes above the number of physical CPUs on the system, a program has to idle and wait for free resources on the CPU.

Suggested Measure for Runtime. When measuring runtime, the obvious measure is to use elapsed time, so as to measure the real-time of a program. However, when setting an experiment, we aim to (i) reduce external influences, (ii) conduct reasonable failure analysis, or (iii) use an alternative measure in the worst-case. Real-time can be unreliable on sequential systems as a program can be influenced by other programs running on the system and the program competes on resources with the operating system. For that reason, dated guides on experimenting suggested to run a clean system and obtain a magic overhead factor, which follows Direction (ii) replacing an expected failure analysis. More recent guides, follow Direction (iii) and suggest to use CPU-time [63, 71], mainly arguing that real-time minus unwanted external interruption should roughly equal used CPU-time when evaluating sequential combinatorial solvers that use a CPU close to 100%. However, we believe that the best approach for an experimental setup is always to follow Direction (i) and reduce external influences. Suggestions on CPU-time are outdated as modern hardware is inherently parallel. Even small single-board computers such as the Raspberry Pi have multi-core processors. This allows to run programs and the operating system simply in parallel. Still, CPU-time might prove useful to estimate a degree of parallelism or debug unexpected behavior.

Expected Errors. Real-time is measured by an internal clock of the computer. Nowadays, hardware clocks are still not very accurate. Expected time drifts are about one second per day [95], which is often negligible for standard experiments as micro-benchmarking is anyways rarely meaningful. But, time drift can be far higher, for example, when system load is very high [72] and systems run within virtual machine guests [44, 6, 85]. Since modern cryptography still requires exact system times, all state-of-the-art operating systems synchronize the system clock frequently. Unfortunately, many widely used tools do not incorporate time drifts and corrections by time synchronization utilities. Thus, if time drifts are high (virtual machines) or a misconfiguration of the synchronization service occurs, measures can be completely unreliable. Note that we can expect difference between CPU-time and real-time in cases where heavy or slow access to storage occurs, slow network is involved, or unexpected parallel execution happens. However, this should be ground to investigate details and either eliminate problems in the experimental setup or update problematic program parts, if possible. A classical example occurs when using the ILP solver CPLEX, which sets by default a number of threads equal to the number of cores or 32 threads (whichever number is smaller). An issue, which can especially happen when measuring

CPU-time, is due to the operating system and specific tooling. Namely, a program starts multiple processes, e.g., the program calls a SAT solver, but the monitoring tool captures only one process.

Tools to Measure Runtime. A standard system tool is `GNU time` [50], which provides CPU-time, real-time, and CPU usage of an executed program when run with the command-line flag `-v`. Note that `time` refers to a function in the Linux shell whereas `GNU time` can be found at `/usr/bin/time`. `GNU time` suffers from issues with time skew. A compact, free, and open source tool with extended functionality is `runsolver` [79]. It can be easily compiled and requires only few additional packages, but also suffers from issues with time skew. An extensive monitoring tool is `perf`, which is available in the linux kernel since version 2.6.31 (2009) [101]. `Perf` provides statistical profiling of the entire system when run with flag `stat`. It is easy to use and well documented, but requires installation of an additional package, an additional kernel module, and setting kernel security parameters (`perf_event_paranoid`, `nmi_watchdog`) [55, 61]. However, `perf` is usually available on maintained HPC environments.

Restricting Runtime. Oftentimes when running experiments, we are interested in setting an upper bound on the runtime, let the program run until this time, then terminate and measure how many inputs have been solved successfully. Classical tools to impose a timeout are `timeout` [11], `prlimit` [13], and `ulimit` (obsolete [88]). These tools use a kernel function (`timer_create`) to register a timer. The tools notify the considered program about the occurred timeout by sending a signal to terminate the program, but only to the started program that is responsible to handle potentially started children (entire process hierarchy). For that reason, these tools are often useless or require to build additional wrapper scripts when running academic code, which often omit proper signal handling. A popular tool in the research community that circumvents these problems is the already above mentioned tool `Runsolver` [79], which uses a sampling based approach. It monitors and terminates the entire hierarchy of processes started by the tested program. However, signals are sent to child processes first, which may need additional exception handling in the tested program. Furthermore, the sampling-based approach may cause measurable overhead in used resources. `runexec` is modern and thorough tool for imposing detailed runtime restrictions. It can be found within the larger framework for reliable benchmarking and resource measurement (`BenchExec`) [14]. `runexec` uses kernel control groups (cgroups) to limit resources [46, 36]. Cgroups are precise, but cause a certain overhead and are fairly quite hard to use manually. Unfortunately, `BenchExec` does not directly support commonly used schedulers in HPC environments (except AWS), requires administrative privileges during setup, specially configured privileges at runtime, and fairly new distributions and kernels. It is only widely available on Ubuntu or systems running kernels of version at least 5.11.

Suggested Tooling. In principle, we find `runexec` quite helpful when restricting runtime. It is reliable and has very helpful features such as warning the user about unexpected high system loads. However, it has strong requirements, both in terms of privileges and dependencies, and can be hard to setup, especially in combination with existing cluster scheduling systems. `GNU time` and `timeout` are both system tools available out of the box. Though, when using `timeout` we require additional tools (e.g., `pstree`) and a bit of scripting to handle an entire process hierarchy. Still, both tools might be the best choice if only standard system resources are available and no libraries can be installed. For older systems that are well-maintained or where additional libraries can be installed, we suggest `runsolver`

(enforcement) in combination with `perf` (measurement). Both tools keep setup and handling at a minimum. Issues on potential time-skew and sampling-based issues are minimized and more detailed statistics (memory) can be outputted if needed. However, using this tooling requires to check carefully if the system is over-committed or if `runsolver` terminated a program too late. If required kernel modules or security parameters for `perf` cannot be installed/set, `runsolver` in combination with `GNU time` can be a reasonable alternative.

3.2 CPUs and Scaling

Modern hardware has features to dynamically overclock the CPU, which then can run at high frequency for a short period of time (*Turbo Boost*). Frequency scaling can save energy (*Thermal power design*) when processes do not require full capabilities of the system. These features can significantly impact performance and uncertainty on modern hardware [34]. We provide a brief experiment in Section 3.3 to illustrate effects. Within the operating system, the concept is known as dynamic CPU frequency scaling or CPU throttling, which allows a processor to run at frequency that is not its maximum frequency to conserve power or to save the CPU from overheating if the frequency is beyond its thermally safe base frequency. In fact, modern operating systems have options to manually set performance states. In Linux, the CPU frequency scaling (CPUFreq) subsystem is responsible for scaling. It consists of three layers, namely, the core, scaling governors, and scaling drivers [97]. Available capabilities to modify the CPU frequency depend on the available hardware and driver [97]. A *scaling governor* implements a scaling algorithm to estimate the required CPU capacity [12]. However, minimum and maximum frequency can also be fixed by modifying kernel values. Specifications of modern CPUs detail the safe operating temperature (*Thermal Velocity Boost Temperature*) that still allows to boost the cores to their maximum frequency.

Tools to Modify the CPU Frequency. The tool `cpupower` provides functions to gather information about the physical CPU and set the scaling frequency. The flag `frequency-info` lists supported limits, activated governor, and current frequency. The tool `turbostat` allows to obtain extended information about base frequency, the maximum frequency, and the maximum turbo frequency depending on how many cores are active. The program `frequency-set` allows to set the maximum and minimum scaling frequency using flags `-u` and `-d`, respectively. However, the values can also be manually read/set in the kernel by modifying a text file. The turbo needs to be manually modified depending on the driver [97]. The current frequency can be tested explicitly by running the command: `perf stat -e cycles -I 1000 cat /dev/urandom > /dev/null`.

Revisiting the Experiment. With the knowledge of frequency scaling at hand, we focus our attention to Table 2. There, we state runtime results and number of solved instances in dependence of platform and CPU frequency. More precisely, the maximum CPU frequency and the chosen frequency scaling. Obviously, the runtime and number of solved instances significantly depends on the frequency scaling of the CPU, which already explains why CPUs that permit a higher frequency show less solved instances. From the number of solved instances for COMET LAKE (17 GEN10) CPU and COPPERMINE (PIII) CPU, we can also see that an increase in CPU frequency alone is clearly not the reason for modern solvers running faster on modern hardware than on old hardware.

■ **Table 2** Number of solved SAT instances running the solver **CaDiCal** on varying platforms. Column $s(x)$ contains the solved instances when the runtime is cut off after x minutes. f_a , f_e , and p refer to the available and effective frequency of the CPU in GHz and number of solvers running in parallel, respectively. The $t[h]$ column contains the total runtime in hours for all instances solved within 15 minutes. We enforced limits using kernel governor parameters. Frequencies marked by \star are CPU base-frequencies. † we could not enforce frequencies due to administrative restrictions. For COPPERMINE (PIII), we directly list the results by Fichte et al. [22].

Processor	f_a	f_e	p	$s(15)$	$t[h]$	Processor	f_a	f_e	p	$s(15)$	$t[h]$
COPPERMINE	0.5	0.5	1	98	8.93	HASWELL	3.3	$\star 2.5$	1	189	3.89
COMET LAKE	4.7	0.5	1	160	9.99	SKYLAKE	3.0	$^\dagger 3.0$	1	190	5.12
COMET LAKE	4.7	$\star 0.8$	1	174	9.09	COMET LAKE	4.7	3.9	1	191	3.81
COMET LAKE	4.7	1.5	1	177	7.12	ROME	3.4	2.0	1	190	3.79
COMET LAKE	4.7	2.0	1	189	5.13						

Suggested Setup. When handling thermal management for experiments, one usually balances between three objectives (i) stability and repeatability of the experiment; (iia) maximum speed vs (iib) throughput; and (iii) low effort or no access to thermal management functions of the operating system while aiming to balance (i) and (ii). If we focus our setup on Objective (i), a conservative choice is to set the CPU frequency to its base frequency and limit the parallel processes according to available NUMA regions. Then, the thermal management has limited effects on an experiment. Running the same experiment another system, where the CPU frequency was fixed to the same value and where the memory layout is comparable, shows similar results for CPU-intensive solvers. Such an approach could simplify certain aspects of repeatability. However, then the number of solved instances is lower than the actual capabilities of the hardware, the experiment takes longer, and fewer instances are solved. If we balance towards Objective (iia) obtaining maximum speed of the individual solvers, we ignore thermal management, run at maximum speed, and execute all runs sequentially. However, then throughput is low, only a low number of instances are solved, and vasts of resources on typical server CPUs are wasted. If we balance towards Objective (iib) obtaining maximum throughput during the experiment, we run a number of solvers in parallel for which there is low effect on the turbo frequency. We can obtain the value by the tool **turbostat**. For example, a turbo frequency of 3.9GHz might be acceptable over 4.7GHz if 4 additional solvers can be run in parallel. In fact, one could also simply try to repeat the experiments often to avoid balancing between Objective (iia) and (iib), which would however often require plenty of resources. If we are in Situation (i) with no access to modify the CPU thermal management capabilities or we just want to keep tuning efforts low while still having a reasonable throughput at low solving time, we can just test a reasonable setup. We lookup the thermal velocity boost (TVB) temperature, e.g., [45]. Then, we execute a run with parallel solvers and sample CPU temperature. After evaluating several parallel runs, we favor a configuration where the median temperature is below the TVB temperature and the maximum temperature rarely exceeds TVB temperature.

3.3 CPUs and Parallel Execution

In the 2000s, the end of Moore’s law [69] seemed near as CPU frequency improvements for silicon-based chips started to slow down [9, 78]. Parallel computation started to compensate for this trend and multi-core hardware found its way into consumer computers around

2004. In 2021, parallel hardware is widespread, for example, standard desktop hardware regularly has 8 cores (Intel i9 or Apple M1) or 12 cores (AMD Ryzen) and server systems go up to 64 cores (AMD Rome) or even 128 cores (Ampere Altra) per CPU where multiple sockets are possible. Still, parallel solving is rare in combinatorial communities such as SAT solving [35, 62] or beyond [23]. So a common question that arises in empirical problem solving is whether one can execute sequential solvers meaningful in parallel and speed-up the solution of the overall set of considered instances for an empirical experiment. While it clearly makes sense to carry out an experiment in parallel, one needs some background understanding on the hardware architecture of multi-core systems and on how to gather information about the actual system on which experiments are run. Modern systems with multiple processors on multiple sockets and processors that have multiple cores use a special memory design, namely *Non-uniform memory access (NUMA)*. There, access time to RAM depends on the memory location relative to the physical core. Each processor is directly connect to separate memory; access to “remote” RAM is still possible, but the requests are much slower since they pass through the CPU that controls the local RAM. If the operating system supports NUMA and the user is aware of the NUMA layout of the used system, the hardware architecture can help to eliminate performance degeneration that can occur due to allocation of RAM that is associated with another socket [37, 57]. The effect can be measurable, if consecutive pages are used by exactly one process as done in combinatorial solving. NUMA hardware layout also effects the cache hierarchy (L1, L2, often L3) and address translation buffers (TLB). Recall that caches can have a measurable effect on effectiveness of combinatorial solvers [24].

Evidently, if running an experiment a modern operating system does not solely execute the program under test. It runs function of the operating system itself, events from the hardware such as input from disk, network, user-interfaces or output to graphics devices. Further, programs or functions to control or monitor the program under test are running. These functions might interrupt the execution of the program under test and are often triggered by a mechanism called interrupt. In system programming an interrupt service routine (ISR) handles a specific interrupt condition and is often associated with system drivers or system calls. A common urban legend among students in the combinatorial solving community is that interrupt handling happens on CPU Core 0 (monarch core) and hence no solver should be scheduled on Core 0. However, this is only true when booting the system when firmware hands over control to the operating system kernel. Then, only one core is running, which usually is Core 0, takes on all ISR handling, initializes the system and starts all other cores. In old operating systems load was not distributed to other cores by default and hence the core that started the system would handle all ISRs. However, since version 2.4 Linux supports a concept called SMP affinity, which allows to distribute interrupt handling [67]. The actual balancing and distribution of hardware interrupts over multiple cores is then done by a system process, namely `irqbalance` [41]. Depending on the Linux distribution the balancing is done one-shot at system start, during runtime, or entirely omitted. Nonetheless, it might be helpful to understand the configured system behavior [76].

Tooling for Information on the CPU. Often, we need information on the CPU as starting point for setting up parallel execution of an experiment. Linux reports information on the CPU in the `proc` filesystem as text (`/proc/cpuinfo`) [10]. Among the information is data about the CPU model, microcode, available cores, and instruction sets. The tool `lscpu`, which is part of `util-linux` in most distributions, reports more details on the CPU such as architecture, cache sizes, number of sockets, number of virtual or physical cores, number of threads per core, details on NUMA regions, and active flags. More detailed information

on NUMA regions can be obtained by running the tool `numactl` with flag `-hardware`, using `lscpu`, or by manually listing details in the `cpulist`. Note that NUMA regions and core numbering can be a bit tricky as cores and NUMA regions are often not in consecutive order.

Restricting NUMA, CPU, and IRQ affinity. When running a program on a multicore system, the scheduler in the operating system decides on which core the program runs. In principle, this depends on the current load and on a memory placement policy of the system. Some enterprise distributions have automated processes running (`numad`), which automatically estimate or balance NUMA affinity. Primary benefits are reported for long-running processes with high resource load, but degeneration for continuous unpredictable memory access patterns. The core and allowed memory regions can also be manually restricted. The tool `numactl` provides functionalities to force the execution of a program to certain NUMA nodes or cores, including strict settings [51]. The tool `runsolver`, which we already mentioned above, allows for setting the NUMA and CPU affinity. On modern distributions, these settings can also be set when running a program by `systemd`. Literature on manually tuning NUMA regions and CPU affinity reports both positive and negative effects, but less than 5% performance gain on full core CPU loads [38, 43]. Hence, detailed manual tuning might have a far less effect than what is usually anticipated within the community. Since combinatorial solvers often rely on fast access to caches, it might be more important to ensure that caches are accidentally shared between several running solvers. In principle, the IRQ affinity can be managed manually by setting dedicated flags for the system service `irqbalance`. However, time might be better spent on avoiding over-committing CPUs.

Suggested Tooling and Setup. Experiments that involve measuring runtime need exclusive access to the machine on which experiments are run, i.e., no other software interferes in the background (e.g., running a system update, database, file server, browser, GUI with visual effects) and no other users access the system in the meantime. If the hardware is used for other purposes, runtime differences of 30% and more are common. If an experiment runs on an HPC environment, a uniform configuration is indispensable, i.e., all nodes have the same CPU, microcode, and memory layout. The number of scheduled solver resources should never equal the number of cores on the system, since almost all combinatorial solvers use CPU(s) at full load and operating system and measurement tools require a certain overhead. If NUMA layout details are missing, one can take a rough estimate. Assume that controlling and monitoring software as well as the operating system need one core per tested program, add the expected number of occupied cores of the tested solver, and for a safe buffer multiply the result by two. However, a better approach is to gather detailed information and test whether an anticipated setup is stable. Information on the available CPUs and NUMA regions can be obtained by using the tools `lscpu` and `numactl`. Modern operating systems implement NUMA scheduling already well. However, it is still important to report details of the system within logs of the experiments. If manual NUMA region enforcement is needed, each running solver should only access the NUMA region on which it is pinned to [77]. Solvers requiring fast caches should not be scheduled in parallel on cores sharing L1 and L2 cache.

Effects of Parallel Runs, CPU Scaling, and Timeouts in Practice

In the previous section, we listed complications that may occur from technical specifications of modern processors and techniques present in modern operating systems. Next, we present a detailed experiment on parallel execution of solvers incorporating effects of actual processor frequency, stability of parallel runs, thermal issues, in combination with runtime and number

■ **Table 3** Overview on frequency scaling, thermal observations, and the number of solved instances (out of 200) on an Intel COMET LAKE (17 GEN10) processor for different number of parallel runs of the solver **CaDiCal**. The column “ p ” refers to an upper bound on the number of instances that are solved in parallel and “ $t_r[h]$ ” refers to the total runtime of the experiment in hours. While the maximum CPU frequency is 4.7 GHz, the column “ f_o ” states the observed frequency in GHz and f_{std} to its standard deviation. Column “ θ_o ” lists the observed CPU temperature; θ_{max} to the maximum temperature in °C. The column “ $s(x)$ ” contains the number of solved instances when the runtime is cut off after x minutes. The column “ t_s ” refers to the total runtime (real-time) of the solved instances in hours at maximum runtime of 1500s for each instance. Finally, “s5k” how many instances can be solved in 5000s if instances are ordered by hardness and each run has at most 1500s. We used a simple python wrapper to start the parallel runs.

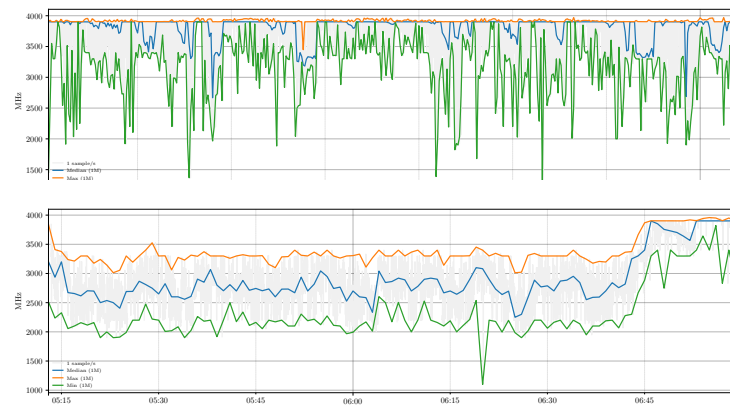
p	$t_r[h]$	$f_o[GHz]$	f_{std}	$\theta_o[^\circ C]$	θ_{max}	$s(1)$	$s(5)$	$s(10)$	$s(15)$	$s(25)$	$t_s[h]$	s5k
1	7.37	3.90	0.26	53.4	64.0	132	179	190	191	193	4.43	161
2	4.06	3.69	0.29	60.8	72.5	125	179	189	191	193	4.97	158
4	2.49	3.30	0.28	74.2	92.0	120	175	183	190	192	5.74	150
6	1.85	2.95	0.30	76.6	94.5	111	171	181	189	191	6.68	142
8	1.77	2.81	0.46	74.5	94.0	98	160	176	183	190	8.28	131
10	1.77	2.71	0.57	74.0	92.0	88	155	174	181	189	9.66	123
12	1.59	2.59	0.51	87.0	72.5	75	145	171	176	187	10.82	117
14	1.47	2.51	0.28	91.5	72.5	70	140	162	174	184	11.17	111

of solved instances. We specify the setup, used measures, and common expectations of which some might be contradictory. In order to obtain a better view on effects of timeouts, we increase the maximum runtime per instance to 1500 seconds.

► **Experiment 1 (Parallel Runs).** *We investigate complications of solving multiple instances in parallel with one sequential SAT solver on a fixed hardware.*

- *Setup: solve 200 instances by one SAT solver (**CaDiCal**) on COMET LAKE (17 GEN10), maximum runtime per instance (timeout) 1500 seconds.*
- *Measures: Runtime (real-time) [h], number of solved instances, temperature (median of sampling each 1s the average temperature over all cores) [°C], and CPU frequency (median of sampling each 1s the average over all cores) [GHz].*
- *Expectation 1a: Solving should never be executed in parallel on one machine as the runtime and number of solved instances significantly differ otherwise.*
- *Expectation 1b: Full parallel capabilities should be employed as long as runtime and number of solved instances remains similar.*
- *Expectation 2: Relying on multithreading degrades runtime.*
- *Expectation 3: Measures are stable over small runtime changes.*

Observations. Results of the first experiment are illustrated in Table 3. The number of solved instances for 1, 5, 10, 15, and 25 minutes provide an overview on how many instances can be solved quickly. Unsurprisingly, the total runtime of an experiment depends on the number of parallel processes running. More precisely, the total runtime of the experiment varies between 7.37 hours and 1.77 hours when running 1 or 10 instances in parallel. Just by running 4 instances in parallel instead of 1 we cut runtime down to 33% of the original runtime and still to 55% for 2 instances. However, the total real-time of the solved instances varies between 4.43 hours and 5.74 hours (23%). The number of solved instances varies by



■ **Figure 1** Illustration of the CPU frequency scaling when running the sequential solver `CaDiCa1` on the considered instance set by solving in parallel 1 instance (upper) and 8 instances (lower).

2% at 25 minutes and 5% at 15 minutes, 13% at 5 minutes, and 33% at 1 minute timeout. When comparing the effect on the measure how many instances can be solved within 5000s, we obtain a notable 24% decrease. Surprisingly, the median CPU frequency never reached 4.7GHz even when running only one instance. The actual frequency reduced significantly when more instances are running. Figure 1 illustrates the changes of the CPU frequency over time for 1 and 8 instances solved in parallel. We see that the frequency is hardly consistent and increases significantly as soon as most instances are finished and less processes run in parallel. When using multiple cores, the median CPU temperature increases significantly and may even spike (94°C) close to the maximum operating temperature of the CPU (100°C).

Interpretation. On the considered set of instances, the number of solved instances and real-time over all solved instances decreases with an increasing number of instances run in parallel. The effect is particularly high, if the timeout was set very low or if the measure is number of instances solved within 5000s. This is not entirely surprising, since instances in the considered set were selected by Hoos et al. [40] using a distribution of instance hardness leading to many instances of medium hardness and a few easy and hard instances. Then, if the considered timeout is low, a small constant improvement by hardware effects can increase the number of solved instances notably. In contrast, there is only a 2% difference between number of solved instances when timeouts are higher. The measure of solved instances within 5000s is particularly runtime dependent and hence configuration of the experimental setup has notable effects. Regarding runtime, we can see that the real-time over all solved instances almost doubles when running almost as many instances as cores are available. However, the entire experiment finishes significantly faster, i.e., about 24% of the original runtime. Surprisingly, the CPU frequency was far below the potential 4.7GHz. If we check more details on the specification of the COMET LAKE (i7 GEN10) CPU or by running the tool `turbostat`, we observe that the maximum frequency of the CPU is only 3.9GHz if 6 cores are active, i.e., not explicitly suspended. While our considered system has 12 MT cores, it has only 6 physical cores. Hence, we observe a measurable degeneration in number of solved instances when running more instances in parallel than present physical cores are present. When considering runtime, we observe a considerable increase when more than 2 instances run in parallel, as CPU frequency measurably drops and temperature increases significantly.

Outcome. After summarizing observations and interpretation of our experiment, we briefly evaluate phrased expectations from above. In theory, we would expect that Expectation 1a is true for real-time and number of solved instances within 5000s, which is also quite sensitive for runtime influences. Indeed, there is a measurable influence in runtime, but only slightly decrease in number of solved instances, while the experiment finishes much faster. If we take higher timeout, the number of parallel executions affects the runtime only if already known rough estimates are exceeded. Still, the number of parallel executions is influenced by throttling of the processor. Expectation 1b clearly does not hold. All measures are influenced by a higher system load and hence by solving several instances in parallel. While we can confirm Expectation 3 in the experiment, multithreading is not the only reason. Clearly, already when using all available cores runtime and number of solved degenerate. Unfortunately, our experiment does not fulfill Expectation 3. All considered measures are influenced by parallel execution. Especially, limiting the total solving time is prone to hardware effects and might accidentally over-highlight constant runtime improvements. Since the frequency is also not stable when running only one instance, fixing the frequency might be a reasonable approach during experimenting. However, if the base-frequency is exceeded, a stable frequency should be estimated and experimentally verified before comparing runtime and number of solved instances with multiple solvers. In our case, operating the CPU at fixed 3GHz showed stable frequency results when running 1–2 instances in parallel. Under the light of the mentioned complications, we fear that a single measure incorporating runtime, number of solved instances, and a cutoff time is problematic if setup is neglected.

3.4 Input/Output

Input and output performance, *I/O* for short, talks about read or write operations involving a storage device. On a desktop computer storage is usually restricted to local disks. On cluster environments, nodes have access to a central storage over network, fast temporary storage (over network), and local disks. Here, a variety of different topics are involved, for example, hardware (storage arrays/network), network protocols, and file systems, which can make it inherently complicated. Therefore, we provide only a brief and simple suggestion: keep external influence as low as possible. When reading input and writing output, use a shared memory file system (shm) to avoid external overhead. Before starting the solver under test, input files are copied in-memory. Then, measuring runtime starts when executing the solver, which takes as input the temporary files on the memory and outputs only to a shared memory file system. The measurement ends when the solver is terminated and afterwards temporary files are copied to the permanent storage and deleted from the temporary storage. This approach minimizes side effects from slow network devices and avoids side effects that may occur with large files and system file caches, especially when running multiple solvers on the same input. However, if files are too large or solvers need the entire RAM, temporary in-memory cannot be used and fast local disks (e.g., NVMe) can provide an alternative.

4 Conclusion

Empirical evaluations are essential to confirm observations in algorithmics and combinatorics beyond theory. Many evaluations typically focus on comparing runtimes and number of solved instances, since both measures are easy targets for comparison and probably roughly reflect needs of end users. However, the number of solved instances is sensitive to the chosen benchmark, so one has to be cautious about it. Playing devils advocate, we can even ask to what extent runtime is even a meaningful measure on modern hardware. If one solver is a factor of ten faster than another, we are fairly confident in it, but does modern hardware

allow for accurate comparisons at a range of, say, 10%, which might be the contribution of an individual feature or optimization towards the hardware? Similar to experimental physics, we can simply repeat an experiment often or repeat in different environments. However, in combinatorial solving this is not always possible if many solvers need to be tested or a reasonably high number of hard instances have to be considered. Hence, we believe that an experimental setup should still be carried out thoroughly. Future work could consider up to what extent certain aspects can be neglected and how repetition can circumvent minor issues. In fact, our work only explains and illustrates certain complications from modern hardware to make researchers aware of potential issues. In a way, we also show that complications do not just concern CPU frequency, but also the experimental setup (timeouts, cutoffs, parallel running processes). Clearly, there is no reason to forbid the use of certain platforms, if we are aware of complications. On the meta level, we believe that clearly marking strengths and weaknesses of solvers provides more insights than finding scenarios where one solver is best.

An interesting question for future research is the boarder topic of SIMD and branch prediction, which could affect repeatability, replicability, and reproducibility. Both features are quite relevant for how a good solver author can write code, but it is unclear whether they can even change the overall results when comparing two solvers. In practice, one could maybe investigate issues by taking different versions of a CPU (or different firmware).

Further, we think that papers presenting experimental evaluations could provide a simple benchmark protocol as appendix, similar to literature as part of reproducibility work. Best practices and checklists could be developed in a community effort after thorough discussions and more detailed works. This can also include detailed guides or suggested configurations for standard cluster schedulers such as Slurm [99]. Having a list of common parameters to report or even practical tools could prevent manual repetitive labor. Thereby, we leave room for actual scientific questions, e.g., why implementations are efficient for certain domains [91, 29].

Finally, our experiments focused on consumer hardware, detailed investigations with server hardware are interesting for future investigations to confine limits of parallel execution.

References

- 1 Altuna Akalin. Scientific data analysis pipelines and reproducibility. <https://aakalin.medium.com/>, 2018. Retrieved 2021-05-29.
- 2 Chris Angelini. Intel’s second-gen core CPUs: The Sandy bridge review. <https://www.tomshardware.com/reviews/sandy-bridge-core-i7-2600k-core-i5-2500k,2833-3.html>, 2011.
- 3 Gilles Audemard, Loïc Paulevé, and Laurent Simon. SAT heritage: A community-driven effort for archiving, building and running more than thousand sat solvers. In Luca Pulina and Martina Seidl, editors, *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT’20)*, volume 12178 of *Lecture Notes in Computer Science*, pages 107–113, Alghero, Italy, July 2020. Springer Verlag.
- 4 David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994. doi:10.1145/197405.197406.
- 5 Daniel Le Berre, Matti Järvisalo, Armin Biere, and Kuldeep S. Meel. The SAT practitioner’s manifesto v1.0. <https://github.com/danielleberre/satpractitionermanifesto>, 2020.
- 6 Donald Berry. Avoiding clock drift on VMs. <https://www.redhat.com/en/blog/avoiding-clock-drift-vm>, 2017.
- 7 Dirk Beyer, Stefan Löwe, and Philipp Wendler. Benchmarking and resource measurement. In Bernd Fischer and Jaco Geldenhuys, editors, *Proceedings of the 22nd International Symposium on Model Checking of Software (SPIN’15)*, volume 9232 of *Lecture Notes in Computer Science*, pages 160–178. Springer Verlag, 2015. doi:10.1007/978-3-319-23404-5_12.

- 8 Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, 2019. doi:10.1007/s10009-017-0469-y.
- 9 Steve Blank. What the GlobalFoundries’ retreat really means. *IEEE Spectrum*, 2018. URL: <https://spectrum.ieee.org/nanoclast/semiconductors/devices/what-globalfoundries-retreat-really-means>.
- 10 Terrehon Bowden, Bodo Bauer, Jorge Nerin, Shen Feng, and Stefani Seibold. The /proc filesystem. <https://www.kernel.org/doc/Documentation/filesystems/proc.txt>, 2009.
- 11 Padraig Brady. Timeout(1). <https://www.gnu.org/software/coreutils/timeout>, 2019.
- 12 Dominik Brodowski, Nico Golde, Rafael J. Wysocki, and Viresh Kumar. CPU frequency and voltage scaling code in the Linux(tm) kernel. <https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt>, 2016.
- 13 Davidlohr Bueso. Util-linux 2.37. <https://www.kernel.org/pub/linux/utils/util-linux/>, 2021.
- 14 Alejandro Colomar et al. Ulimit(3). <https://www.kernel.org/doc/man-pages/>, 2017.
- 15 Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. Identifiers for digital objects: the case of software source code preservation. In *iPRES 2018 - 15th International Conference on Digital Preservation*, 2018. doi:10.17605/OSF.IO/KDE56.
- 16 Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In *iPRES 2017: 14th International Conference on Digital Preservation*, Kyoto, Japan, 2017. URL: <https://www.softwareheritage.org/wp-content/uploads/2020/01/ipres-2017-swh.pdf><https://hal.archives-ouvertes.fr/hal-01590958>.
- 17 Roberto Di Cosmo, Stefano Zacchiroli, Gérard Berry, Jean-François Abramatic, Julia Lawall, and Serge Abiteboul. Software heritage. <https://www.softwareheritage.org/mission/heritage/>, 2020.
- 18 Chris Drummond. Replicability is not reproducibility: Nor is it good science. In *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*, Montreal, Canada, 2009.
- 19 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 1291–1299. International Joint Conferences on Artificial Intelligence Organization, July 2018. doi:10.24963/ijcai.2018/180.
- 20 Johannes K. Fichte and Markus Hecher. The model counting competition 2021. https://mccompetition.org/past_iterations, 2021.
- 21 Johannes K. Fichte, Markus Hecher, and Florim Hamiti. The model counting competition 2020. *ACM Journal of Experimental Algorithmics*, 2021. In press.
- 22 Johannes K. Fichte, Markus Hecher, and Stefan Szeider. A time leap challenge for SAT-solving. In Helmut Simonis, editor, *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP’20)*, pages 267–285, Louvain-la-Neuve, Belgium, September 2020. Springer Verlag. doi:10.1007/978-3-030-58475-7_16.
- 23 Johannes K. Fichte, Markus Hecher, Stefan Woltran, and Markus Zisser. Weighted model counting on the GPU by exploiting small treewidth. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *Proceedings of the 26th Annual European Symposium on Algorithms (ESA’18)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:16. Dagstuhl Publishing, 2018. doi:10.4230/LIPIcs.ESA.2018.28.
- 24 Johannes K. Fichte, Norbert Manthey, André Schidler, and Julian Stecklina. Towards faster reasoners by using transparent huge pages. In Helmut Simonis, editor, *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP’20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 304–322, Louvain-la-Neuve, Belgium, September 2020. Springer Verlag. doi:10.1007/978-3-030-58475-7_18.
- 25 Ronald Fisher. Arrangement of field experiments. *Journal of the Ministry of Agriculture of Great Britain*, pages 503–513, 1926.

- 26 Ronald Fisher. *The Design of Experiments*. Oliver and Boyd, 1935.
- 27 Vijay Ganesh and Moshe Y. Vardi. On the unreasonable effectiveness of sat solvers, 2021.
- 28 Ian P. Gent. The recomputation manifesto. *CoRR*, abs/1304.3674, 2013. [arXiv:1304.3674](https://arxiv.org/abs/1304.3674).
- 29 Ian P Gent and Toby Walsh. The SAT phase transition. In Anthony G. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI'94)*, volume 94, pages 105–109, Amsterdam, The Netherlands, 1994. PITMAN.
- 30 Stephan Gocht, Jakob Nordström, and Amir Yehudayoff. On division versus saturation in pseudo-boolean solving. In Sarit Kraus, editor, *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*, pages 1711–1718. International Joint Conferences on Artificial Intelligence Organization, July 2019. doi:10.24963/ijcai.2019/237.
- 31 Odd Erik Gundersen. The reproducibility crisis is real. *AI Magazine*, 2020. doi:10.1609/aimag.v41i3.5318.
- 32 Odd Erik Gundersen. AAI'21 reproducibility checklist. <https://aaai.org/Conferences/AAAI-21/reproducibility-checklist/>, 2021.
- 33 Odd Erik Gundersen and Sigbjørn Kjensmo. State of the art: Reproducibility in artificial intelligence. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI'18)*, 2018.
- 34 Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the Intel Haswell processor. In Jean-Francois Lalande and Teng Moh, editors, *Proceedings of the 17th International Conference on High Performance Computing & Simulation (HPCS'19)*, 2019.
- 35 Youssef Hamadi and Christoph M. Wintersteiger. Seven challenges in parallel SAT solving. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI'12)*, pages 2120–2125, Toronto, Ontario, Canada, 2012. The AAAI Press.
- 36 Tejun Heo. Control group v2. <https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt>, 2015.
- 37 Christopher Hollowell, Costin Caramarcu, William Strecker-Kellogg, Antonio Wong, and Alexandr Zaytsev. The effect of numa tunings on cpu performance. *Journal of Physics: Conference Series*, 2015.
- 38 Christopher Hollowell, Costin Caramarcu, William Strecker-Kellogg, Antonio Wong, and Alexandr Zaytsev. The effect of NUMA tunings on CPU performance. *Journal of Physics: Conference Series*, 664(9):092010, December 2015. doi:10.1088/1742-6596/664/9/092010.
- 39 John N. Hooker. Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1:33–42, 1995. doi:10.1007/BF02430364.
- 40 Holger H. Hoos, Benjamin Kaufmann, Torsten Schaub, and Marius Schneider. Robust benchmark set selection for Boolean constraint solvers. In *Proceedings of the 7th International Conference on Learning and Intelligent Optimization (LION'13)*, volume 7997 of *Lecture Notes in Computer Science*, pages 138–152, Catania, Italy, January 2013. Springer Verlag. Revised Selected Papers.
- 41 Neil Horman, PJ Waskiewicz, and Anton Arapov. Irqbalance. <http://irqbalance.github.io/irqbalance/>, 2020.
- 42 Sascha Hunold, Alexandra Carpen-Amarie, and Jesper Larsson Träff. Reproducible MPI micro-benchmarking isn't as easy as you think. In *Proceedings of the 21st European MPI Users' Group Meeting, EuroMPI/ASIA '14*, pages 69–76, New York, NY, USA, 2014. Association for Computing Machinery, New York. doi:10.1145/2642769.2642785.
- 43 Satoshi Imamura, Keitaro Oka, Yuichiro Yasui, Yuichi Inadomi, Katsuki Fujisawa, Toshio Endo, Koji Ueno, Keiichiro Fukazawa, Nozomi Hata, Yuta Kakibuka, Koji Inoue, and Takatsugu Ono. Evaluating the impacts of code-level performance tunings on power efficiency. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 362–369, 2016. doi:10.1109/BigData.2016.7840624.

- 44 VmWare Inc. Timekeeping in VMware virtual machines. http://www.vmware.com/pdf/vmware_timekeeping.pdf, 2008.
- 45 Intel. Intel product specifications: Processors. <https://ark.intel.com/content/www/us/en/ark.html#Processors>, 2021.
- 46 Paul Jackson and Christoph Lameter. cgroups - Linux control groups. <https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt>, 2006.
- 47 Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazin*, 33(1), 2012. URL: <http://www.aaai.org/ojs/index.php/aimagazine/article/view/2395>, doi:10.1609/aimag.v33i1.2395.
- 48 D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of 7th International Symposium on High-Performance Computer Architecture (HPCA'01)*, pages 197–206, 2001.
- 49 Armin Biere Katalin Fazekas, Daniela Kaufmann. Ranking robustness under sub-sampling for the SAT competition 2018. In Matti Järvisalo and Daniel Le Berre, editors, *Proceedings of the 10th Workshop on Pragmatics of SAT (POS'19)*, 2019.
- 50 Michael Kerrisk. *The Linux Programming Interface*. No Starch Press, 2010. URL: <https://man7.org/tlpi/>.
- 51 Andi Kleen, Cliff Wickman, Christoph Lameter, and Lee Schermerhorn. numactl. <https://github.com/numactl/numactl>, 2014.
- 52 Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley, 2015.
- 53 Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.
- 54 Michael Kohlase. The theorem prover museum – conserving the system heritage of automated reasoning. *CoRR*, abs/1904.10414, 2019. arXiv:1904.10414.
- 55 Divya Kiran Kumar. Installing and using Perf in Ubuntu and CentOS. <https://www.fossilinux.com/7069/installing-and-using-perf-in-ubuntu-and-centos.htm>, 2019.
- 56 G. M. Kurtzer, V. Sochat, and Michael W Bauer. Singularity: Scientific containers for mobility of compute. *PLoS ONE*, 12, 2017. doi:10.1371/journal.pone.0177459.
- 57 Christoph Lameter. Numa (non-uniform memory access): An overview: Numa becomes more common because memory controllers get close to execution units on microprocessors. *Queue*, 11(7):40–51, 2013.
- 58 Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
- 59 Zhiyuan Liu and Jian Tang. IJCAI 2021 reproducibility guidelines. <http://ijcai-21.org/wp-content/uploads/2020/12/20201226-IJCAI-Reproducibility.pdf>, 2020.
- 60 Eric Ma. How to statically link C and C++ programs on Linux with gcc. <https://www.systutorials.com/how-to-statically-link-c-and-c-programs-on-linux-with-gcc/>, 2020. Accessed 20-April-2021.
- 61 Kamil Maciorowski and meuh. Run perf without root-rights. <https://superuser.com/questions/980632/run-perf-without-root-rights>, 2015.
- 62 Ruben Martins, Vasco Manquinho, and Inês Lynce. An overview of parallel sat solving. *Constraints*, 17(3):304–347, 2012. doi:10.1007/s10601-012-9121-3.
- 63 Catherine C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012.
- 64 Doug McIlroy. Unix time-sharing system: Foreword. *The Bell System Technical Journal*, pages 1902–1903, 1978.

- 65 Marcin Miłkowski, Witold M. Hensel, and Mateusz Hohol. Replicability or reproducibility? on the replication crisis in computational neuroscience and sharing only relevant detail. *Journal of Computational Neuroscience*, 2018.
- 66 Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 261–270, 2009. doi:10.1109/PACT.2009.22.
- 67 Ingo Molnar and Max Krasnyansky. Smp irq affinity. <https://www.kernel.org/doc/Documentation/IRQ-affinity.txt>, 2012.
- 68 Matteo Monchiero and Gianluca Palermo. The combined perceptron branch predictor. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, pages 487–496, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- 69 Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.
- 70 Hassan Mujtaba. Intel Xeon E7 'Ivy Bridge-EX' lineup detailed – Xeon E7-8890 V2 'Ivy Town' chip with 15 cores and 37.5 mb LLC. wccftech.com, 2014.
- 71 Matthias Müller-Hannemann and Stefan Schirra, editors. *Algorithm Engineering*, volume 5971 of *Theoretical Computer Science and General Issues*. Springer Verlag, 2010. Bridging the Gap Between Algorithm Theory and Practice. doi:10.1007/978-3-642-14866-8.
- 72 Steven J. Murdoch. Hot or not: Revealing hidden services by their clock skew. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 27–36, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1180405.1180410.
- 73 Gonzalo Navarro. RCR: Replicated computational results initiative. <https://dl.acm.org/journal/jea/rcr-initiative>, 2021.
- 74 Naomi Oreskes. *Why Trust Science?* The University Center for Human Values Series, 2021.
- 75 Jeffrey M. Perkel. Challenge to scientists: does your ten-year-old code still run? *Nature*, 584:656–658, 2020. doi:10.1038/d41586-020-02462-7.
- 76 Red Hat Developers. Optimizing Red Hat Enterprise Linux performance by tuning IRQ affinity. <https://access.redhat.com/articles/216733>, 2011.
- 77 Benoit Rostykyus and Gabriel Hartmann. Predictive CPU isolation of containers at Netflix. <https://netflixtechblog.com/predictive-cpu-isolation-of-containers-at-netflix-91f014d856c7>, 2019. accessed 20-May-2021.
- 78 David Rotman. We're not prepared for the end of moore's law. *MIT Technology Review*, 2020.
- 79 Olivier Roussel. Controlling a solver execution with the runsolver tool. *J. on Satisfiability, Boolean Modeling and Computation*, 7:139–144, 2011.
- 80 Olivier Roussel and Christophe Lecoutre. XCSP3 competition 2019. <http://xcsp.org/competition>, July 2019. A joint event with the 25th International Conference on Principles and Practice of Constraint Programming (CP'19).
- 81 Marta Serra-Garcia and Uri Gneezy. Nonreplicable publications are cited more than replicable ones. *Science Advances*, 7(21), 2021. doi:10.1126/sciadv.abd1705.
- 82 Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. The SAT2002 competition (preliminary draft). <http://www.satcompetition.org/2002/online-report.pdf>, 2002.
- 83 Helmut Simonis, George Katsirelos, Matt Streeter, and Emmanuel Hebrard. CSP 2009 competition (CSC'2009). <http://www.cril.univ-artois.fr/CSC09/>, July 2009.
- 84 Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Proceedings of the 7th International Joint Conference on Automated Reasoning (IJCAR'14)*, volume 8562 of *Lecture Notes in Computer Science*, pages 367–373, Vienna, Austria, July 2014. Springer Verlag. Held as Part of the Vienna Summer of Logic, VSL 2014. doi:10.1007/978-3-319-08587-6_28.

- 85 SUSE Support. Clock drifts in KVM virtual machines. <https://www.suse.com/support/kb/doc/?id=000017652>, 2020.
- 86 Guido Tack and Peter J. Stuckey. The MiniZinc challenge 2019. <https://www.minizinc.org/challenge.html>, 2019. A joint event with the 25th International Conference on Principles and Practice of Constraint Programming (CP'19).
- 87 Andrew S. Tanenbaum and James R. Goodman. *Structured Computer Organization*. Prentice Hall PTR, USA, 4th edition, 1998.
- 88 The Open Group. The open group base specifications issue 7, 2018 edition ieee std 1003.1-2017 (revision of ieee std 1003.1-2008). <https://pubs.opengroup.org/onlinepubs/9699919799/functions/ulimit.html>, 2018.
- 89 Linus Torvalds. LKML archive on lore.kernel.org: Very slow clang kernel config. https://lore.kernel.org/lkml/CAHk--whs8QZf3YnifdLv57+FhBi5_WeNTG1B-su0ES=RcUSmQg@gmail.com/, 2021.
- 90 A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. doi:10.1112/plms/s2-42.1.230.
- 91 Moshe Y. Vardi. Boolean satisfiability: Theory and engineering. *Communications of the ACM*, 57(3):5, 2014. doi:10.1145/2578043.
- 92 Felix von Leitner. Source code optimization. Linux Congress 2009. http://www.linux-kongress.org/2009/slides/compiler_survey_felix_von_leitner.pdf, 2009.
- 93 J. von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993. doi:10.1109/85.238389.
- 94 Szymon Wasik, Maciej Antczak, Jan Badura, Artur Laskowski, and Tomasz Sternal. Cscw '16 companion. In Eric Gilbert and Karrie Karahalios, editors, *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*, Association for Computing Machinery, New York, pages 433–436, New York, NY, USA, Optil.io: Cloud Based Platform For Solving Optimization Problems Using Crowdsourcing Approach. acm. doi:10.1145/2818052.2869098.
- 95 Ulrich Windl, David Dalton, Marc Martinec, and Dale R. Worley. The NTP FAQ and HOWTO: Understanding and using the network time protocol. <http://www.ntp.org/ntpfaq/NTP-s-sw-clocks-quality.htm#AEN1220>, 2006.
- 96 Ricardo Wurmus, Bora Uyar, Brendan Osberg, Vedran Franke, Alexander Godschan, Katarzyna Wreczycka, Jonathan Ronen, and Altuna Akalin. PiGx: reproducible genomics analysis pipelines with GNU Guix. *GigaScience*, 7(12), October 2018. giy123. doi:10.1093/gigascience/giy123.
- 97 Rafael J. Wysocki. intel_pstate CPU performance scaling driver. https://www.kernel.org/doc/html/v5.12/admin-guide/pm/intel_pstate.html, 2017.
- 98 Rengan Xu, Frank Han, and Nishanth Dandapanthula. Containerizing HPC applications with singularity. https://downloads.dell.com/manuals/all-products/esuprt_solutions_int/esuprt_solutions_int_solutions_resources/high-computing-solution-resources-white-papers10_en-us.pdf, 2017.
- 99 Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple Linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'03)*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60, Seattle, WA, USA, 2003. Springer Verlag.
- 100 Zhenyun Zhuang, Cuong Tran, and Jerry Weng. Don't let Linux control groups run uncontrolled: Addressing memory-related performance pitfalls of cgroups. https://engineering.linkedin.com/blog/2016/08/don_t-let-linux-control-groups-uncontrolled, 2016. Accessed Apr-28-2020.
- 101 Peter Zijlstra, Ingo Molnar, and Arnaldo Carvalho de Melo. Performance events subsystem. <https://github.com/torvalds/linux/tree/master/tools/perf>, 2009.


A Job Dispatcher for Large and Heterogeneous HPC Systems Running Modern Applications

Cristian Galleguillos  

Pontificia Universidad Católica de Valparaíso, Chile
University of Bologna, Italy

Zeynep Kiziltan  

University of Bologna, Italy

Ricardo Soto  

Pontificia Universidad Católica de Valparaíso, Chile

Abstract

High-performance Computing (HPC) systems have become essential instruments in our modern society. As they get closer to exascale performance, HPC systems become larger in size and more heterogeneous in their computing resources. With recent advances in AI, HPC systems are also increasingly being used for applications that employ many short jobs with strict timing requirements. HPC job dispatchers need to therefore adopt techniques to go beyond the capabilities of those developed for small or homogeneous systems, or for traditional compute-intensive applications. In this paper, we present a job dispatcher suitable for today's large and heterogeneous systems running modern applications. Unlike its predecessors, our dispatcher solves the entire dispatching problem using Constraint Programming (CP) with a model size independent of the system size. Experimental results based on a simulation study show that our approach can bring about significant performance gains over the existing CP-based dispatchers in a large or heterogeneous system.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Computing methodologies → Planning and scheduling

Keywords and phrases Constraint programming, HPC systems, heterogeneous systems, large systems, on-line job dispatching, resource allocation

Digital Object Identifier 10.4230/LIPIcs.CP.2021.26

Related Version *Previous Version:* <https://arxiv.org/abs/2009.10348>

Supplementary Material *Software (Source Code):* <https://git.io/fjia1>
archived at `swb:1:dir:763b27390dd764a27ae8e7dff5dc0d724cbabe88`

Funding *Cristian Galleguillos:* Funded by Postgraduate Grant INF-PUCV 2020.

Acknowledgements We thank A. Bartolini, L. Benini, M. Milano, M. Lombardi and the SCAI group at Cineca for providing the Eurora data, and A. Borghesi and T. Bridi for sharing the implementations of the original CP-based dispatchers. We also thank the School of Computer Engineering of PUCV in Chile for providing access to computing resources.

1 Introduction

Motivations

High Performance Computing (HPC) is the application of supercomputers to solve complex computational problems, which has become indispensable for scientific progress, industrial competitiveness, economic growth and quality of life in our modern society [18, 22]. An HPC system is a network of computing nodes, each containing several powerful CPUs and a large pool of memory. The world's fastest systems today can reach hundreds of petaFLOPs (10^{15} floating-point operations per second) and they are expected to reach soon the exaFLOP level (10^{18} FLOPs) [16]. Indeed, today's most powerful system Fugaku has recently increased



© Cristian Galleguillos, Zeynep Kiziltan, and Ricardo Soto;
licensed under Creative Commons License CC-BY 4.0

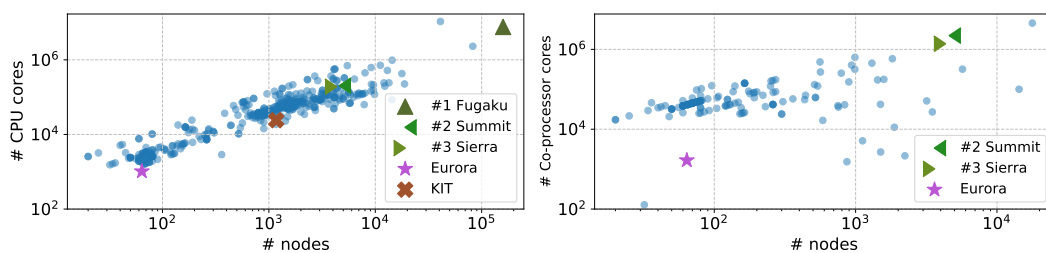
27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 26; pp. 26:1–26:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Eurora, KIT ForHLR II and top 500 HPC systems of June 2021.

its performance on a mixed-precision HPC-AI benchmark to 2 exaFLOPs.¹ In their march towards this elevated performance, HPC systems are getting larger in size and becoming more heterogeneous in their computing resources in an effort to keep the power consumption at bay. Figure 1 shows in blue dots the size of today’s top 500 systems¹ and their number of CPU cores and co-processor cores (with the green triangles referring to the top 3 systems). The majority of these systems have thousands of nodes with tens and hundreds of thousands of CPU cores in total. Around 30% of them employ energy-efficient accelerators such as GPUs and Many Integrated Cores (MICs), in addition to the traditional CPUs and memory. The number of co-processor cores are above ten thousand in most of such systems.

Central to the efficiency and the Quality-of-Service (QoS) of an HPC system is the *job dispatcher* which decides the jobs to run next among those waiting in the queue (*scheduling*) and on which resources to run them (*allocation*). This is an on-line decision making problem because the process is repeated periodically as new jobs arrive to the system while some previously dispatched jobs are still running. Traditionally, HPC job dispatchers have been designed for compute-intensive jobs requiring days to complete. There is an increasing trend where HPC systems are being used for modern applications that employ many short jobs (< 1 h), such as data analytics as data is being streamed from a monitored system [23]. In such application scenarios, response times are critical for acceptable user experience, hence job dispatchers need to rapidly process a large number of short jobs in making on-line decisions. Though optimal dispatching is a critical requirement in HPC systems, the on-line job dispatching is an NP-hard optimization problem [5].

In this paper, we propose an on-line job dispatcher suitable for today’s large and heterogeneous HPC systems running modern applications. Differently from the existing techniques based on heuristic algorithms [11, 27], we exploit the power of Constraint Programming (CP), which has a long track record of success in job scheduling and resource allocation problems [2]. While past work had already used CP in this context, the focus was on small or homogeneous systems where the nodes have only CPUs and memory.

Related work

The first CP-based HPC dispatcher was introduced in [3] and shown to obtain better solutions compared to a Priority Rule-Based (PRB) dispatcher [9, 15], which is widely adopted in commercial HPC workload management systems such as Altair PBS Professional [1] and SLURM Workload Manager [25]. The dispatcher was later embedded as a plug-in within the software framework of PBS professional [8]. Another CP-based dispatcher with the additional feature of limiting system power consumption was presented in [7, 6] and proved to outperform a PRB dispatcher on the instances with tight power capping values.

¹ <https://www.top500.org>

Subsequently, [13] presented two CP-based on-line job dispatchers for HPC systems, which we here refer to as PCP'19 and HCP'19. They were built on the previous CP-based dispatchers [3, 7] and redesigned for satisfying the challenges of systems running modern applications that employ many short jobs and that have strict timing requirements. A simulation study [13] based on a workload trace collected from an heterogeneous system Eurora [10] reveals that PCP'19 and HCP'19 yield substantial improvements over the original dispatchers [3, 7] and provide a better QoS compared to Eurora's dispatcher [17], which is a part of PBS Professional.

PCP'19 and HCP'19 are, however, not designed for today's large and heterogeneous systems. In PCP'19, the number of decision variables in the CP model increases proportionally to the number of nodes and the possible allocations of jobs in each node. Figure 1 shows where the Eurora system stands compared to today's top 500 systems. As we will show in our experimental results, PCP'19 cannot be used in a larger system like KIT ForHLR II², whose size is comparable to that of the majority of the top systems.

In HCP'19 instead, the problem is decoupled into scheduling and allocation problems. Only the scheduling problem is addressed using CP, and this is done without representing the nodes in the model by treating the resources of the same type across all nodes as a pool of resources. The allocation problem is then solved with a heuristic algorithm using the best-fit strategy [24], while fixing any inconsistencies introduced during scheduling due to the absence of the nodes in the model. The decoupled approach drops the number of decision variables dramatically compared to PCP'19, enabling HCP'19 to scale to larger systems. However, it mainly suits to homogeneous systems where all the nodes have only CPUs and memory, and thus the actual node of an allocated resource is not relevant. In an heterogeneous system, on the contrary, some nodes contain scarce resource types, such as GPUs and MICs, and allocating their CPUs carelessly (i.e., to jobs that do not need any of GPUs and MICs) may cause resource fragmentation [20]. The decoupled approach therefore may result in several iterations between scheduling and allocation in an heterogeneous system, decreasing the dispatching performance, as we will show in our experimental results with Eurora. The advantages of tackling the entire problem using CP, as was done in PCP'19, are that scheduling and allocations decisions are made jointly and that with the presence of nodes in the model, allocation strategies dedicated for heterogeneous systems [20] can be encoded as constraints.

Contributions

We exploit the strengths of PCP'19 and HCP'19 to overcome their limitations. We tackle the entire dispatching problem using CP, and to do that we present a new allocation model where the number of variables is system size independent. We combine this model with the scheduling model common to PCP'19 and HCP'19, and showcase the practical value of our approach. Our contributions are (i) a new HPC application domain emerging from today's large and heterogeneous systems to push the limits of complete methods for optimization, (ii) a novel CP-based online job dispatcher (PCP'21) suitable for such systems (iii) experimental evidence of the benefits of PCP'21 over PCP'19 and HCP'19 supported by a simulation study based on workload traces collected from the Eurora and KIT ForHLR II systems.

² <https://www.scc.kit.edu/dienste/forh1r2.php>

Organization

The rest of the paper is organized as follows. In Section 2, we introduce the on-line job dispatching problem in HPC systems, and describe briefly the CP scheduling and allocation models of PCP'19 as we will later use the same scheduling model in PCP'21 and contrast the allocation model with ours. In Section 3, we present our new CP allocation model and search algorithm. In Sections 4 and 5, we detail our simulation study and present our results. We conclude and describe the future work in Section 6.

2 Formal Background

2.1 On-line job dispatching problem in HPC systems

A *job* is a user request in an HPC system and consists of the execution of a computational application over the system resources. A set of jobs is a *workload*. A job has a name, required resource types (cores, memory, etc) to run the corresponding application, and an *expected duration* which is the maximum time it is allowed to execute on the system. An HPC system typically receives multiple jobs simultaneously from different users and places them in a *queue* together with the other waiting jobs (if there are any). The *waiting time* of a job is the time interval during which the job remains in the queue until its execution time.

An HPC system has N nodes, with each node $n \in N$ having a capacity $cap_{n,r}$ for each resource type $r \in R$. Each job i in the queue Q has an arrival time q_i , maximum number of requested nodes rn_i and a demand $req_{i,r}$ giving the amount of resources required from r during its expected duration d_i . The resource request of i is distributed among rn_i identical job units, preserving for each one $req_{i,r}/rn_i$ amount of resources from r , thus allowing to execute the rn_i job units in parallel. Job units can be tasks that are spanned across multiple nodes and that communicate between them during their entire execution (for instance an MPI job). A specific resource can be used by one job unit only. We have $rn_i = 1$ for serial jobs and $rn_i > 1$ for parallel jobs. The units of a job can be allocated on the same or different nodes, depending on the system availability. On-line job dispatching takes place at a specific time t for (a subset of) the queued jobs Q . The *on-line job dispatching problem* at a time t consists in *scheduling* each job i by assigning it a start time $s_i \geq t$, and *allocating* i to the requested resources during its expected duration d_i , such that the capacity constraints are satisfied: at any time in the schedule, the capacity $cap_{n,r}$ of a resource r is not exceeded by the total demand $req_{i,r}$ of the jobs i allocated on it, taking into account the presence of jobs already in execution. The objective is to dispatch in the best possible way according a measure of QoS, such as with minimum waiting times $s_i - q_i$ or the slowdown ($\frac{s_i - q_i + d_i}{d_i}$) for the jobs, which is directly perceived by the HPC users. A solution to the problem is a *dispatching decision*. Once the problem is solved, only the jobs with $s_i = t$ are dispatched. The remaining jobs with $s_i > t$ are queued again with their original q_i . During execution, a job exceeding its expected duration is killed. It is the workload management system that decides the dispatching time t and the subsequent dispatching times.

2.2 PCP'19 dispatcher

Scheduling model

The scheduling problem is modeled using Conditional Interval Variables (CIVs) [19]. A CIV $\tau_i \in \tau$ represents a job i and defines the time interval during which i runs. At a dispatching time t , there may already be jobs in execution which were previously scheduled and allocated.

We refer to such jobs as running jobs. The scheduling model considers in the τ variables both the running jobs and a subset $\bar{Q} \subseteq Q$ of the queued jobs that can start execution as of time t . The properties $s(\tau_i)$ and $d(\tau_i)$ correspond respectively to the start time and the duration of the job i . All job units of a job i start at the same time, therefore they share the same τ_i . Since the actual runtime (real) duration d_i^r of a running or queued job i is unknown at the modeling time, PCP'19 uses an *expected duration* d_i for $d(\tau_i)$, which is supplied by a job duration *prediction method*. For the queued jobs, we have $d(\tau_i) = d_i$. For the running jobs instead, $d(\tau_i) = \max(1, s(\tau_i) + d_i - t)$ taking into account the possibility that $d_i < d_i^r$ due to underestimation. While the start time of the running jobs have already been decided, the queued jobs have $s(\tau_i) \in [t, eoh]$, where *eoh* is the end of the worst-case makespan calculated as $t + \sum_{\tau_i} d(\tau_i)$.

The capacity constraints are enforced via $\text{cumulative}([s(\tau_i)], [d(\tau_i)], [req_{i,r}], Tcap_r)$, for all $n \in N$ and for all $r \in R$, with $Tcap_r = \sum_n cap_{n,r}$. The objective function minimizes the total job slowdown $\sum_{\tau_i} \frac{s(\tau_i) - q_i + d(\tau_i)}{d(\tau_i)}$. The search for solutions focuses on the jobs with highest priority where the *priority* of a job i is its slowdown $\frac{t - q_i + d(\tau_i)}{d(\tau_i)}$ at the dispatching time t . We note that HCP'19 uses the same scheduling model and search.

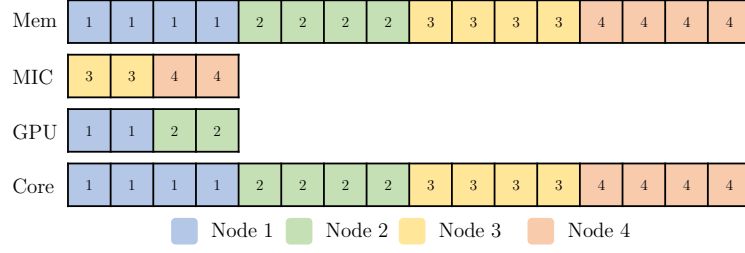
Allocation model

The allocation model replicates each τ_i variable $p_{i,n}$ times for each $n \in N$, where $p_{i,n} = \min(rn_i, \min_{r \in R} \lfloor \frac{cap_{n,r}}{req_{i,r}/rn_i} \rfloor)$ giving the minimum times a job unit can fit on n . The variable $u_{i,n,j}$ represents a possible allocation of a job unit j of i on node n and has $s(u_{i,n,j}) = s(\tau_i)$ and $d(u_{i,n,j}) = d(\tau_i)$. To define the allocation, the model relies on the execution state property (x) of CIVs. We have $x(u_{i,n,j}) \in [0, 1]$, meaning that it can be present or not in the solution. Instead for the scheduling variables we have $x(\tau_i) = 1$ because all of them need to be scheduled and thus be present in the solution. The model uses the **alternative** constraint [19] to restrict the number of variables in $\cup_{n \in N} [x(u_{i,n,j})]$ present in the solution to be the maximum number of requested nodes rn_i , that is $\sum_{n \in N} \sum_j x(u_{i,n,j}) = rn_i$ with $s(\tau_i) = s(u_{i,n,j})$ iff $x(u_{i,n,j}) = 1$. Additionally, the capacity constraints are enforced for each $n \in N$ and for each $r \in R$ as $\text{cumulative}([s(u_{i,n,j})], [d(u_{i,n,j})], [req_{i,r}/rn], cap_{n,r})$.

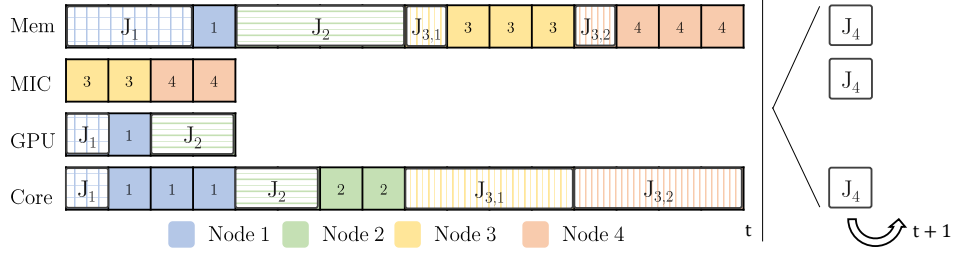
A drawback of this model is its number of variables. While the scheduling model has $|\bar{Q}|$ variables, the allocation model has $\sum_{i \in \bar{Q}} \sum_{n \in N} p_{i,n}$ variables, which increases proportionally to N (i.e., system size). A minimum of $1 + |N|$ variables are needed to model a serial job. Parallel jobs will require even more variables which may create difficulty in large systems with many parallel jobs.

3 PCP'21: a New CP-based Job Dispatcher

Our dispatcher PCP'21 imports the scheduling model, the objective function and the job priorities of PCP'19 and contains a new allocation model with $|\bar{Q}| + \sum_{i \in \bar{Q}} rn_i * |R|$ variables, which is system size independent. The number of variables thus depends on the number of resource types (which is a small value) multiplied by the sum of the requested nodes (which is usually much smaller than the system size) of all jobs in \bar{Q} (which is a fixed value). Next, we present the allocation model and describe how we search on the scheduling and the allocation variables.



■ **Figure 2** Representation of an example system.



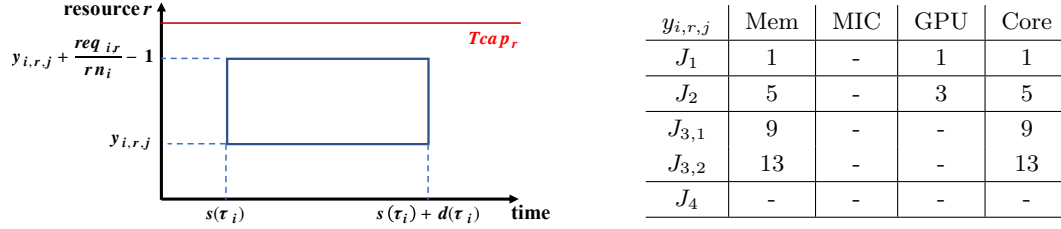
■ **Figure 3** Allocation of some jobs on the example system at a dispatching time t .

Allocation model

In this new model, we represent the system in a way to emphasize the resources instead of the nodes as in the previous model. We consider all the resources of a certain type r in an ordered array by following the sequence of the nodes. This is exemplified in Figure 2 which represents a system with 4 nodes. Each node has 4 cores and 4 units of memory. The first two nodes have 2 GPUs, and the next two has 2 MICs. The array labelled as GPU, for instance, lists all the GPU resources available in the system. There are in total $2 * 2$ GPUs, the first two in the array are from the first node, the third and the forth from the second node. An array position refers to a specific resource of type r in a node n , which is highlighted with a colour and a number in Figure 2.

Let us assume that, at a dispatching time t , a job J_1 is still running, and three more jobs J_2, J_3, J_4 are extracted from the queue. As for their resource requests, let us assume that J_1 requires 3 memory units, 1 GPU and 1 core; J_2 4 memory units, 2 GPUs and 2 cores; J_3 2 memory units and 8 cores via two job units $J_{3,1}$ and $J_{3,2}$; and J_4 1 memory unit, 1 MIC and 1 core. Figure 3 shows a possible allocation of these jobs in the system after a dispatcher call at time t . The running job J_1 is allocated to the minimum positions available in the arrays corresponding to the requested resource types, hence it is allocated in node 1 (the first available node), occupying the first 3 memory, the first GPU and the first core positions in the corresponding arrays. Since J_2 requires two GPUs, it is allocated in the second node, occupying all the memory and GPU and the first two core positions in the green parts of the resp. arrays. The job units of J_3 do not fit in the same node, so they are equally distributed to the next two nodes, each occupying the first memory and all the core positions of the yellow and orange parts of the resp. arrays. As for J_4 , it can be allocated only in the last two nodes, because it needs an MIC. As the cores of these nodes are all occupied, J_4 cannot be allocated and is postponed to the next dispatching time $t + 1$.

Following this representation, we model the positions of a job unit j of a queued job i on a resource type r via the variables $y_{i,r,j}$. As we also need to represent the time during which the allocation is valid, we use a two-dimensional box to model an allocation, as depicted in



■ **Figure 4** Modelling the allocation of a job unit j of a job i on a resource type r (left), and the assignment of the $y_{i,r,j}$ variables at time t in the example allocation (right).

Figure 4 (left). The y-axis gives the available positions and the x-axis gives the time interval during which the resource is consumed. The vertices of the box are defined by the variables in the origin: $s(\tau_i)$ which is the starting time of the job i and $y_{i,r,j}$ which is the starting position of the allocation. The box spans from the origin by the expected duration $d(\tau_i)$ in the x-axis, instead in the y-axis by $req_{i,r}/rn_i$ which is the required resource amount. As for the domains, we have $D(y_{i,r,j}) = [1, Tcap_r]$, where $Tcap_r = \sum_{n \in N} cap_{n,r}$. The domain of the starting time is the same as in the scheduling model, that is $D(s(\tau_i)) = [t, eoh]$.

Figure 4 (right) shows the assignment of the $y_{i,r,j}$ variables at time t in our example allocation. Take for instance J_2 which has one job unit (itself) and requires 4 memory units, 2 GPU and 2 cores. As we saw in Figure 3, it occupies in the Memory array the 5th to the 9th positions, in the GPU array the 3rd to the 5th, and in the Core array the 5th to the 7th. Consequently, the corresponding $y_{i,r,j}$ variables are assigned to the starting positions 5, 3 and 5, and the relative boxes span in the y-axis to the last positions 9, 5, and 7, respectively.

We also need to model the running jobs. To a job unit j of a job i on a resource type r , which was previously assigned the resources of a certain node, we now assign the minimum available position among those that refer to the same node. We have already exemplified this with J_1 in Figure 3. These resources are allocated to i during its $d(\tau_i)$. If multiple job units are assigned to the same node, the resources are occupied consecutively, leaving the higher indices free.

To enforce that a resource is used by one job unit only, we forbid the boxes to overlap via the `diffn` constraint [4]. For each $r \in R$, we have `diffn` $([s(\tau_i)], [d(\tau_i)], [y_{i,r,j}], [req_{i,r}/rn_i])$. As the domain size of the $y_{i,r,j}$ variables depends on the system size and can be large, we add implied constraints to shrink them. They are the classical `cumulative` constraints used together with a `diffn` constraint in packing problems, as was also done in [4]: `cumulative` $([s(\tau_i)], [d(\tau_i)], [req_{i,r}/rn_i], Tcap_r)$ and `cumulative` $([y_{i,r,j}], [req_{i,r}/rn_i], [s(\tau_i)], eoh)$. Given that the jobs units of jobs with $rn_i > 1$ have identical resource requests, their allocations are symmetric. We post an ordering constraint on the positions of the jobs unit of a job i on a resource type r to break symmetry: $y_{i,r,j} < y_{i,r,j+1}$. It is a strict ordering as the position variables take different values.

Finally, we need additional constraints to guarantee that certain allocations are in the same node. For that, we utilize a mapping array map_r for each resource type r , which is based on the system representation introduced earlier. The positions of map_r correspond to the available resources, indexed by 1 to $Tcap_r = \sum_{n \in N} cap_{n,r}$, and each value in the array is a number corresponding to a node. To ensure that the allocated resources of a job unit are in the same node, we post an `element` constraint, which indexes an array with a variable, as `element` $(map_{r_1}, y_{i,r_1,j}) = \text{element}(map_{r_2}, y_{i,r_2,j}) \forall r_1, r_2 \in \hat{R}$, where \hat{R} is the set of the

requested resource types of the job unit j of job i . We use the `element` constraint also to enforce that the positions spanning from $y_{i,r,j}$ to $y_{i,r,j} + (req_{i,r}/rn_i) - 1$ refer to the same node: $\text{element}(map_r, y_{i,r,j}) = \text{element}(map_r, y_{i,r,j} + (req_{i,r}/rn_i) - 1) \forall r \in \hat{R}$ iff $req_{i,r}/rn_i > 0$.

Search

Similarly to PCP'19 and HCP'19, we use a custom search algorithm derived from the schedule-or-postpone algorithm [21] to search on the scheduling variables $s(\tau_i)$. At each decision node, we select the job i whose priority is highest and that can start first, and assign to $s(\tau_i)$ its earliest start time $\min(s(\tau_i))$. Note that the priorities are calculated once statically at the dispatching time t before search starts.

Differently from PCP'19 and HCP'19, we interleave the scheduling and the allocation assignments of a selected job i . After assigning a scheduling variable $s(\tau_i)$, we search on the allocation variables $[y_{i,r,j}]$ of i . We start with the resource r which has the lowest availability at time t . Then we search on $[y_{i,r,j}]$ in lexicographical order and assign them their minimum values $\min(y_{i,r,j})$, which guarantees consistency with the symmetry breaking constraints on the allocation variables.

Even though we have designed PCP'21 for systems engaged with heterogeneous workloads, an heterogeneous system may as well receive a workload with homogeneous resource requests (i.e., only CPU and memory) which creates symmetry among the requested resources. We adapt the search algorithm to break symmetry in such a scenario as follows. After a resource allocation attempt for a job i , if the search fails or wants to find a better solution, it backtracks to the scheduling variable $s(\tau_i)$, as opposed to backtracking within the $[y_{i,r,j}]$ variables.

Following PCP'19 and HCP'19, search is bounded by a time limit δ due to the problem complexity. Thus, the best solution returned within the limit is the dispatching decision. If, however, no satisfiability answer is obtained within the limit, the time limit is extended to $2 * \delta$, as opposed to restarting search with the new time limit $2 * \delta$ as was done in PCP'19 and HCP'19. This procedure continues until the time limit reaches δ_{max} . We suspend the search if the solution quality did not change after k consecutive time limit extensions.

4 Experimental Study

To evaluate the significance of our approach, we conducted an experimental study by simulating on-line job submission to two HPC systems. We dispatched jobs using PCP'21, PCP'19, HCP'19, and sought answers to the following questions: (1) how do the dispatchers compare when they are engaged in a workload with heterogeneous resource requests? (2) can PCP'19 and PCP'21 scale to a large system? As we said earlier, an heterogeneous system may as well receive a workload with homogeneous resource requests. We thus sought an answer also to the following question: (3) how much do we lose by using PCP'21 for a workload with homogeneous resource requests compared to using HCP'19 which is more suitable for an homogeneous system? Before we present the answers in Section 5, we describe in this section the ingredients of our experimental study.

HPC systems and workload datasets

Our study is based on two different workload traces collected from two different HPC systems. The first system is the Eurora [10], which was in production at CINECA datacenter in Italy until 2015. With 64 nodes, the system size is small compared to the current trend (see Figure 1), but the architecture is heterogeneous with each node containing 2 octa-core CPUs,

16 GB memory, and two of GPU or MIC. To answer the first question, we use the workload dataset with which PCP'19 and HCP'19 were tested in [13]. It consists of logs over 400,000 jobs submitted during the time period March 2014–August 2015 and is dominated by short jobs, making up 93.14% of all the jobs. As for resource requests, 22.8% of the jobs require only CPU and memory while 77.2% need in addition one of GPU or MIC.

The second system is the KIT ForHLR II², located at Karlsruhe Institute of Technology in Germany. We use this system to answer the second question because it has 1,173 nodes, a size comparable to the current trend (see Figure 1). 1,152 of these nodes are thin, each equipped with 20 cores and 64 GB memory, and the remaining 21 are fat each containing 48 cores, 4 GPUs, and 1 TB memory. Even though a small fraction of the nodes contain GPUs, we use a workload with homogeneous resource requests to answer also the third question. The workload dataset is available on-line.³ It contains logs for 114,355 jobs submitted during the time period June 2016–January 2018. All the jobs require only CPU and memory, and 66.26% of them are short (< 1h).

Job duration prediction

We derived the expected durations d_i of jobs via three prediction methods. The first is a data-driven heuristic first proposed in [14] and later was shown to work well with PCP'19 and HCP'19 when simulating the Eurora dataset [13]. The heuristic constructs job profiles from the workload. Prediction is based on the observation that jobs with similar profiles have the same duration for long periods of time. For each job, the heuristic searches for the last job with a similar profile, and uses its duration to predict the duration of the new one. Each user is analyzed separately. The similar profile is identified using a set of rules. If all rules fail, then the user-declared wall-time is taken as the predicted duration. In all cases, the prediction is capped by the wall-time.

Despite being a valid alternative, this method relies on job names, a type of data omitted in the KIT ForHLR II and some other public datasets. We thus employed a second heuristic method that uses the run times of the last two jobs to predict the duration of the next job [26]. In both methods, the predictions are calculated on-line during the simulation and the knowledge base is updated upon job termination. The last prediction method is an oracle which gives the actual runtime (real) durations d_i^r and provides a baseline during the simulation of both datasets.

Simulation

We used the AccaSim workload management system simulator [12] to simulate the HPC systems with their workload datasets. Each job submission is simulated by using its available data, for instance, the owner, the requested resources, and the real duration, the execution command or the name of the application executed. AccaSim uses the real duration to simulate the job execution during its entire duration. Therefore job duration prediction errors do not affect the running time of the jobs with respect to the real workload data. The dispatchers are implemented using the AccaSim directives to allow them to generate the dispatching decisions during the system simulation.

³ <https://www.cse.huji.ac.il/labs/parallel/workload/logs.html>

■ **Table 1** Times obtained from the Eurora system.

Dispatcher	Avg. disp. time [ms]	Total sim. time [s]
HCP'19-D	392	208,231
PCP'19-D	511	271,586
PCP'21-D	209	111,373
HCP'19-R	357	189,522
PCP'19-R	469	249,367
PCP'21-R	256	136,401

Experimental settings

As a CP modelling and solving toolkit, we customized Google OR-Tools⁴ 7.3 by implementing the `alternative` constraint and the proposed search algorithm and by making visible some variables of the solver, and ported it to Python 3.6 to implement PCP'21 in AccaSim. As for PCP'19 and HCP'19, we used their publicly available implementations⁵, and carried over their parameters $m = 100$, $\delta = 1s$, $\delta_{max} = 16s$, $k = 2$. For the simulation of the KIT ForHLR II workload, which has only homogeneous resource requests, we adapted the search algorithm of PCP'21 to break symmetry among the requested resources as described in Section 3. We refer to this version of PCP'21 as PCP'21_s in the experimental results. All experiments were performed on a CentOS machine equipped with Intel Xeon CPU E5-2640 Processor and 16GB of RAM. The source code is publicly available at <https://git.io/fjia1>.

5 Experimental Results

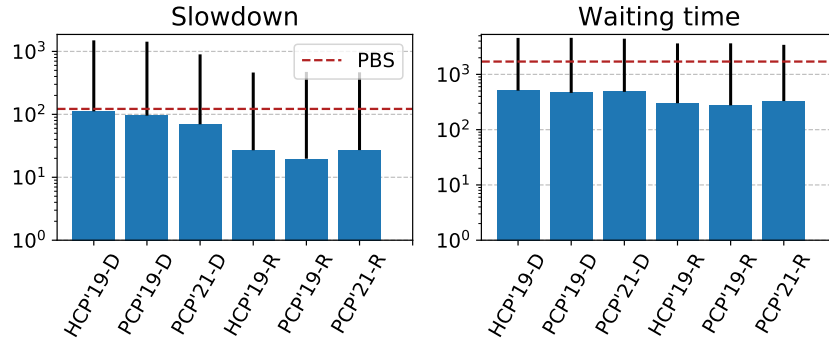
In this section, we show our experimental results. In each simulation, we compare the dispatchers' performance (in Tables 1 and 2) in terms of (i) the average CPU time spent in generating a dispatching decision over all dispatcher invocations, including the time for modeling the dispatching problem instance and searching for a solution, and (ii) the total simulation time from the first job submission until the last job completion. We also compare the dispatchers' QoS (in Figures 5 and 10) in terms of the average slowdown and waiting times of the jobs. To refer to a dispatcher using a certain job duration prediction method, we append -D, -L2 or -R to the name of the dispatcher for the data-driven heuristic, the last-two heuristic and the real duration, respectively.

5.1 Simulation of the Eurora workload

We remind that we simulate a system like Eurora to compare all the dispatchers when they are engaged in a workload with heterogeneous resource requests. All the dispatchers complete the simulation. Comparing their performance in Table 1, we can clearly see the benefits of using PCP'21. With the decoupled approach of HCP'19, the performance drops almost by half (around 47%) when using -D. We observe a further performance decrease (around 59%) with PCP'19, which could be attributed to its higher number of decisions variables. PCP'21 is the most efficient dispatcher also when using -R, with gains around 28% and 45% compared to HCP'19 and PCP'19, resp.

⁴ <https://developers.google.com/optimization/>

⁵ <https://git.io/fjia1>



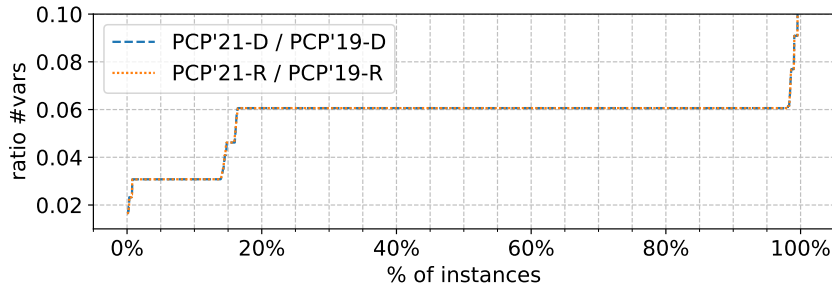
■ **Figure 5** Average and error bars showing one std. deviation of slowdown and waiting times [s] obtained from the Eurora system.

As for the quality of decisions, we observe in Figure 5 that all dispatchers return better solutions than Eurora's PBS dispatcher. Among the CP-based dispatchers, PCP'21 results in the best average slowdown and a substantial decrease in the error when using -D. Otherwise, the dispatchers have similar (low) slowdown values when using -R and have similar waiting times when using either of job duration prediction methods.

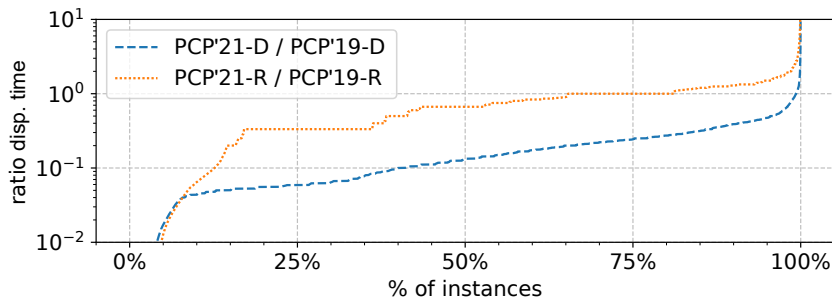
As a side comment, we note in Table 1 that PCP'21-D performs better than PCP'21-R. Looking at Figure 5, however, we see that PCP'21-R finds better solutions than PCP'21-D within the time limit. In the instances of -D, jobs have similar durations (due to the way the data-driven prediction method works), instead in the instances of -R, jobs have a more diverse duration. The -R instances tend to be more difficult and take longer to solve, especially with PCP'21-R which uses the expected durations also in the allocation model.

Additional analysis is needed in order to quantify the reduction in the number of decision variables obtained by going from PCP'19 to PCP'21. During the simulation of an HPC system and its workload data, all dispatchers start with the same dispatching instance, but then they schedule and allocate jobs diversely. This in turn leads to different jobs running on different resources of the system as well as to different jobs waiting in the queue in the next dispatching time. We cannot therefore compare the dispatchers' model size on the distinct instances they entail throughout the simulation period. To analyze the dispatchers on the same instances, we saved the instances created during the simulation of the Eurora workload while using PCP'19-D and PCP'19-R as a dispatcher. Each instance is created when the simulator calls the corresponding dispatcher, and the instance is described by the queued jobs, the running jobs and their specific allocation on the system. We obtained in total 624,569 instances.

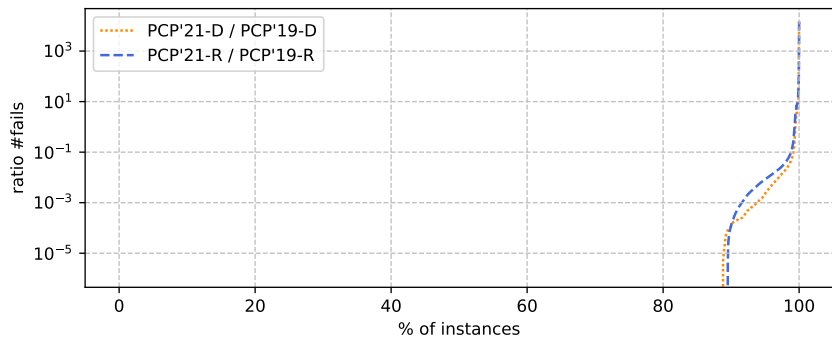
Figure 6 shows the ratio of the number of decision variables between PCP'21 and PCP'19 versus the percentage of the instances. For all instances, the ratio is below 0.1, proving the significance of the new allocation model. To confirm the impact on the search performance, we give in Figures 7 and 8, the ratio of the dispatching time and the number of fails. We note that while some instances are solved to optimality, some instances hit the time limit but even in that case PCP'19 and PCP'21 extend the time limit differently, as was described in Section 3. For almost all the instances, the ratio of the dispatching time is between 1 and 0.01, and the ratio of the number of fails is between 0.1 and 0, supporting the direct effect of model size on the dispatcher performance. In Figure 9, we show the ratio of the dispatching time versus the ratio of the number of fails for each individual instance. 93% and 88% of the instances fall into the region where both ratios are between 0 and 1 when using -D and -R, respectively.



■ **Figure 6** Ratio of the #decision variables between PCP'21 and PCP'19 on the individual Eurora instances.



■ **Figure 7** Ratio of the dispatching time between PCP'21 and PCP'19 on the individual Eurora instances.

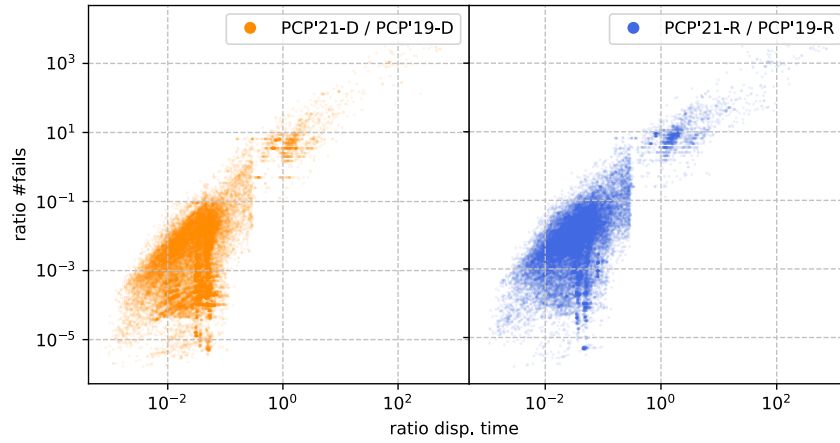


■ **Figure 8** Ratio of the #fails between PCP'21 and PCP'19 on the individual Eurora instances.

We also analyzed the ratio of the quality of the dispatching decisions. The results (not shown here) are in line with those shown in Figure 5. The ratio is 1 for the vast majority of the instances.

5.2 Simulation of the KIT ForHLR II workload

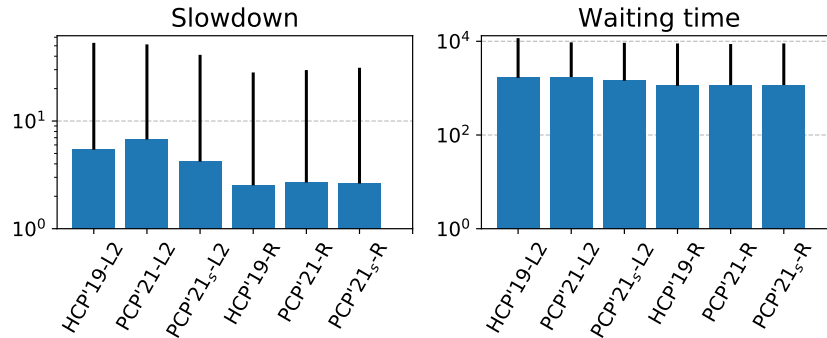
We remind that we simulate a system like KIT ForHLR II to observe whether PCP'19 and PCP'21 can scale to a large system. We do it so by using a workload with homogeneous resource requests, because an heterogeneous system may as well receive an homogeneous workload and in that case we want to quantify any possible loss with respect to HCP'19 which is more suitable for an homogeneous system. We experiment with both PCP'21 and PCP'21_s to see the importance of symmetry breaking with an homogeneous workload.



■ **Figure 9** Ratio of the dispatching time (x-axis) vs ratio of the #fails (y-axis) between PCP'21 and PCP'19 on the individual Eurora instances.

■ **Table 2** Times obtained from the KIT ForHLR II system.

Dispatcher	Avg. disp. time [ms]	Total sim. time [s]
HCP'19-L2	278	56,083
PCP'19-L2	∞	∞
PCP'21-L2	493	99,590
PCP'21 _s -L2	334	67,471
HCP'19-R	269	54,289
PCP'19-R	∞	∞
PCP'21-R	476	96,030
PCP'21 _s -R	342	69,094



■ **Figure 10** Average and error bars showing one std. deviation of slowdown and waiting times [s] obtained from the KIT ForHLR II system.

PCP'19 cannot complete the simulation for several days. At some point in time, it stops dispatching, even if new jobs are entering in the queue and the system is empty with all its resources available. This is because PCP'19 cannot handle certain dispatching instances within the available time limit and blocks the current and the next dispatching decisions. Instead PCP'21 and PCP'21_s complete the simulation, as can be seen in Table 2, confirming its advantage to PCP'19 in a large system. We observe in the table that symmetry breaking

is crucial with an homogeneous workload. PCP'21_s significantly reduces the total simulation time and average dispatching time compared to PCP'21. The performance of PCP'21_s is not too far from that of HCP'19. The performance gap is around 27% and 20% when using -R and -D, resp. We can see in Figure 10 that in terms of the QoS, the dispatchers behave almost identically.

5.3 Discussion

We showed that when dispatching an Eurora workload dominated by short jobs with heterogeneous resource requests, PCP'21 is more efficient than the other dispatchers by 28% to 59%. While PCP'21 can scale to a large system like KIT ForHLR II, PCP'19 cannot. This is probably due to the high number of decision variables in the allocation model of PCP'19. Additional experiments on the individual Eurora instances generated by PCP'19 confirmed that PCP'21 substantially reduces the number of variables, the dispatching time, and the number of fails. We also argued that an heterogeneous system may as well receive an homogeneous workload. We showed that when dispatching a KIT ForHLR II workload dominated by short jobs with homogeneous resources requests, the use of an adapted version of the search algorithm that breaks symmetry among the identical resources is crucial. With this version of PCP'21 (called PCP'21_s), the performance loss relative to HCP'19 is limited to 27%. Our results thus provide evidence for the significance of our approach in dispatcher performance in a large or heterogeneous system running modern applications.

While we have used real data representing the workload of modern systems and applications, our conclusions are based on a simulation study which is restricted by the capabilities of the simulator. For instance, AccaSim does not add the dispatching time to the waiting times of jobs. This seems to be the reason why we have not observed noteworthy gains with PCP'21 in the QoS. In a real system, jobs' waiting time (and slowdown) would increase during dispatching time, therefore dispatcher performance would directly affect the QoS.

6 Conclusions and Future Work

Constraint Programming (CP) has been successfully applied to solve the on-line job dispatching problem in HPC systems [3, 7] including those running modern applications [13]. We argued that the limitations of the available CP-based job dispatchers may hinder their practical use in today's systems that are becoming larger in size and more heterogeneous in their computing resources. In an attempt to bring CP closer to a deployed application, we presented a new CP-based on-line job dispatcher for HPC systems (PCP'21). Unlike its predecessors, PCP'21 solves the entire problem using CP and its model size is independent of the system size. Experimental results based on a simulation study show that our approach can bring about significant performance gains over the existing dispatchers in a large or heterogeneous system.

In future work, we will devise and experiment with a meta-dispatcher that can switch between PCP'21 and HCP'19 depending on the workload type. Moreover, we will investigate the impact of performance in the QoS of a dispatcher by adapting the simulator to take into account the dispatching time in the calculation of the job waiting time. To improve the dispatcher performance further, we will study breaking the symmetry among the identical nodes (i.e. the nodes that have the same resource availability at a dispatching time t) and dominance breaking during search. We will also investigate whether large neighbourhood search can be beneficial. Towards our objective to deploy and evaluate a CP-based dispatcher in a real system, we plan to encode as constraints the allocation strategies proposed for heterogeneous systems [20].

References

- 1 Altair. Altair PBS professional (accessed may 27 2021), 2021. URL: <https://www.altair.com/pbs-works/>.
- 2 Philippe Baptiste, Philippe Laborie, Claude Le Pape, and Wim Nuijten. Chapter 22 - constraint-based scheduling and planning. In *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 761–799. Elsevier, 2006.
- 3 Andrea Bartolini, Andrea Borghesi, Thomas Bridi, Michele Lombardi, and Michela Milano. Proactive workload dispatching on the EURORA supercomputer. In *Proceedings of Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014.*, volume 8656 of *Lecture Notes in Computer Science*, pages 765–780. Springer, 2014. doi:10.1007/978-3-319-10428-7_55.
- 4 Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, 1994. doi:10.1016/0895-7177(94)90127-9.
- 5 Jacek Blazewicz, Jan Karel Lenstra, and A. H. G. Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24, 1983. doi:10.1016/0166-218X(83)90012-4.
- 6 A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini. Scheduling-based power capping in high performance computing systems. *Sustainable Computing: Informatics and Systems*, 19:1–13, 2018.
- 7 Andrea Borghesi, Francesca Collina, Michele Lombardi, Michela Milano, and Luca Benini. Power capping in high performance computing systems. In *Proceedings of 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 524–540. Springer, 2015. doi:10.1007/978-3-319-23219-5_37.
- 8 Thomas Bridi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. A constraint programming scheduler for heterogeneous high-performance computing machines. *IEEE Trans. Parallel Distrib. Syst.*, 27(10):2781–2794, 2016.
- 9 Jirachai Buddhakulsomsiri and David S. Kim. Priority rule-based heuristic for multi-mode resource-constrained project scheduling problems with resource vacations and activity splitting. *European Journal of Operational Research*, 178(2):374–390, 2007. doi:10.1016/j.ejor.2006.02.010.
- 10 Carlo Cavazzoni. EURORA: a european architecture toward exascale. In *Proceedings of the Future HPC Systems - the Challenges of Power-Constrained Performance, FutureHPC@ICS 2012, Venezia, Italy, June 25, 2012*, pages 1:1–1:4. ACM, 2012. doi:10.1145/2322156.2322157.
- 11 Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling - A status report. In *Job Scheduling Strategies for Parallel Processing, 10th International Workshop, JSSPP 2004, New York, NY, USA, June 13, 2004, Revised Selected Papers*, volume 3277 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2004. doi:10.1007/11407522_1.
- 12 Cristian Galleguillos, Zeynep Kiziltan, Alessio Netti, and Ricardo Soto. Accasim: a customizable workload management simulator for job dispatching research in HPC systems. *Cluster Computing*, 23(1):107–122, 2020. doi:10.1007/s10586-019-02905-5.
- 13 Cristian Galleguillos, Zeynep Kiziltan, Alina Sirbu, and Özalp Babaoglu. Constraint programming-based job dispatching for modern HPC applications. In *Proceeding of Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019*, volume 11802 of *Lecture Notes in Computer Science*, pages 438–455. Springer, 2019. doi:10.1007/978-3-030-30048-7_26.
- 14 Cristian Galleguillos, Alina Sirbu, Zeynep Kiziltan, Özalp Babaoglu, Andrea Borghesi, and Thomas Bridi. Data-driven job dispatching in HPC systems. In *Proceedings of Machine Learning, Optimization, and Big Data - Third International Conference, MOD 2017, Volterra, Italy, September 14-17, 2017, Revised Selected Papers*, volume 10710 of *Lecture Notes in Computer Science*, pages 449–461. Springer, 2017. doi:10.1007/978-3-319-72926-8_37.

- 15 R. Haupt. A survey of priority rule-based scheduling. *Operations-Research-Spektrum*, 11(1):3–16, March 1989. doi:10.1007/BF01721162.
- 16 Stijn Heldens, Pieter Hijma, Ben van Werkhoven, Jason Maassen, Adam S. Z. Belloum, and Rob van Nieuwpoort. The landscape of exascale research: A data-driven literature analysis. *ACM Comput. Surv.*, 53(2):23:1–23:43, 2020. doi:10.1145/3372390.
- 17 Robert L. Henderson. Job scheduling under the portable batch system. In *Proceedings of Job Scheduling Strategies for Parallel Processing, IPPS’95 Workshop, Santa Barbara, CA, USA, April 25, 1995.*, volume 949 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 1995. doi:10.1007/3-540-60153-8_34.
- 18 ITIF. The vital importance of high-performance computing to u.s. competitiveness. information technology and innovation foundation. (accessed september 4, 2020), 2016. URL: <http://www2.itif.org/2016-high-performance-computing.pdf>.
- 19 Philippe Laborie and Jerome Rogerie. Reasoning with conditional time-intervals. In *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference, May 15-17, 2008, Coconut Grove, Florida, USA*, pages 555–560. AAAI Press, 2008. URL: <http://www.aaai.org/Library/FLAIRS/2008/flairs08-126.php>.
- 20 Alessio Netti, Cristian Galleguillos, Zeynep Kiziltan, Alina Sîrbu, and Özalp Babaoglu. Heterogeneity-aware resource allocation in HPC systems. In *Proceedings of High Performance Computing - 33rd International Conference, ISC High Performance 2018, Frankfurt, Germany, June 24-28, 2018*, volume 10876 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2018. doi:10.1007/978-3-319-92040-5_1.
- 21 C. Le Pape, P. Couronne, D. Vergamini, and V. Gosselin. Time-versus-capacity compromises in project scheduling. *AISB Quartetly*, pages 19–31, 1995.
- 22 PRACE. The scientific case for computing in europe 2018-2026. prace scientific steering committee. (accessed september 4, 2020), 2018. URL: <https://prace-ri.eu/wp-content/uploads/2019/08/PRACEscientificCase.pdf>.
- 23 Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, and Jeremy Kepner. Scalable system scheduling for HPC and big data. *J. Parallel Distributed Comput.*, 111:76–92, 2018. doi:10.1016/j.jpdc.2017.06.009.
- 24 Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, 9th Edition*. Wiley, 2014.
- 25 SLURM. SLURM workload manager, 2019. URL: <http://slurm.schedmd.com>.
- 26 Dan Tsafir, Yoav Etsion, and Dror G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Trans. Parallel Distrib. Syst.*, 18(6):789–803, 2007. doi:10.1109/TPDS.2007.70606.
- 27 Rinki Tyagi and Santosh Kumar Gupta. A survey on scheduling algorithms for parallel and distributed systems. In *Silicon Photonics & High Performance Computing*, pages 51–64, Singapore, 2018. Springer Singapore.

The Dungeon Variations Problem Using Constraint Programming

Gaël Glorian ✉🏠

LaBRI – CNRS UMR 5800, Université de Bordeaux, Talence, Nouvelle-Aquitaine, France
Ubisoft, Bordeaux, Nouvelle-Aquitaine, France

Adrien Debesson ✉

Ubisoft, Bordeaux, Nouvelle-Aquitaine, France

Sylvain Yvon-Palio ✉

Ubisoft, Bordeaux, Nouvelle-Aquitaine, France

Laurent Simon ✉

LaBRI – CNRS UMR 5800, Université de Bordeaux, Talence, Nouvelle-Aquitaine, France

Abstract

The video games industry generates billions of dollars in sales every year. Video games can offer increasingly complex gaming experiences, with gigantic (but consistent) open worlds, thanks to larger and larger teams of developers and artists. In this paper, we propose a constraint-based approach for procedural dungeon generation in an open world/universe context, in order to provide players with consistent, open worlds with an excellent quality of storytelling. Thanks to a global description capturing all the possible rooms and situations of a given dungeon, our approach allows enumerating variations of this global pattern, which can then be presented to the player for more diversity. We formalise this problem in constraint programming by exploiting a graph abstraction of the dungeon pattern structure. Every path of the graph represents a possible variation matching a given set of constraints. We introduce a new propagator extending the “connected” graph constraint, which allows considering directed graphs with cycles. We show that thanks to this model and the proposed new propagator, it is possible to handle scenarios at the forefront of the game industry (AAA+ games). We demonstrate that our approach outperforms non-specialised solutions consisting of filtering only the relevant solutions *a posteriori*. We then conclude and offer several exciting perspectives raised by this approach to the *Dungeon Variations Problem*.

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

Keywords and phrases constraint programming, video games, modelization, procedural generation

Digital Object Identifier 10.4230/LIPIcs.CP.2021.27

Supplementary Material *Software (Source Code)*: <https://bit.ly/3xSXK2B>

Funding This work was supported by the “KIWI” project of the Nouvelle-Aquitaine region.

1 Introduction

The video game industry is an important economic sector, generating billions of dollars each year, at the forefront of innovation. Since the appearance of the first games in the 1970s, the landscape of this industry has changed dramatically. Video games are now produced by large teams of artists and developers, offering photo-realistic graphics, exciting and complex scenarios, with a constantly improved degree of simulation.

One of the biggest challenges of the gaming industry is to be able to build open worlds, in which users must feel free and where a huge number of actions are possible for them. Generating such a world without the final intervention of a *Level Designer (LD)* to validate the level is still a dream: a single inconsistent game level can quickly shatter the reputation of a game, possibly costing millions of dollars. The role of *LD* is to ensure the consistency of all playable levels, and the interest of every level w.r.t the overall scenario. The formal



© Gaël Glorian, Adrien Debesson, Sylvain Yvon-Palio, and Laurent Simon;
licensed under Creative Commons License CC-BY 4.0

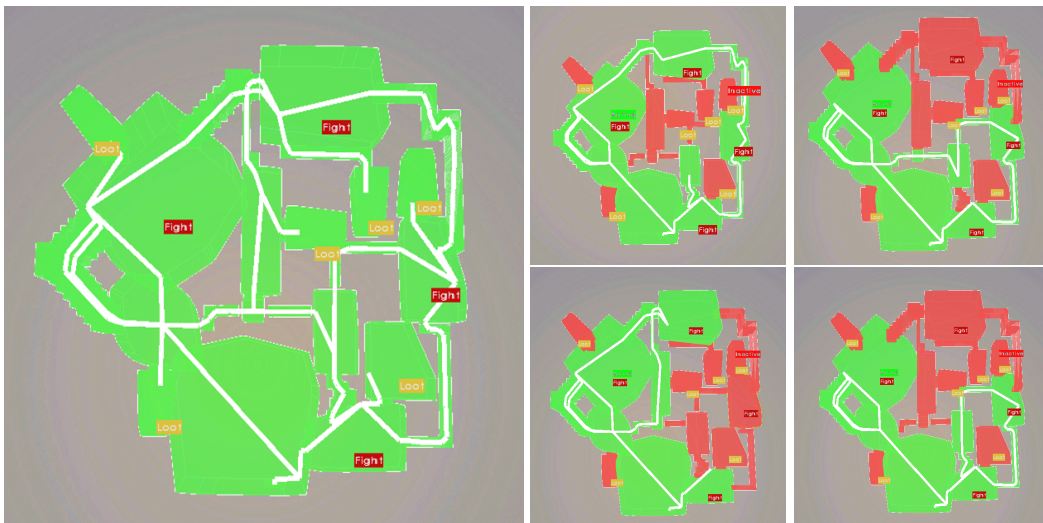
27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 27; pp. 27:1–27:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** A source dungeon.

■ **Figure 2** Some variations of the source dungeon (Figure 1). In green, the rooms and corridors preserved in the variations.

verification of generated levels will probably play an important role in the future of the gaming industry, but, for now, it is still impossible. Instead of formally verify the properties of the generated levels *a posteriori*, we propose, in this article, to enforce the desired properties of generated levels by construction. This work is developed in collaboration with an AAA+ video game studio.

Intuitively, in the proposed approach, an *LD* produces what we call a *source dungeon*, which contains a set of rooms connected by corridors. Each room has its properties and areas identified for possible situations (fights, treasures, ...). The *Dungeon Variations Problem* is the problem of generating an appropriate variation of the source dungeon by disabling a subset of initial rooms and corridors so that the remaining set of rooms matches a given set of constraints (e.g. at least one entrance, at least a given number of monsters on any path). The consistency of a source dungeon can also be validated by generating a variation taking into account all rooms as well as corridors.

While this approach does not yet fully address the global problem of procedural content generation [19], it opens up a challenging new field for constraint programming. Additionally, it allows level designers to check the consistency and quality of dungeon variations before delivering the game. We formally specify the *Dungeon Variations Problem* in Section 2 and describe it as a constraint programming problem using the graph constraint *connected* in Section 3. We then introduce, in Section 4, a new propagator extending this constraint (*connected+*) which takes into account directed graphs with cycles. In the experimental part (Section 5), we show that this propagator offers a spectacular improvement over a more naive *filtering* approach. Before concluding, we propose a list of interesting research topics, which could extend our work and have an important impact for the video game industry, providing novel challenges for constraint programming.

2 The *Dungeon Variations Problem*

2.1 Motivations

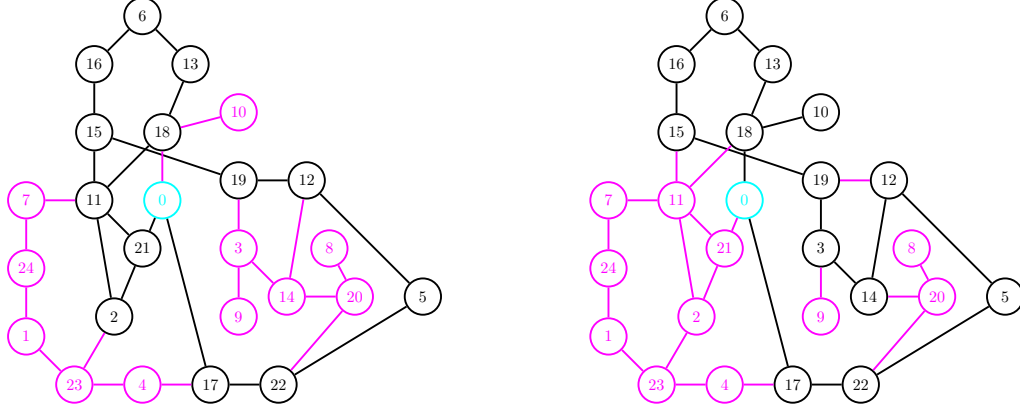
Our goal is to provide a game creation assistant tool for level designers (*LD*). This tool should provide *LD* an efficient way of building level variations to fill the open world at distinct places. This is a restricted view of the more general problem of procedural game generation, the aim of which is to guarantee high-quality automatic content generations. In these games, each level must be consistent with the narrative and the user's progression, which is not yet possible with fully autonomous level generation solutions. Delivering an AAA+ game with an unplayable level, or an inconsistent story can have dramatic effects in terms of images and costs for the responsible game studio. In order to reach the best possible quality, each level is handcrafted and validated by a *LD* guaranteeing a precise purpose of this level in the overall story. Our approach is thus not to generate levels from scratch but to propose variations (with guaranteed properties) of a dungeon pattern designed by an artist. All the variations have, by design, a strong story consistency (monsters, possible quests, ...).

We call *dungeon source* the original *design* (see for instance a very simple example of such a design Figure 1). This dungeon contains *rooms* with properties (in practice, the concept of *rooms* can be misleading: a room can be composed of a set of areas which will be considered as a whole block). A room can be an *entry* (a connection from the outside), an *exit* (a connection to the outside). Rooms can also have a set of *tags* to encode gameplay situations, design, or any other optional set up. The set of rooms that the user can explore and the situations (e.g. rewards, fights) encountered is called the *flow*. It is a common practice in level designing to add *final* rooms to dungeons to allow side quests or optional explorations. These final rooms, which are *dead-end* rooms (other than the hallway leading to them), usually contain treasures, keys, special items or monsters, but do not block the player. They are important for players because, in addition to optional rewards, *final* rooms offers a non-linear way of exploring the dungeon, which is a crucial aspect for the positive perception of levels by players. It should be noted that the *connections* between the rooms are directed (a door can be one-way access, the player can fall from one room to another, ...).

The goal of the *Dungeon Variations Problem* is to generate a coherent subset of rooms from the source dungeon satisfying some constraints (see for instance Figure 2). The first set of constraints is structural: each set of rooms must be playable (all rooms are connected, we have at least one entry and one exit), and the final rooms must be tagged as such (the role of the *LD* is typically to be able to check the interest of final rooms). The second set of constraints are *configuration* constraints, which are optional and can be defined by the user: some special rooms can be forced to be *active*, *entry*, *exit* or *final* in the solution. Our tool will have to offer, within a reasonable time (a few seconds), interesting variants to the *LD*, as illustrated, for example, Figure 2. The *LD* will set some tags defining the desired solutions to explore and will be able to navigate through a number of variations.

Procedural Generation of levels and flows is not new [6, 17, 18]. However, as pointed out in these references, systems are not yet mature to be autonomous and AAA+ studios are only using them in very restricted subareas of game creations. Moreover, none of these works has the potential to reach the level of quality expected in our context. To the best of our knowledge, no previous work is able to handle both the dungeon generation (structure of the dungeon) and the story itself (which is a cornerstone of our work, and will be ensured by the definition of the source dungeon). As described in the next section, we propose to tackle this problem by defining it as a graph problem. The connectivity property will be guaranteed by

a new extension of the graph constraint [5, 14] adapted to our problem. To illustrate this formalisation, Figures 3 and 4 show simplified versions of a graph-encoding of two generated variations used in production.



■ **Figure 3** Two generated variations of a source dungeon used in production for a AAA+ video game (for clarity sake, we did not represent oriented edges in this example). We can see the differences between the two variations on the use of final rooms and alternative paths. The blue node 0 is both the entry and the exit of this level. Nodes and edges in red are not kept in each variation.

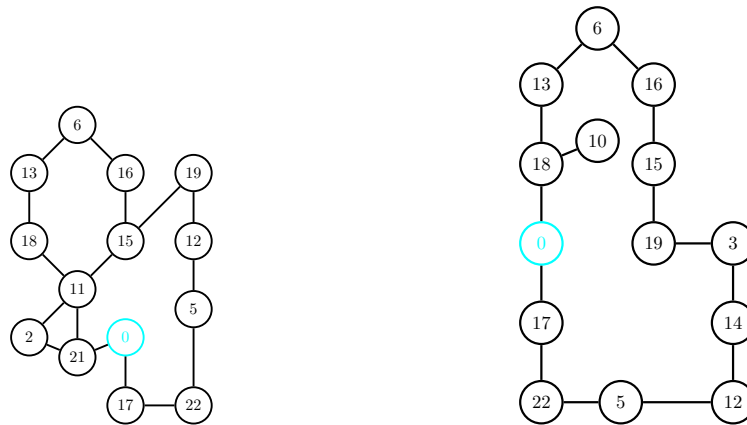
2.2 Related Works

Most work on constraints and graphs (e.g. [3, 4, 8, 11, 15]) generally consider one source node and one objective node with path/circuit problem. Unfortunately, we cannot rely directly on these works. They are either not applicable or too constrained: by definition, the path (or circuit) problem force the edges to be used only once, which is not relevant in our case. In addition, we have to consider the fact that the graphs are directed, and each variation can have a lot of source nodes (inputs) as well as several objective nodes (outputs). Of course, in our case, a node can be an entry and an exit at the same time¹, which is another argument to develop a novel approach. It is more classical, in state-of-the-art approaches, to suppose that entries and exit nodes are distinct (e.g., in path constraint [7]). Using the *reachable* constraint from every entry will unfortunately introduce a prohibitive overhead. It is also important to note, at this point, that we are not yet trying to optimise the paths in any way. We plan to do this as part of a further work by, for example, considering path optimisation of optimal objects placements [11]. It may also be important to optimise the flow in many additional ways, for example, by optimising a constraint-based formalisation of the fun as perceived by the player, or by optimising the hardware cost of rendering the scenes during the flow in order to guarantee a graphical performances.

3 CP model of the *Dungeon Variations Problem*

In this section, we formulate the problem as a constraint satisfaction problem. The constraints are presented in natural language at first, then in clausal and set form (Table 1).

¹ In practice, in real-world problems, this often happens.



■ **Figure 4** The two variations of Figure 3 with the remaining edges and nodes, reported for simplicity. The blue node 0 is both the entry and the exit of this level.

3.1 Model Variables

Let n be the number of nodes in the pattern (numbered from 0 to $n - 1$). The connections (e.g. *corridors*) between the nodes are identified by node numbers (e.g. C_{ij} is a connection from node i to node j). We use six arrays of size n to model the problem: Four arrays of *booleans* **entries**, **exits**, **actives** and **finals** which indicates, respectively, if a node is an entry, an exit, if it is active and if it is final. Two arrays **sumToNode** and **sumFromNode** which respectively counts the number of connections entering each node and leaving each node. To encode the connections between rooms, we use an $n \times n$ adjacency matrix C that can indicate the active edges (connections) of the graph since the nodes can have multiple successors. Arrays **sumToNode** and **sumFromNode** are defined using sum constraints. We also avoid multiple corridors between the same rooms. Such a tricky variation, when needed, can be handled at another level (e.g. on top of a variation that admits two connected rooms allowing multiple connections). To prevent trivial loops, our CP model must also block the diagonal of the adjacency matrix C .

Formally, the problem can be expressed as a graph problem in a fairly simple way: let $G(V, E)$ be a graph with $V \subset \mathbb{N}$ represents the rooms and $E \subset \{(i, j) \mid i, j \in V \wedge i \neq j\}$ the corridors. The goal of the *Dungeon Variations Problem* is to generate a graph $G'(V', E')$ with $V' \subset V$ and $E' \subset E$ with additional constraints encoding the fact that the variation is playable. Note that **actives** $\equiv V'$. The sets **entries**, **exits** and **finals** are subsets of V' .

3.2 Model Constraints

The constraints of the model ensure the consistency of each variation, given in natural language in the following list (see Table 1 for the clausal and set form). These constraints are expressed as intention constraints in the *XCSP3* model [1]:

- (1) If a node is an entry, it must be active.
- (2) If a node is an exit, it must be active.
- (3) If a node is final, it must be active.
- (4) If a node is an entry, it cannot be final.
- (5) If a node is an exit, it cannot be final.
- (6) If a connection is used, then the associated nodes must be active.
- (7) If a node is active, at least one connection must go to (or leave from) this node.

27:6 The Dungeon Variations Problem Using CP

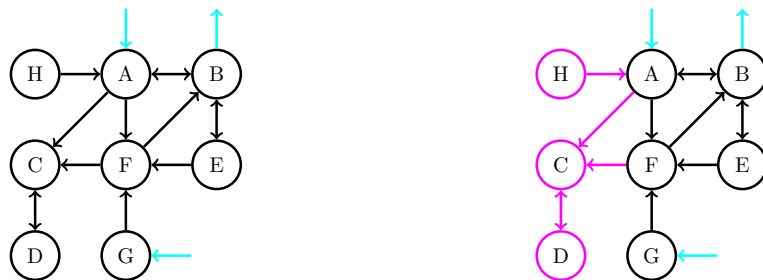
■ **Table 1** Constraints in clausal and set form. Note that the clausal form, lines 8-11, can be encoded as a single set constraint.

	Clausal form	Set form
1	$entries_i = 0 \vee actives_i = 1$	$i \in entries \Rightarrow i \in V'$
2	$exits_i = 0 \vee actives_i = 1$	$i \in exits \Rightarrow i \in V'$
3	$finals_i = 0 \vee actives_i = 1$	$i \in finals \Rightarrow i \in V'$
4	$entries_i = 0 \vee finals_i = 0$	$entries \cap finals = \emptyset$
5	$exits_i = 0 \vee finals_i = 0$	$exits \cap finals = \emptyset$
6	$C_{ij} = 0 \vee (actives_i = 1 \wedge actives_j = 1)$	$(i, j) \in E' \Rightarrow \{i, j\} \in V'$
7	$actives_i = 0 \vee sumFromNode_i > 0$ $\vee sumToNode_i > 0$	$i \in V' \Rightarrow \exists j \mid ((i, j) \in E' \vee (j, i) \in E')$
8	$finals_i = 0 \vee sumToNode_i = 1$	$i \in finals \Leftrightarrow \exists! j \mid (\{(i, j), (j, i)\} \subset E')$
9	$finals_i = 0 \vee sumFromNode_i = 1$	See 8.
10	$finals_i = 0 \vee C_{ij} = 0 \vee C_{ji} = 1$	See 8.
11	$finals_i = 0 \vee C_{ij} = 1 \vee C_{ji} = 0$	See 8.
12	$C_{ij} = 0 \vee C_{ji} = 0 \vee sumToNode_i \neq 1$ $\vee sumFromNode_i \neq 1 \vee finals_i = 1$	$\{(i, j), (j, i)\} \subset E' \wedge \forall k \mid \{(i, k)\} \in E' = 1$ $\wedge \forall k \mid \{(k, i)\} \in E' = 1 \Rightarrow i \in finals$

- (8) If a node is final, then one and only one connection must go to this node.
- (9) If a node is final, then one and only one connection must leave from this node.
- (10) If a node i is final and one connection goes from i to j then a must go from j to i .
- (11) If a node i is final and one connection goes from j to i then a must go from i to j .
- (12) If there is a round trip between two nodes i and j and there is only one connection to i and from i , then the node i must be final.

The above set of constraints allows to specify the structure of the desired variations (the problem of unconnected solutions will be discussed in the next section). Note that the management of *final* rooms is covered by lines 8 to 11. It ensures that the player can return to the main path from a dead-end, possibly corresponding to a secondary quest. This set of constraints (lines 8–11) is encoded as a single constraint when using the set-based form, Table 1.

Some other constraints, called *configuration* constraints, are also allowed in the proposed tool, but not reported here for simplicity (for instance, the tool allows the *LD* to limit the number of inputs, outputs, active and final rooms). It is also necessary that our tool allows the user to force specific parts to be active (or not) in all the generated variants (for instance, the *LD* may activate or deactivate certain parts of the source dungeon depending on where the variation will be placed on the map, for instance). The user can also deactivate a subset of corridors to explore the corresponding variations (for instance, some corridors may require the user to use some items/capacities that may not be available at all steps of the overall story). Likewise, it is possible to give control over situations (and therefore over the dungeon *flow*) by limiting the number of rooms with some given tags. For example, if we have 8 rooms marked as a combat room in a source dungeon, an *LD* may want to generate a variation with only 3 of them. To handle these limits easily, we can create a sum constraint on the *active* property of the tagged rooms. We do not further describe this part of the model since it is covered by a classical constraint-based approach and can be handled as soon as the overall problem is tackled by CP.



■ **Figure 5** The graph displayed on the left represents a source dungeon (or a variation) with two entries (nodes *A* and *G*) and one exit (node *B*). We can see that the node *H* is not accessible and will be filtered by the constraint introduced in Section 3.2. A *cul-de-sac* is also present in this dungeon (nodes *C* and *D*). Indeed, when entering node *C*, a player would be stuck inside the two rooms without any possibility to escape this “trap”. This kind of *cul-de-sac* is not filtered by the structural constraints and are not welcome in the resulting variations. The goal of the *connected+* constraint is to filter these cases and obtain the dungeon displayed on the right.

4 The Connected Constraint Extended

As mentioned before, we need to make sure that our variations are connected to satisfy all the constraints. We can ensure this in two different ways. (1) We may verify each solution, afterwards, with an *ad hoc* algorithm (this is a classic approach in video games: the levels generated can be tested subsequently by algorithms, bots and/or humans) or, (2) we may force any proposed variation to be correct by construction, filtering out partial solutions as soon as possible when looking for variations. Let us recall here that the goal of our approach is to help the *LD* navigating through the possible alternatives as easily as possible. Being able to guarantee that each proposed solution has certain properties by construction is, therefore, a crucial element. Figure 5 shows a very simple example of the *connected+* constraint we propose, in practice.

The approach we propose is a two-level propagator that first relies on a simple implementation of the constraint **connected** [5] on the undirected version of the graph (Algorithm 1). Then, another level of filtering is added (Algorithm 4 and 3) to guarantee the validity of the paths. The **connected** constraint guarantees that we do not have disjoint dungeons but does not consider directed graphs. We, therefore, have no guarantee that the dungeon is playable (i.e. all nodes are accessible from the entrances, in particular, the exit nodes). To overcome this problem, we introduce an algorithm to find paths from each entry to at least one exit (this algorithm can be extended in many ways, for instance to handle the size of the paths, whether they are constrained or not, or to obtain information (average length, etc.)). For this reason, the propagator need to know the values of the **entries** and **exits** arrays.

Algorithm 1 starts from an active node (line 2, the `selectActiveNode()` function selects an active node in the graph, i.e. a node x such that $active_s_x = 1$, or returns an empty set if no active node is found). Then, it follows each connection by considering the undirected version of the graph (line 12). From line 3 to 7, if we do not find any active node, we need to check the consistency of the graph. Indeed, if some of the nodes are not yet decided (i.e. their Boolean domain size is 2), the constraint is consistent since we cannot filter anything. Otherwise, an inconsistency is detected if all the nodes are marked inactive (lines 4–5). Until the stack is empty, we mark the current node (line 11) and add its neighbours to the stack (line 12). When the stack is empty, Algorithm 2 is called to filter the nodes and to check

■ **Algorithm 1** `connected()`: Boolean.

```

1 seen  $\leftarrow \{\emptyset\}$ ;
2 stack  $\leftarrow \text{selectActiveNode}()$ ;
3 if stack =  $\emptyset$  then
4   if  $\{node \text{ s.t. } |dom(activess_{node})| = 2\} = \emptyset$  then
5     return false; // conflict detected
6   else
7     return true;

8 while stack  $\neq \emptyset$  do
9   current  $\leftarrow pop(stack)$ ;
10  if current  $\notin seen$  then
11    seen  $\leftarrow seen \cup current$ ;
12    stack  $\leftarrow stack \cup \{x \text{ s.t. } activess_x \neq 0 \wedge (C_{current,x} \neq 0 \vee C_{x,current} \neq 0)\}$ ;

13 return removeUnseen(seen);

```

■ **Algorithm 2** `removeUnseen(seen : Set of nodes)` : Boolean.

```

1 foreach node  $\notin seen$  do
2   if  $|dom(activess_{node})| = 1 \wedge activess_{node} = 1$  then
3     return false; // conflict detected
4   activess_{node}  $\leftarrow 0$ ;
5 return true;

```

the consistency of the constraints. In fact, for each node that we did not see in the search of Algorithm 1 (therefore not connected to the considered graph), we must set their active value to 0. A conflict is identified when an already decided active node is processed.

After running Algorithm 2, the undirected graph is guaranteed to be connected. Now, we need to check the directed paths from each entry node and identify the unreachable nodes to disable them. We formally introduce Algorithms 4 and 3 to manage the directed graph.

Algorithm 3 checks that there is a path between each entry and at least one exit node. Three sets are used: **seen**, **possible** and **kept**. These sets contain, respectively, the nodes that have been seen, the nodes in a possibly valid path (i.e. which reach an exit) from the considered entry, as well as the nodes which are kept if no conflict is detected before the end of the algorithm. The graph is then traversed from each potential entry using Algorithm 4 (line 6). If the considered entry is valid, it is counted (line 7); otherwise, it is marked as invalid following a consistency test (line 9 to 11). If no valid entry is found (lines 13 and 14), a conflict is raised. We then call Algorithm 2 (line 15) to filter the nodes that have not been kept (and possibly detect a conflict).

Algorithm 4 (called on line 6 of Algorithm 3) is used to test the validity of a node provided as a parameter (**node**). First, the node to be tested is added to the **seen** and **possible** sets (lines 2 and 3). Then, if the node is already in the set of nodes to keep **kept** or if this node is a possible exit, then it is marked as valid and kept (lines 4 to 6). It should be noted that the search is not stopped when a node (or more particularly an entry) is found valid since we want to mark all the nodes reachable from the node which is currently considered. The main

■ **Algorithm 3** `checkPaths(G : the graph) : Boolean.`

```

1  $seen \leftarrow \{\emptyset\};$ 
2  $possible \leftarrow \{\emptyset\};$ 
3  $kept \leftarrow \{\emptyset\};$ 
4  $nbValidEntries \leftarrow 0;$ 
5 foreach  $node$  s.t.  $entries_{node} \neq 0 \wedge actives_{node} \neq 0$  do
6   if isValidNode( $node, seen, possible, kept$ ) then
7      $nbValidEntries \leftarrow nbValidEntries + 1$ 
8   else
9     if  $entries_{node} = 1$  then
10      return false; // conflict detected
11      $entries_{node} \leftarrow 0$ 
12    $possible \leftarrow \{\emptyset\};$ 
13 if  $nbValidEntries = 0$  then
14   return false; // conflict detected
15 return removeUnseen( $kept$ );

```

■ **Algorithm 4** `isValidNode($node$: a node, $seen, possible, kept$: node sets) : Boolean.`

```

1  $valid \leftarrow false;$ 
2  $seen \leftarrow seen \cup \{node\};$ 
3  $possible \leftarrow possible \cup \{node\};$ 
4 if  $node \in kept \vee exits_{node} \neq 0$  then
5    $valid \leftarrow true;$ 
6    $kept \leftarrow kept \cup possible;$ 
7 foreach  $a$  s.t.  $actives_{node} \neq 0 \wedge C_{node,a} \neq 0$  do
8   if  $a \in kept$  then
9      $valid \leftarrow true;$ 
10     $kept \leftarrow kept \cup possible;$ 
11   else if  $a \notin seen$  then
12     if isValidNode( $a, seen, possible, kept$ ) then
13        $valid \leftarrow true;$ 
14  $possible \leftarrow possible \setminus \{node\};$ 
15 return  $valid;$ 

```

loop (lines 7 to 13) allows considering the children of the node considered (**node**). Indeed, like line 12 of Algorithm 1, we add the neighbours of the current node without considering the undirected case this time. For each of the children **a**, that are not deactivated, two cases are possible:

1. **a** is already in the set of kept nodes (**kept**) (lines 8 to 10). This implies that **node** is valid because it joins a path already validated previously. The path of the possible nodes is therefore kept (line 10).

■ **Algorithm 5** `propagate()` : Boolean.

```
1 return connected() && checkPaths();
```

2. `a` has never been explored (lines 11 to 13). Algorithm 4 is therefore called recursively to test the validity of `a` (as well as that of `node` if `a` is valid ²).

Finally, before returning the validity of the original node (`node`), it is removed from the `possible` set (line 14).

The `connected+` propagator (Algorithm 5) first applies the `connected` algorithm (Algorithm 1), and if no conflict occurs, then the paths are checked (Algorithm 3).

5 Experiments

We implemented the algorithms in the *Nacre* [9] solver to evaluate them. The *XCSP3* team kindly provided us with an alternative version of the official parser to manage our new global constraint.

The experiments were performed on a computer with a 6-core processor clocked at 3.70 GHz (Intel i7-8700K) with 32GB of RAM. We used the complete search method (MAC) of the *Nacre* solver without restarts to count the solutions found³.

The experiments are divided into three parts. The first one presents a real-world use case called *the 1,000 dungeons*. It is a simple scenario that illustrate how our tool can be used in practice, on a simple use case. This “finalised” proof of concept was used to demonstrate the practical interest of our approach to the AAA+ game studio we collaborated with for this study. Indeed, generating a large number of variations from a few source patterns lets us quickly fill the world with many high-quality playgrounds. The second part shows the performances of our approach against the previously used one (the *a posteriori* method used in the AAA+ game studio) on a real-world source dungeon. Our procedure generates only valid variations for an insignificant overhead compared to the one without the global constraint. In the last part, we present a source dungeon generator, based on a structured problem random generator, to evaluate our approach to more dungeons. This generator allows us to conclude on the usefulness and performances of our procedure in different cases.

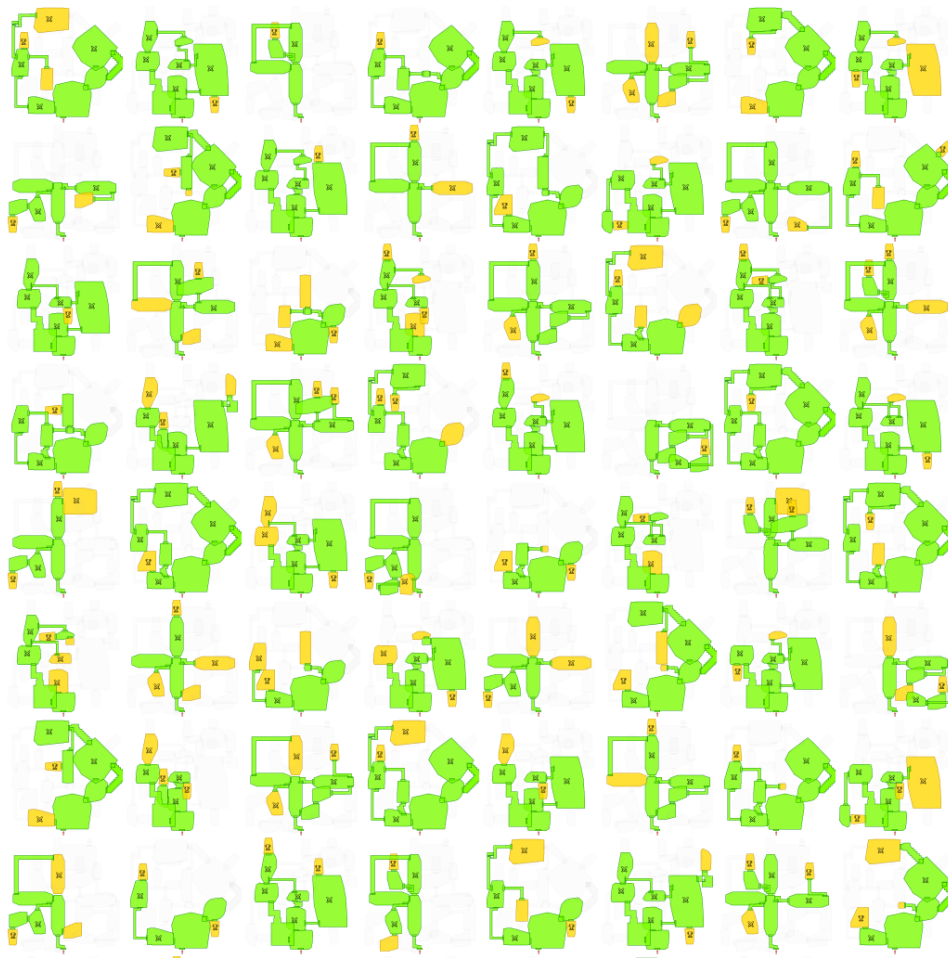
5.1 The 1,000 Dungeons Experiment

The *1,000 dungeons* experiment is a proof of concept developed in the AAA+ video game studio with which we collaborated to show the practical interest of our approach. It corresponds to a finalised tool, build on top of many other internal tools. In this experiment, we generated, from three source patterns, 1,000 variations. The structural constraints of the model are the ones presented in Section 3.2. The configuration constraints for the computed variations are set up as follows: between 3 and 12 rooms; at most 3 final rooms; one treasure in a final room; and between 3 and 8 fights.

Figure 6 shows a sneak peek of the generated dungeons. We can distinguish the different source patterns used as input. The green and golden rooms are kept in the variation; the golden ones are the final rooms. The crossed swords point out a fight room, and the cup shows a treasure room.

² It is not necessary to replicate line 10 after line 13 if `a` and `node` are valid, this has already been done in the call at line 12.

³ `./nacre_mini_release DUNGEON.xml -complete -sols=X`



■ **Figure 6** A sneak peek of the 1,000 dungeons experiment using 3 source dungeons that shows the integration of our CP model and propagator in a production tool for a AAA+ video game.

This proof of concept shows that our approach could be useful to quickly fill a world with a lot of different variations or to train bots (e.g., for exploration, for fights). It also illustrates how variants of the same source dungeon could be very different in practice. Most of the considered dungeons Figure 6 will be perceived as different levels by the player. To increase the player’s feeling of visiting different levels, for this experiment, we adapted our tool to avoid close variations. We generated much more variations and selected distant ones in the search tree (choosing one solution every fifty solutions, during the backtrack search, allowed us to increase the chances of increasing the diversity, as shown on the random selection of some computed variations, Figure 6). To further expand this work and ensure the generation of a relevant set of variations from one (or more) source dungeon in these kinds of experiment, one could use some metrics (mission linearity, map linearity, leniency and path redundancy [12, 16, 17], for example) to score each variation. The use of additional tags (for instance by specifying the textures of the walls, the positions of furniture and items) are another way of enforcing the novelty feeling by the player. Improving our tool to take this kind of new constraints is easy with CP and will be discussed in the last section.

■ **Table 2** Data from experiments on the real-world instance considered, with 3,019 variables and 11,169 when expressed as a XCSP3 instance. The last column shows how many variations are valid over the number of variations produced by the given method. As discussed in the text, only a small fraction of generated variations are valid when the *connected+* propagator is not used.

#Variations	Method	Time (seconds)	#Conflicts	#Valid Variations
100	W/o GC	2.7	15	0
	Connected+	2.78	173	100
1,000	W/o GC	4.33	130	0
	Connected+	5.42	1,462	1,000
10,000	W/o GC	21.72	820	248
	Connected+	32.23	11,781	10,000

5.2 Study of a real-world Instance

To evaluate our approach against the standard *a posteriori* method, we used a real-world source dungeon taken from an actual game level. The source dungeon used for this experiment has 52 nodes, 58 connections, 7 possible entrances, 7 possible exits, and 30 possible final rooms. Even though this problem seems small, it is already difficult and represents real problems well (mainly between 40 and 60 nodes, up to about 100 nodes for larger ones). The XCSP3 instance associated to this problem has 3,019 variables and 11,169 constraints (1 more for the graph constraint *connected+*).

We evaluate the two approaches on generating 100, 1,000 and 10,000 variations, respectively. The number of variants seems large (a typical shipped game will never contain so many variations) but the goal of our tool is to give the *LD* as many variations as possible for them to pick up the most interesting ones. It is thus expected that our tool will have to handle such large sets of variations. We may have to score each variation (according to the metrics mentioned, for instance, in section 5.1), and to list them for the *LD*.

The first approach (*W/o GC* in Table 2) does not use the new global constraint. The second one (*Connected+* in Table 2) uses the new global constraint and computes exactly the number of variations specified (since they are all valid). The search stops when the specified number of variations is found (but not necessarily valid ones for the *W/o GC* approach).

In Table 2, we can check that, as expected, having the graph checker as a built-in propagator only produces valid variations for the price of small overhead. We conducted further experiments to measure how many variations the method *W/o GC* would have to compute in order to obtain 100 valid variations. We found that 5,390 solutions would have to be computed (in 7.63 seconds with 542 conflicts, to be compared with the 2.78 seconds of our approach, with only 173 conflicts). We also tried to compute 1,000 valid variations with the *W/o GC* method but, after hitting the 2,400 seconds timeout, it generated less than 300 valid variations for this instance. This has to be compared with our approach, that allows to generate 1,000 valid variations in 5.42 seconds.

This clearly shows the interest of our global constraint. It is now possible to produce a tool allowing a real time exploration of many variants, to rank them and change the desired tags almost on the fly. This was clearly not even possible with the previous approach.

5.3 Small-World Random Generated Instances

In order to generalise our findings, we propose a deeper experimental analyse thanks to a random instance generator (<https://bit.ly/3xSxK2B>), available for further works to compare with (our previous problems are unfortunately not publicly available). To kept the random generated graphs interesting and close to real-world problems, designed by humans, our generator is based on the *Watts–Strogatz* model [10] adapted for directed graphs [13]. This model produces graphs with small-world properties; it allows us to simulate real-world instances (e.g., with local clustering) and not random graph with uniform structure only. The model takes the number of nodes of the graph we want to generate N , the mean degree K of the graph and some special probability parameters (detailed below, these are reals between 0 and 1). The associated algorithm is composed of two parts:

1. construct a ring lattice (a graph of N nodes each connected to K neighbours) ;
2. rewire some edges following a probability p .

As mentioned before, in our case, the algorithm is modified to handle directed graphs (Section 3.2 of [13]). The first step remains the same, but the connections are set both ways. The second step uses another probability d to choose if only one way is rewired or both. We also randomly select nodes to be entries and exits. Our generator takes a few parameters:

1. n , the number of nodes of the graph;
2. k , the number of edges by nodes for the ring lattice;
3. a , the access density (percentage of entries and exits);
4. d , the probability for reciprocal edges during rewiring steps;
5. p , the probability of edge rewiring;
6. s , a seed can be specified for reproducibility.

This generator builds the structure of a source dungeon and does not propose to (randomly) generate the configuration constraints. Indeed, choosing and generating a realistic configuration is another work by itself.

We generated 100 instances⁴ from 10 nodes to 100 nodes with a step of 10 (10 instances for each step). The generation tool is called with the following default parameters: 20% accesses density (a parameter; 20% of nodes are entries and 20% of nodes are exits); 20% chance for reciprocal edges (d parameter); 20% chance for rewiring (p parameter); and finally, k , the number of edges by nodes for the ring lattice, is 40% of n , the number of nodes of the graph.

The results of the two methods (*W/o GC* and *Connected+*) on the 100 generated instances are reported Table 3. As we have done in the previous experiment, we evaluate the two methods on the generation of 100, 1,000 and 10,000 variations. The data presented in the Table are the computation time (in seconds) and the number of valid variations generated. For each of the metrics, the minimum, maximum, median and mean are shown. This allows us to show the behaviour on the easier (usually fewer nodes in the graph) and harder instances.

This last experiment, based on 300 runs for each method (each run generating many variations, with a total of more than 2 millions variations generated), confirms the conclusions on the previous real-world instance:

- the overhead due to the addition of our global constraint *connected+* constraint is very small;

⁴ We used a script with the following parameters to generate the instances presented in Table 3:
`./genDungeons.sh 10 100 10 10 0.`

■ **Table 3** Data from experiments on the Watts-Stragatz small-world random instances. Each reported number summarises 100 points, 10 points per increasing size of 10 to 100 (see text).

#Var	Method	Time (sec)				#Valid Variations			
		Min	Max	Med.	Mean	Min	Max	Med.	Mean
100	W/o GC	0.13	23.42	6.28	7.57	0	46	24	22.7
	Connected+	0.14	25.94	6.3	7.62	100	100	100	100
1,000	W/o GC	0.27	27.48	7.7	9.07	0	380	173	166.74
	Connected+	0.39	29.15	7.91	9.37	1K	1K	1K	1K
10,000	W/o GC	1.89	57.58	20.39	22.7	0	4798	1,244	1,578.45
	Connected+	2.56	66.45	24.31	26.68	10K	10K	10K	10K

- the variation problem, and the use of classical graph constraint, generates a very large number of non valid solutions;
- the median time obtained by our approach makes it possible to use it in an on-line manner, to help *LD* navigating and sorting the solutions. This is not possible with an approach based on existing constraints.

6 Conclusion & Perspectives

We have presented a new problem of industrial importance for constraint programming, the *Dungeon Variations Problem*, and proposed a first approach to tackle it, based on the introduction of a new global constraint with its propagator. Our solution is already used in pre-production as an internal support tool for *Levels Designers*, supported by the *XCSP3* model and the *Nacre* solver. There is, of course, still rooms for improvements, but we believe our approach has already proven its usefulness and its practical interest for the studio. It is a pragmatic and efficient solution to help *Levels Designers* in their daily work for the gaming industry.

We are, of course, planning to expand this work in several ways. We can use metrics (mission linearity, map linearity, leniency and path redundancy [12, 16, 17], for example) to score each variation to generate a relevant set of variations from a source dungeon. These metrics could be used for the optimisation version (COP) of the *Dungeon Variations Problem*, allowing more control over the dungeons generated (an *LD* often plays on the linearity of the *flow*). Using an approach similar to [2], we could find paths where the dungeon structure is made using some data (or approximations): time or hardness to complete a fight or a puzzle, for example. We could then build levels based on difficulty or time.

We could also enrich the model with different constraints to provide more control and automation for the *Dungeon Variations Problem* (for example, considering the orientation of the room, allowing its rotation). We can also think of models of rooms and connections (for example, a connection can be a hallway, a window, a breakable wall). We can also extend graph constraints to handle distance constraints (e.g., between accesses, from entrances to combat rooms). A final improvement would be to consider qualitative constraint networks (*QCN*) to manage the relative positions of the different rooms and connections, allowing an *LD* to specify the topological constraints of the connections from the dungeon to the outside.

A long-term goal of our work is to generate levels on the fly, based on each user experience and preferences, with strong guarantees. We believe that this work is the first step in this direction.

References

- 1 Gilles Audemard, Frédéric Boussemart, Christophe Lecoutre, Cédric Piette, and Olivier Roussel. Xcsp³ and its ecosystem. *Constraints An Int. J.*, 25(1-2):47–69, 2020. doi:10.1007/s10601-019-09307-9.
- 2 Daniel Le Berre, Pierre Marquis, and Stéphanie Roussel. Planning personalised museum visits. In Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini, editors, *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*. AAAI, 2013. URL: <http://www.aaai.org/ocs/index.php/ICAPS/ICAPS13/paper/view/6025>.
- 3 Diego de Uña. *Discrete optimization over graph problems*. PhD thesis, University of Melbourne, Parkville, Victoria, Australia, 2018. URL: <http://hdl.handle.net/11343/217321>.
- 4 Diego de Uña, Graeme Gange, Peter Schachte, and Peter J. Stuckey. A bounded path propagator on directed graphs. 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 189–206. Springer, 2016. doi:10.1007/978-3-319-44953-1_13.
- 5 Grégoire Doms, Yves Deville, and Pierre Dupont. Cp(graph): Introducing a graph computation domain in constraint programming. 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 211–225. Springer, 2005. doi:10.1007/11564751_18.
- 6 Joris Dormans. A handcrafted feel: Unexplored explores cyclic dungeon generation. URL: <https://ctrl500.com/tech/handcrafted-feel-dungeon-generation-unexplored-explores-cyclic-dungeon-generation/>.
- 7 Jean-Guillaume Fages. *Exploitation de structures de graphe en programmation par contraintes*. Theses, Ecole des Mines de Nantes, 2014. URL: <https://tel.archives-ouvertes.fr/tel-01085253>.
- 8 Jean-Guillaume Fages. On the use of graphs within constraint-programming. *Constraints An Int. J.*, 20(4):498–499, 2015. doi:10.1007/s10601-015-9223-9.
- 9 Gael Glorian, Jean-Marie Lagniez, and Christophe Lecoutre. NACRE - A nogood and clause reasoning engine. In Elvira Albert and Laura Kovács, editors, *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 of *EPiC Series in Computing*, pages 249–259. EasyChair, 2020. URL: <https://easychair.org/publications/paper/rN67>.
- 10 Carsten Grabow, Stefan Grosskinsky, Jürgen Kurths, and Marc Timme. Collective relaxation dynamics of small-world networks. *CoRR*, abs/1507.04624, 2015. arXiv:1507.04624.
- 11 Ian Horswill and Leif Foged. Fast procedural level population with playability constraints. In *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE'12*, page 20–25. AAAI Press, 2012.
- 12 R. Lavender. The zelda dungeon generator: Adopting generative grammars to create levels for action-adventure games, 2016.
- 13 Brenton Pettejohn, Matthew Berryman, and Mark McDonnell. Methods for generating complex networks with selected structural properties for simulations: A review and tutorial for neuroscientists. *Frontiers in Computational Neuroscience*, 5:11, 2011. doi:10.3389/fncom.2011.00011.
- 14 Patrick Prosser and Chris Unsworth. A connectivity constraint using bridges. 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 707–708. IOS Press, 2006.

- 15 Luis Quesada, Peter Van Roy, Yves Deville, and Raphaël Collet. Using dominators for solving constrained path problems. In Pascal Van Hentenryck, editor, *Practical Aspects of Declarative Languages, 8th International Symposium, PADL 2006, Charleston, SC, USA, January 9-10, 2006, Proceedings*, volume 3819 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 2006. doi:10.1007/11603023_6.
- 16 Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games, PCGames '10*, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1814256.1814260.
- 17 Thomas Smith, Julian A. Padget, and Andrew Vidler. Graph-based generation of action-adventure dungeon levels using answer set programming. In Steve Dahlsgog, Sebastian Deterding, José M. Font, Mitu Khandaker, Carl Magnus Olsson, Sebastian Risi, and Christoph Salge, editors, *Proceedings of the 13th International Conference on the Foundations of Digital Games, FDG 2018, Malmö, Sweden, August 07-10, 2018*, pages 52:1–52:10. ACM, 2018. doi:10.1145/3235765.3235817.
- 18 Valtchan Valtchanov and Joseph Alexander Brown. Evolving dungeon crawler levels with relative placement. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering, C3S2E '12*, page 27–35, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2347583.2347587.
- 19 Breno M. F. Viana and Selan R. dos Santos. A survey of procedural dungeon generation. In *18th Brazilian Symposium on Computer Games and Digital Entertainment, SBGames 2019, Rio de Janeiro, Brazil, October 28-31, 2019*, pages 29–38. IEEE, 2019. doi:10.1109/SBGames.2019.00015.

Refined Core Relaxation for Core-Guided MaxSAT Solving

Hannes Ihalainen ✉

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

Jeremias Berg ✉ 

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

Matti Järvisalo ✉ 

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

Abstract

Maximum satisfiability (MaxSAT) is a viable approach to solving NP-hard optimization problems. In the realm of core-guided MaxSAT solving – one of the most effective MaxSAT solving paradigms today – algorithmic variants employing so-called soft cardinality constraints have proven very effective. In this work, we propose to combine weight-aware core extraction (WCE) – a recently proposed approach that enables relaxing multiple cores instead of a single one during iterations of core-guided search – with a novel form of structure sharing in the cardinality-based core relaxation steps performed in core-guided MaxSAT solvers. In particular, the proposed form of structure sharing is enabled by WCE, which has so far not been widely integrated to MaxSAT solvers, and allows for introducing fewer variables and clauses during the MaxSAT solving process. Our results show that the proposed techniques allow for avoiding potential overheads in the context of soft cardinality constraint based core-guided MaxSAT solving both in theory and in practice. In particular, the combination of WCE and structure sharing improves the runtime performance of a state-of-the-art core-guided MaxSAT solver implementing the central OLL algorithm.

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

Keywords and phrases maximum satisfiability, MaxSAT, core-guided MaxSAT solving

Digital Object Identifier 10.4230/LIPIcs.CP.2021.28

Supplementary Material Source code and experiment data are available at <https://bitbucket.org/coreo-group/cgss/>.

Funding Work financially supported by Academy of Finland under grants 322869 and 342145.

1 Introduction

Maximum satisfiability (MaxSAT) [8, 19] has in recent years developed into a noteworthy declarative Boolean optimization paradigm, with successful applications in various NP-hard industrial problem domains and artificial intelligence applications (see e.g. [8] and references therein).

So-called core-guided MaxSAT algorithms form one of the most central MaxSAT solving paradigms today [15, 26, 25, 28, 12, 18, 5, 27, 4]. The core-guided approach consists of iteratively extracting unsatisfiable cores, i.e., inconsistent subsets of soft constraints, using a Boolean satisfiability (SAT) solver, and at each iteration transforming the MaxSAT instance to include knowledge of the new unsatisfiable cores. This transformation involves compiling the cores into the instance via additional cardinality constraints. A key aspect in which the various core-guided algorithms differ is how exactly the transformations are performed. The use of so-called soft cardinality constraints [1, 28, 26] has recently proven to be a particularly effective approach to core-guided MaxSAT solving [7, 6].



© Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 28; pp. 28:1–28:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this work, we focus on improving and understanding core-guided MaxSAT algorithms through fine-grained changes to the way the transformations at each iteration are performed and realized. Here we focus in particular on realizing the proposed improvements and understanding their effects in the context of OLL [26, 1], which gives one of the currently most successful MaxSAT solving approaches through its implementation in the RC2 MaxSAT solver [18], and arguably represents the current state of the art in complete core-guided MaxSAT solving. That said, the techniques proposed in this work are also applicable in the context of other current and foreseeable variants of the core-guided approach using soft cardinality constraints; we will more shortly provide evidence on this with an implementation of the PMRES [28] core-guided algorithm, in addition to OLL.

Our main contributions are centered around and build on the recently proposed approach of weight-aware core extraction (WCE) [11]. In short, WCE enables relaxing multiple cores instead of a single one during iterations of core-guided search by exploiting soft clause weights in weighted MaxSAT instances, and can be viewed as an extension of work on computing lower bounds for MaxSAT [16, 20].

As our main contribution, we propose a novel form of structure sharing in the cardinality-based core transformation steps performed in core-guided MaxSAT solvers. The proposed form of structure sharing is enabled by WCE. While WCE has so far not been widely integrated to MaxSAT solvers, its combination with the here proposed structure sharing approach provides performance improvements to RC2, as we will empirically show. Syntactically, the structure sharing approach allows for introducing fewer variables and clauses during the MaxSAT solving process, which alleviates to an extent issues resulting from iteratively performing core transformation steps which in turn results in the working formulas of the algorithm increasing in size beyond the capabilities of modern SAT solvers. Here we note that, while so-called incremental cardinality constraints have been proposed and are applied in core-guided MaxSAT solvers [25, 24, 23] with the goal of keeping the working formulas smaller by more carefully introducing necessary parts of cardinality constraint encodings, the structure sharing approach we propose is a conceptually different technique. Structure sharing focuses on sharing substructures of *multiple* cores extracted via WCE during a *single* iteration of core-guided search. Furthermore, as we will show, shared substructures obtained via structure sharing allow for a more careful introduction of equivalence in the core transformations steps.

By a careful implementation on top of the state-of-the-art RC2 MaxSAT solver, we demonstrate that structure sharing and refined equivalences, combined with WCE, improves the runtime performance of RC2 on standard weighted MaxSAT benchmarks from MaxSAT Evaluations. Complementing the main practical contributions, we also provide a theoretical analysis of on the impact of integrating WCE into the OLL algorithm, more specifically on the effect that WCE has on the number of core extractions needed for termination and the sizes of cores extracted (as well as its potential drawbacks).

2 Maximum Satisfiability

A literal l is a Boolean variable x or its negation $\neg x$, the negation of a variable satisfies $\neg\neg x = x$. A clause C is disjunction (or set) of literals and a CNF formula F is a conjunction (or set) of clauses. A truth assignment τ maps Boolean variables to 0 (false) and 1 (true). τ is extended to literals l , clauses C and formulas F , respectively, by $\tau(\neg l) = 1 - \tau(l)$, $\tau(C) = \max\{\tau(l) \mid l \in C\}$ and $\tau(F) = \min\{\tau(C) \mid C \in F\}$. An assignment τ is a model of F if $\tau(F) = 1$. A formula is satisfiable if it has model, and otherwise unsatisfiable.

A MaxSAT instance \mathcal{F} consists of two CNF formulas, the set of hard clauses $\text{HARD}(\mathcal{F})$ and the set of soft clauses $\text{SOFT}(\mathcal{F})$, and a weight function $w: \text{SOFT}(\mathcal{F}) \rightarrow \mathbb{N}$ that assigns a positive weight to each soft clause. An assignment τ is a solution to \mathcal{F} if it satisfies the hard clauses. The cost $\text{COST}(\mathcal{F}, \tau)$ of τ is the sum of the weights of the soft clauses it falsifies, i.e. $\text{COST}(\mathcal{F}, \tau) = \sum_{C \in \text{SOFT}(\mathcal{F})} (1 - \tau(C))w(C)$. A solution τ is optimal if $\text{COST}(\mathcal{F}, \tau) \leq \text{COST}(\mathcal{F}, \tau')$ for all solutions τ' to \mathcal{F} . The cost $\text{COST}(\mathcal{F})$ of an instance \mathcal{F} is the cost of its optimal solutions. In the rest of this paper, we assume all MaxSAT instances have solutions, i.e. that $\text{HARD}(\mathcal{F})$ is satisfiable. In practice, the implementations of core-guided MaxSAT algorithms that we are aware of check this assumption by invoking a SAT solver on the hard clauses prior to search.

To simplify the discussion we will assume that every $C_i \in \text{SOFT}(\mathcal{F})$ is of form $(\neg b_i)$ for a variable b_i . The assumption can be made without loss of generality since one can always transform any soft clause $C \in \text{SOFT}(\mathcal{F})$ into the hard clause $C \vee b$ and the soft clause $(\neg b)$ with $w((\neg b)) = w(C)$. A variable b for which $(\neg b) \in \text{SOFT}(\mathcal{F})$ is a *blocking variable*. Let $\mathcal{B}(\mathcal{F})$ be the set of all blocking variables of \mathcal{F} . As setting $b = 1$ falsifies the soft clause $(\neg b)$, we can refer to soft clauses and blocking variables interchangeably, and extend the weight function w to blocking variables by $w(b) = w((\neg b))$. The cost of a solution τ is then $\text{COST}(\mathcal{F}, \tau) = \sum_{b \in \mathcal{B}(\mathcal{F})} \tau(b)w(b)$. For a set $B \subset \mathcal{B}(\mathcal{F})$ we let $\text{MINW}(B) = \min\{w(b) \mid b \in B\}$ be the smallest weight of the variables in B .

A central concept in modern MaxSAT solving is that of a (an unsatisfiable) core. Given a MaxSAT instance \mathcal{F} , a set $\kappa \subseteq \text{SOFT}(\mathcal{F})$ is a core if the formula $\text{HARD}(\mathcal{F}) \wedge \kappa$ is unsatisfiable. As each soft clause is a unit clause containing the negation of a blocking variable, this implies that any solution to \mathcal{F} sets $b = 1$ for at least one $b \in \mathcal{B}(\mathcal{F})$ for which $(\neg b) \in \kappa$. Hence we often view a core as a set of blocking variables (or a clause) $\{b \mid (\neg b) \in \kappa\}$ that is entailed by $\text{HARD}(\mathcal{F})$. A core κ is minimal (an MUS) if $\text{HARD}(\mathcal{F}) \wedge \kappa_s$ is satisfiable for all $\kappa_s \subset \kappa$.

► **Example 1.** Let n and r be two integers with $0 < r < n$, and $\mathcal{F}^{n,r}$ a MaxSAT instance such that $\text{HARD}(\mathcal{F}^{n,r})$ contains clauses equivalent to $\sum_{i=1}^n b_i \geq r$ and $\mathcal{B}(\mathcal{F}^{n,r}) = \{b_1, \dots, b_n\}$. In words, the clauses of $\mathcal{F}^{n,r}$ enforce that at least r soft clauses should be falsified (recall that assigning a blocking variable b to 1 corresponds to falsifying a soft clause). Assuming $w(b_i) = 1$ for all $i = 1 \dots n$, any solution τ that sets exactly r variables in $\mathcal{B}(\mathcal{F}^{n,r})$ to 1 and the rest to 0 is an optimal solution to $\mathcal{F}^{n,r}$ and has $\text{COST}(\mathcal{F}^{n,r}, \tau) = \text{COST}(\mathcal{F}^{n,r}) = r$. Any $\kappa \subset \mathcal{B}(\mathcal{F}^{n,r})$ that contains at least $n - r + 1$ variables is a core. In order to see this note that since r blocking variables need to be set to 1 – or equivalently r soft clauses need to be falsified – by any solution, it follows that any subset of blocking variables with at least $(n - r + 1)$ variables has to contain at least 1 that is set to 1 by any solution. The MUSes of $\mathcal{F}^{n,r}$ are the subsets of $\mathcal{B}(\mathcal{F}^{n,r})$ that contain exactly $n - r + 1$ variables.

Unit propagation is the main form of constraint propagation applied in CDCL SAT solvers. Consider a clause $C = \{l_1, \dots, l_n\}$ and assume the value of l_i has been fixed to 0 for all $i = 1, \dots, n - 1$. Then in order for C to be satisfied, the value of the literal l_n needs to be fixed to 1. We say that $l_n = 1$ is (unit) propagated by the clause C and the assignments $l_i = 0$ for $i = 1, \dots, n - 1$. CDCL SAT solvers perform unit propagation whenever the current partial assignment sets all but one literal from a clause to false (0).

3 MaxSAT by Soft Cardinality Constraints

Our main focus is on the OLL algorithm, which is one of the most successful core-guided MaxSAT solving approaches using so-called soft cardinality constraints. Algorithm 1 represents a generic abstraction of the core-guided approach using soft cardinality constraints to

■ **Algorithm 1** CG, a generic view on core-guided MaxSAT solving with soft cardinality constraints.

Input: A MaxSAT instance \mathcal{F}
Output: An optimal solution τ to \mathcal{F}

```

1 begin
2    $\mathcal{F}^1 \leftarrow \mathcal{F}$ 
3   for  $i = 1, \dots$  do
4      $(res, \kappa, \tau) \leftarrow \text{EXTRACT-CORE}(\mathcal{F}^i, \mathcal{B}(\mathcal{F}^i))$ 
5     if  $res = \text{"satisfiable"}$  then return  $\tau$ 
6      $w^\kappa \leftarrow \text{MINW}(\kappa)$ 
7     for  $b \in \kappa$  do
8        $w(b) \leftarrow w(b) - w^\kappa$ 
9      $\mathcal{F}^{i+1} \leftarrow \mathcal{F}^i \cup \text{RELAX}(\kappa, w^\kappa)$ 

```

MaxSAT solving. When invoked on an instance \mathcal{F} , the algorithm begins by initializing a working instance \mathcal{F}^1 to \mathcal{F} on Line 2. Then the main search loop (Lines 3-9) is started. In each iteration of the loop, a core of the current working instance \mathcal{F}^i is extracted on Line 4 using the assumption interface of the underlying SAT solver [14]. In Algorithm 1 the use of the SAT solver is abstracted into the function `EXTRACT-CORE` that takes as input the current instance and its blocking variables (to use as assumptions). The function returns a triple (res, κ, τ) . If $res = \text{"satisfiable"}$ then τ is an assignment satisfying $\text{HARD}(\mathcal{F}^i)$ that sets $\tau(b) = 0$ for each $b \in \mathcal{B}(\mathcal{F}^i)$. Such τ has $\text{COST}(\mathcal{F}^i, \tau) = 0$ and will be optimal for both \mathcal{F}^i and \mathcal{F} , so Algorithm 1 terminates and returns it (Line 5).

If the SAT solver instead reports unsatisfiable, it will return a core κ of \mathcal{F}^i expressed as a subset of $\mathcal{B}(\mathcal{F}^i)$ (i.e. as a clause over blocking variables). Next, the variables in κ are refined (Lines 6-8). During refinement, the weight of each blocking variable $b \in \kappa$ is lowered by $\text{MINW}(\kappa)$. An important intuition here is that at least one of the variables in κ will have its weight set to 0, and will thus not be treated as a blocking variable in subsequent iterations. Refining the blocking variables essentially corresponds to *clause cloning* via assumptions (see, e.g., [11, 3, 21]), a common way for core-guided MaxSAT solvers to take soft clause weights into account.

After refining the blocking variables, the core κ is relaxed on Line 9 via the function `RELAX`. Core-guided MaxSAT algorithms differ mainly in the specifics of the `RELAX` function. For each κ of \mathcal{F}^i , at least one variable in κ needs to be assigned to 1 by any optimal solution to \mathcal{F}^i . Essentially all instantiations of `RELAX` we are aware of relax the instance by adding new clauses and blocking variables to \mathcal{F} that allow, in a controlled way, a single blocking variable in κ to be set to 1 in subsequent iterations. In this work we focus especially on the transformation used by the OLL core-guided MaxSAT algorithm [26, 1], defined as follows.

► **Definition 2.** Given $(\kappa, w) = (\{b_1, \dots, b_n\}, w)$, OLL implements `RELAX` by adding $\mathcal{O}(n \log n)$ new variables and $\mathcal{O}(n^2)$ new clauses corresponding to $(\sum_{k=1}^n b_k \geq (i+1)) \rightarrow b_i^\kappa$ for $i = 1, \dots, n-1$. Each b_i^κ is a new blocking variable of weight w .

An informal argument for the correctness of OLL, i.e., the fact that the final assignment returned by it will be an optimal solution to \mathcal{F} , is as follows (a formal argument can be found in [26]). Since κ is a core of \mathcal{F}^i , any optimal solution of \mathcal{F}^i will assign at least one $b \in \kappa$ to 1 (i.e., falsify at least one $(\neg b) \in \kappa$ on the clause-level). During refinement, at least one blocking variable will have its weight lowered to 0 and will not be considered a

blocking variable in subsequent iterations. This means that the SAT solver is free to assign it to 1 in subsequent iterations. At the same time, the clauses added by RELAX (detailed in Definition 2) enforce that for any $k > 1$ number of blocking variables in κ assigned to 1, $k - 1$ of the new blocking variables will also be unit propagated to 1, thus incurring more cost. In other words, the OLL relaxation allows one – but only one – of the blocking variables in κ to incur cost “for free” in subsequent iterations.

In the rest of the paper we use the name OLL to refer to Algorithm 1 with RELAX instantiated as in Definition 2.

4 Structure Sharing for Improving Core-Guided MaxSAT Solvers

In this section we present *structure sharing*, the main contribution of this work. Structure sharing is a technique that enables more compact core relaxation steps within the OLL algorithm. In addition to requiring fewer clauses, we will also demonstrate that core relaxation with structure sharing enables more propagation in the underlying SAT solver during subsequent iterations.

We begin by discussing the totalizer encoding for cardinality constraints [9], a common way of realizing the OLL algorithm, and weight-aware core extraction [11], a refinement to the standard way in which OLL extracts cores. Both of these are central for understanding structure sharing.

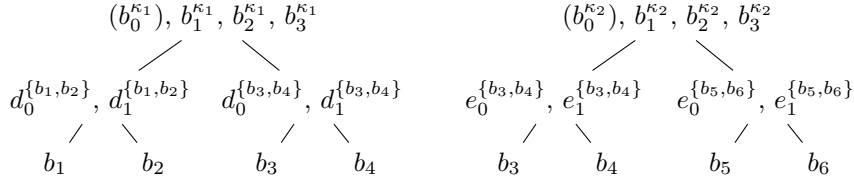
4.1 The Totalizer Encoding of Cardinality Constraints

For our purposes, a totalizer $\mathcal{T}(V)$ over a set $V = \{v_1, \dots, v_n\}$ of variables is a satisfiable CNF formula that defines a set $O(V) = \{o_0, \dots, o_{n-1}\}$ of output variables. Informally speaking, the output variables count the number of input variables set to true by assignments satisfying $\mathcal{T}(V)$. More precisely, if τ satisfies $\mathcal{T}(V)$, then $\tau(o_k) = 1$ if and only if $\sum_{v \in V} \tau(v) \geq k + 1$. A common way of realizing the OLL algorithm – used for example in the state-of-the-art solver RC2 – is to relax a core κ by adding a totalizer $\mathcal{T}(\kappa)$ to the instance and treating the output variables of $\mathcal{T}(\kappa)$ as blocking variables in subsequent iterations, i.e., setting $b_i^\kappa = o_i$.

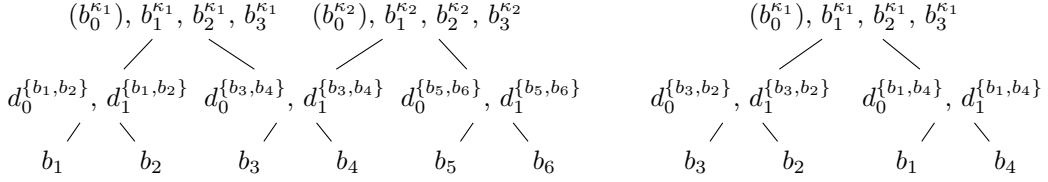
$\mathcal{T}(\kappa)$ can be viewed as a tree with $|\kappa|$ leaves each associated with a distinct variable of κ . As an example, Figure 1 (left) gives the tree representation of the totalizer $\mathcal{T}(\kappa)$ over $\kappa = \{b_1, b_2, b_3, b_4\}$. The root of the tree is associated with the output variables of the totalizer, i.e., the new blocking variables added when relaxing κ . Notice that since the definition of a core implies that at least one variable in κ is set to true by any solution, the output variable b_0^κ of $\mathcal{T}(\kappa)$ is often omitted in realizations of OLL, including RC2.

The non-leaf internal nodes are associated with the so-called linking variables of $\mathcal{T}(\kappa)$ used in the encoding of the semantics of the output variables. For an internal node D of a totalizer $\mathcal{T}(\kappa)$, let $\mathcal{S}(D) \subset \kappa$ be the set of variables associated with the leaves of the subtree rooted at D . The linking variables $\{d_0^{\mathcal{S}(D)}, \dots, d_{|\mathcal{S}(D)|-1}^{\mathcal{S}(D)}\}$ associated with D count the number of variables of $\mathcal{S}(D)$ set to true by a satisfying assignment τ of $\mathcal{T}(\kappa)$. More precisely, if τ satisfies $\mathcal{T}(\kappa)$, then $\tau(d_k^{\mathcal{S}(D)}) = 1$ if and only if $\sum_{b \in \mathcal{S}(D)} \tau(b) \geq k + 1$. Essentially, the linking variables of a totalizer can be viewed as the output variables of a sub-totalizer built over $\mathcal{S}(D)$.

Consider a totalizer $\mathcal{T}(\kappa)$ and one of its output variable b_k^κ . We say that the constraints encoding $(\sum_{b \in \kappa} b \geq k + 1) \rightarrow b_k^\kappa$ are the *implication constraints* of b_k^κ . Analogously, we say that the constraints encoding $b_k^\kappa \rightarrow (\sum_{b \in \kappa} b \geq k + 1)$ are the *equivalence constraints* of b_k^κ . The terminology is extended to linking variables $d_k^{\mathcal{S}(D)}$; the implication constraints



■ **Figure 1** The structure of totalizers built when relaxing cores $\{b_1, b_2, b_3, b_4\}$ (left) and $\{b_3, b_4, b_5, b_6\}$ (right).



■ **Figure 2** Left: The structure of totalizers when relaxing the cores $\{b_1, b_2, b_3, b_4\}$ and $\{b_3, b_4, b_5, b_6\}$ with structure sharing. Right: An alternative totalizer for relaxing $\{b_1, b_2, b_3, b_4\}$.

of $d_k^{S(D)}$ encode $\left(\sum_{b \in S(D)} b \geq k+1\right) \rightarrow d_k^{S(D)}$ and the equivalence constraints encode $d_k^{S(D)} \rightarrow \left(\sum_{b \in S(D)} b \geq k+1\right)$. The implication and equivalence constraints of the entire $\mathcal{T}(\kappa)$ are then the implication and equivalence constraints of each of its output and linking variables.

It is fairly straight-forward to show that for a realization of OLL that makes use of totalizers, it is sufficient to add only the implication constraints of the totalizers that relax cores. In fact – to the best of our understanding – most realizations of OLL that use totalizers do not add equivalence constraints when relaxing cores at all. The motivation for leaving out the equivalence clauses is to keep the size of the working instance smaller. While redundant in the context of computing optimal solutions of MaxSAT instances, the equivalence constraints of totalizers can however enable additional propagations in the SAT solver during subsequent iterations.

► **Example 3.** Consider the two totalizer structures $\mathcal{T}(\kappa_1)$ and $\mathcal{T}(\kappa_2)$, depicted on the left and right sides of Figure 1, respectively. Assume that we fix $b_1^{\kappa_1} = b_5 = 1$ and $b_2 = 0$. The implication clause $(\neg b_5 \vee e_0^{\{b_5, b_6\}})$ of $\mathcal{T}(\kappa_2)$ then propagates $e_0^{\{b_5, b_6\}} = 1$. This is the the only propagation that happens due to the implication constraints. The equivalence constraints of $\mathcal{T}(\kappa_1)$ additionally propagate $d_1^{\{b_1, b_2\}} = 0$ due to the clause $(\neg d_1^{\{b_1, b_2\}} \vee b_2)$ and $d_0^{\{b_3, b_4\}} = 1$ due to the clause $(\neg b_1^{\kappa_1} \vee d_1^{\{b_1, b_2\}} \vee d_0^{\{b_3, b_4\}})$.

4.2 Weight-Aware Core Extraction

Weight-aware core extraction (WCE) is an essential techniques towards structure sharing as proposed in this work. Originally proposed for the PMRES [28] core-guided MaxSAT algorithm in [11], WCE enables the extraction of multiple cores during a single iteration of core-guided MaxSAT solving by using information on clause weights during core extraction.

Algorithm 2 details the generic abstraction CG discussed in Section 3 extended with WCE. In short, the algorithm delays the core-relaxation steps as long as possible. When a core κ is obtained, the blocking variables in κ are refined as before. However, instead of immediately invoking RELAX, CG+WCE instead stores $(\kappa, \text{MINW}(\kappa))$ in a set \mathcal{K} and invokes

■ **Algorithm 2** CG+WCE: CG with WCE.

Input: A MaxSAT instance \mathcal{F}
Output: An optimal solution τ to \mathcal{F}

```

1 begin
2    $\mathcal{F}^1 \leftarrow \mathcal{F}$ 
3   for  $i = 1, \dots$  do
4      $\mathcal{K} \leftarrow \emptyset$ 
5     while true do
6        $(res, \kappa, \tau) \leftarrow \text{EXTRACT-CORE}(\mathcal{F}^i, \mathcal{B}(\mathcal{F}^i))$ 
7       if  $res = \text{"satisfiable"}$  then break
8        $w^\kappa \leftarrow \text{MINW}(\kappa)$ 
9       for  $b \in \kappa$  do
10         $w(b) \leftarrow w(b) - w^\kappa$ 
11       $\mathcal{K} \leftarrow \mathcal{K} \cup \{(\kappa, w^\kappa)\}$ 
12    if  $\mathcal{K} = \emptyset$  then return  $\tau$ 
13     $\mathcal{F}^{i+1} \leftarrow \mathcal{F}^i \cup \bigcup_{(\kappa, w^\kappa) \in \mathcal{K}} \text{RELAX}(\kappa, w^\kappa)$ 

```

the SAT solver again. Recall that the weight of at least one of the blocking variables in κ will be lowered to 0 during variable refinement so it will not be treated as a blocking variable in subsequent iterations. In other words, the next call to EXTRACT-CORE is guaranteed to either obtain a new core, or report the instance to be satisfiable. When no new cores can be found (i.e. the solver reports satisfiable) the algorithm relaxes all cores stored in \mathcal{K} before reriterating. Termination occurs when no new cores can be found between two relaxation steps, that is when $\mathcal{K} = \emptyset$ on line 12.

While WCE was shown to be effective for PMRES in [11], currently we are unaware of any implementations of OLL that would make use of WCE. However, as we will discuss next, WCE is instrumental in enabling structure sharing.

4.3 Structure Sharing for OLL

With the necessary background on totalizers and WCE in place, we turn to detailing structure sharing. In order to motivate structure sharing we consider what happens when several, possibly overlapping, cores are extracted and relaxed by OLL. In particular, consider an instance \mathcal{F} with $\text{HARD}(\mathcal{F}) = \{(b_1 \vee b_2 \vee b_3 \vee b_4), (b_3 \vee b_4 \vee b_5 \vee b_6)\}$, $\mathcal{B}(\mathcal{F}) = \{b_1, \dots, b_6\}$, $w(b_1) = w(b_2) = w(b_5) = w(b_6) = 1$ and $w(b_3) = w(b_4) = 2$. The two cores of \mathcal{F} that are important for our discussion are $\kappa_1 = \{b_1, b_2, b_3, b_4\}$ and $\kappa_2 = \{b_3, b_4, b_5, b_6\}$. Note that since $2 = w(b_3) = w(b_4) > \text{MINW}(\kappa_1) = \text{MINW}(\kappa_2) = 1$, both cores can be extracted by OLL. Figure 1 depicts two totalizers $\mathcal{T}(\kappa_1)$ and $\mathcal{T}(\kappa_2)$ that can be built when OLL relaxes κ_1 and κ_2 . A key motivation for structure sharing is that both of these trees contain an internal node D for which $\mathcal{S}(D) = \{b_3, b_4\}$. As a consequence, the working instance obtained after relaxing both cores contains separate clauses defining the linking variables $d_0^{\mathcal{S}(D)}$ and $d_1^{\mathcal{S}(D)}$ and $e_0^{\mathcal{S}(D)}$ and $e_1^{\mathcal{S}(D)}$, even though the semantics of them are exactly the same.

By structure sharing in this example, we refer to sharing the subtree with the leaves $\{b_3, b_4\}$ between both $\mathcal{T}(\kappa_1)$ and $\mathcal{T}(\kappa_2)$, resulting in the structure depicted in Figure 2 (left). Generally, correctness of structure sharing follows from the fact that it does not alter the semantics of the new blocking variables added in core relaxation. In addition to

decreasing the size of the working instance (for example, the structures depicted in Figure 1 require in total 24 clauses while the equivalent single structure shown in Figure 2 (left) only requires 21) structure sharing can also both *decrease* the number of – essentially unnecessary – propagations that the SAT solver does in subsequent iterations, as well as *increase* the number of further propagations. For an example of the former, note that when the two totalizers are disjoint (as in Figure 1), fixing one of the variables in $\{b_3, b_4\}$ to 1 propagates both $d_0^{S(D)} = 1$ and $e_0^{S(D)} = 1$. In contrast, with structure sharing (Figure 2, left) only the variable $d_0^{S(D)} = 1$ is propagated. For an example of the latter, consider the following.

► **Example 4.** Consider the shared totalizer structure depicted in Figure 2 and assume again the fixings $b_1^{\kappa_1} = b_5 = 1$ and $b_2 = 0$. Similarly to Example 3, these propagate $d_0^{\{b_5, b_6\}} = d_0^{\{b_3, b_4\}} = 1$. However, in this structure, the implication clause $(\neg d_0^{\{b_3, b_4\}} \vee \neg d_0^{\{b_5, b_6\}} \vee b_1^{\kappa_2})$ also propagates $b_1^{\kappa_2} = 1$. We emphasize that $b_1^{\kappa_2} = 1$ can not be derived by unit propagation alone in the disjoint structures depicted in Figure 1.

Recall that, while the implication constraints of totalizers are needed in order to compute an optimal solution to a MaxSAT instance, the equivalence constraints are not. Instead, there is an inherent-trade off between the number of equivalence clauses that are added, and the number of additional propagations they enable. While extra propagations have the potential of speeding up subsequent SAT solver calls, adding too many clauses may instead end up slowing down the solver. Structure sharing seeks to limit the number of equivalence clauses added by identifying which ones of them are most likely to result in additional propagation. Example 4 offers some intuition on this. Note that propagating $b_1^{\kappa_2} = 1$ in the shared subtree does not require all of the equivalence constraints of the structure. Instead, the equivalence constraints of all nodes except for the ones in the subtree rooted at $d_0^{\{b_3, b_4\}}, d_1^{\{b_3, b_4\}}$ suffice. We detail the realization of structure sharing and the selective addition of equivalence constraints in the next section.

► **Remark 5.** We shortly note the distinguishing features of structure sharing as proposed to related recently proposed techniques from the literature. The *iterative encoding of totalizers* [23] allows for extending a single totalizer with more inputs. While commonly used in MaxSAT algorithms that extend cardinality constraints during search (such as MSU3 [3, 22]), in contrast to structure sharing, the iterative encoding it is not applicable to OLL or PMRES. The *WPM3 core-guided MaxSAT algorithm* [5] uses the semantics of blocking variables introduced during core relaxations to maintain knowledge of the global core structure in order to obtain a better encoding of any core containing blocking variables introduced in previous relaxation steps. However, in contrast to structure sharing, WPM3 does not maintain knowledge of blocking variables potentially being found in multiple cores. If a set of previously relaxed blocking variables are extracted as part of a core, the structure introduced by WPM3 will be disjoint from the structure introduced when originally relaxing the variables. In this sense structure sharing as we propose it here is orthogonal to the ideas of WPM3. Indeed, structure sharing and WCE could be integrated into WPM3 as well and appears interesting future work. The *abstract cores* technique proposed in the context of the implicit hitting set (IHS) approach to MaxSAT [10] also aims to build totalizers with variables that often appear in cores together being assigned to the same subtree. However, the abstraction sets over which totalisers are built in the abstract cores technique are not overlapping. In that sense, the way totalisers are used in the abstract cores technique resembles more closely the incremental encoding. Finally, the idea of sharing structure with the aim of obtaining *more effective representations of weight rules* (tightly connected to pseudo-Boolean constraints) in the context of answer set programming was explored in [13].

4.4 Realizing Structure Sharing

Structure sharing is realized in an OLL algorithm making use of WCE with a greedy procedure. Given a set \mathcal{K} of cores to be relaxed, the procedure maintains a collection \mathcal{S} of sets of blocking variables initialized to \mathcal{K} and proceeds iteratively. In each iteration the sets in \mathcal{S} are compared pairwise in order to find a maximal subset M of blocking variables shared by two sets in \mathcal{S} . After finding such M , the set \mathcal{S} is refined by: i) adding a pointer $s \rightarrow M$ for each $s \in \mathcal{S}$ for which $M \subseteq s$, ii) removing the variables of M from every set $s \in \mathcal{S}$ for which $M \subseteq s$ and iii) adding M to \mathcal{S} . Finally, a totalizer is built for every set in \mathcal{S} and linked following the pointers, i.e., if there is a pointer $a \rightarrow b$ for $a, b \in \mathcal{S}$, then $\mathcal{T}(b)$ is linked as a subtree to $\mathcal{T}(a)$.

We note that the use of WCE is central in enabling structure sharing. This is due to the fact that there are several possible totalizer structures for a core. Figure 2 (right) depicts another possible structure of $\mathcal{T}(\kappa_1)$ (from before) that does not include any subtrees that can be shared with $\mathcal{T}(\kappa_2)$ in Figure 1 (nor with any other possible structure of $\mathcal{T}(\kappa_2)$). The existence of different choices of equivalent totalizer structures makes it very difficult – if not impossible – to realize structure sharing without WCE. As an example, assume that OLL *without* WCE is invoked on \mathcal{F} and the core κ_1 is extracted first. After refining the blocking variables, a totalizer $\mathcal{T}(\kappa_1)$ is built and added to the working instance. At this point, there is no obvious reason to prefer either one of the structures depicted in Figure 1 left or Figure 2 (right) for building $\mathcal{T}(\kappa_1)$; however, only the tree in Figure 1 enables structure sharing.

Selective Addition of Equivalences

After building shared totalizer structures for a set of cores, we next select which equivalence constraints should be included in the working instance. Intuitively, the aim is to add – in some sense – useful equivalence constraints that enable propagation without inflating the size of the working instance beyond the capabilities of modern SAT solvers.

Example 4 provides more intuition. The additional propagation enabled by structure sharing follows from the outputs of shared structures being propagated to 1. Furthermore, such outputs are propagated to 1 due to either (i) the implication constraints of the substructure itself (which have to be included anyway) or (ii) the equivalence constraints of the rest of the structure.

More generally, we propose to selectively include equivalence constraints by looping over the leaves of the structure – which are treated as roots of a subtree containing a single node – and the roots of any shared subtrees. For each node two values are computed: 1) the number of decisions needed before the equivalence constraints would propagate the shared variable to true; the decisions are either setting some leaves outside the subtree to false, or setting the output variables of the structure to true, and 2) the (estimated) total number of equivalence clauses for the nodes outside of the subtree, which is quadratic in the number of leaves in the structure outside of the subtree. If both of these numbers are below some user given parameter, we include the equivalence constraints of all variables outside of the subtree. Even in structures with shared subtrees both values are computed w.r.t. the totalizer tree corresponding to a single core.

► **Example 6.** Consider the shared structure in Figure 2 and the root of the shared subtree corresponding to the variables $d_0^{\{b_3, b_4\}}, d_1^{\{b_3, b_4\}}$. Assume that OLL has managed to derive the unit clause $(b_1^{\kappa_1})$, i.e. fix $b_1^{\kappa_1} = 1$. This could happen for example via the so-called core-exhaustion heuristic [18]. In order for the equivalence constraints of the structure corresponding to κ_1 to propagate $d_0^{\{b_3, b_4\}} = 1$ one more decision is needed, either $b_1 = 0$, $b_2 = 0$ or $b_2^{\kappa_1} = 1$. Similarly, in order for the equivalence constraints to propagate $d_1^{\{b_3, b_4\}} = 1$

two additional decisions are needed, one of the following four sets of alternatives: (i) $b_1 = b_2 = 0$, (ii) $b_2^{\kappa_1} = 1 \wedge b_1 = 0$, (iii) $b_2^{\kappa_1} = 1 \wedge b_2 = 0$, or (iv) $b_2^{\kappa_1} = b_3^{\kappa_1} = 1$. As for the the estimated number of equivalence clauses for the nodes outside of the subtree; in this case there are two leaves outside of the subtree (b_1, b_2) so the estimate is 4 constraints.

Similarly, in order for the equivalence constraints of the structure corresponding to κ_2 to propagate $d_0^{\{b_3, b_4\}} = 1$, two decisions are needed. One of the following four sets of alternatives: (i) $b_5 = b_6 = 0$, (ii) $b_1^{\kappa_2} = 1 \wedge b_5 = 0$, (iii) $b_1^{\kappa_2} = 1 \wedge b_6 = 0$, or (iv) $b_1^{\kappa_2} = b_2^{\kappa_2} = 1$. For propagating $d_1^{\{b_3, b_4\}} = 1$ three decisions are needed, one of the following four sets of alternatives: (i) $b_5 = b_6 = 0 \wedge b_1^{\kappa_2} = 1$, (ii) $b_1^{\kappa_2} = b_2^{\kappa_2} = 1 \wedge b_5 = 0$, (iii) $b_1^{\kappa_2} = b_2^{\kappa_2} = 1 \wedge b_6 = 0$, (iv) $b_1^{\kappa_2} = b_2^{\kappa_2} = b_3^{\kappa_2} = 1$. This structure also has two leaves outside of the shared tree (b_5, b_6) , so the estimated number of added equivalence constraints is again 4.

5 Empirical Evaluation

We evaluate the impact of WCE and structure sharing on OLL. For the evaluation, we extending the state-of-the-art RC2 MaxSAT solver [18] implementation of OLL with WCE and structure sharing, using the RC2 version that performed best in MaxSAT Evaluation 2020. The implementation is available online at: <https://bitbucket.org/coreo-group/cgss/>.

More specifically, we compare the following solvers:

- **RC2**: the original RC2 solver from MaxSAT Evaluation 2020.
- **RC2***: our **RC2** refactorization with a reimplement of totalizers geared towards implementing WCE and structure sharing (SS).
- **RC2*+WCE**: **RC2*** extended with WCE.
- **RC2*+WCE+SS**: **RC2*+WCE** extended with SS and selective addition of equivalences.

In the implementation of SS, we stop the greedy procedure for computing shareable structures once the set M containing overlapping blocking variables has $|M| < 16$. Furthermore, for **RC2*+WCE+SS** we only add equivalence constraints of a node in a shared subtree if doing so adds at most 50 clauses and at most 50 decisions are needed in order for the variable to be propagated to 1. All of these parameter values were chosen based on preliminary experimentation.

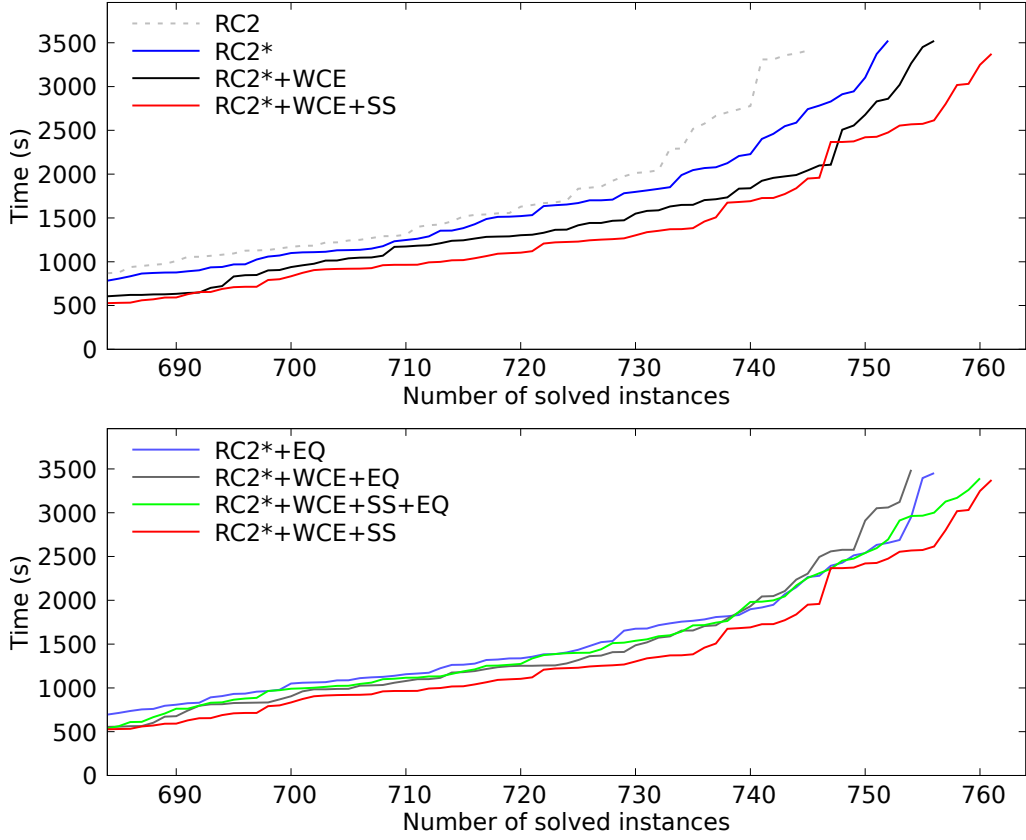
As benchmarks we used the combined set of 1033 weighted MaxSAT instances from the complete tracks of MaxSAT Evaluation 2019 and 2020. The experiments were run single-threaded using 2.6-GHz Intel Xeon E5-2670 processors. A per-instance time limit of 3600 seconds and a memory limit of 32GB was enforced.

Figure 3 (top) shows the effect of WCE and structure sharing on RC2. First, we observe that our refactorization **RC2*** actually improves on the performance of **RC2**, improving on the number of solved instances from 745 to 752. Employing WCE and SS allows for solving the greatest number of 761 instances (**RC2*+WCE+SS**). The marginal impact of SS is significant, as employing WCE alone allows for solving 756 instances (**RC2*+WCE**) (As a side-note, to the best of our understanding, this is the first time that WCE is integrated to OLL and shown to provide performance improvements).

For more details on impact of SS and in particular the idea of selective addition of equivalences, consider Figure 3 (bottom). First recall that **RC2*+WCE+SS** employs selective addition of equivalence. Now, **RC2*+EQ**, **RC2*+WCE+EQ** correspond to **RC2*** and **RC2*+WCE**, respectively, which add all equivalence constraints of each totalizer when relaxing cores. The solver **RC2*+WCE+SS+EQ** corresponds the modification of **RC2*+WCE+SS** that adds all equivalence constraints of each totalizer. Somewhat

■ **Table 1** Detailed comparison of RC2*, RC2*+WCE and RC2*+WCE+SS per benchmark domain.

Benchmark family	RC2*		RC2*+WCE		RC2*+WCE+SS	
	solved	PAR-2	solved	PAR-2	solved	PAR-2
wpms	3	18.12	3	19.68	3	18.18
dcalculus	27	21.15	27	21.50	27	21.26
MLIC	5	58737.43	5	57821.14	5	57885.03
causal-discovery	18	62988.31	18	61955.50	19	56543.76
relational-inference	5	26912.08	5	27023.60	5	26886.39
tcp	20	23698.21	20	26803.44	19	30008.36
protein_ins	11	738.45	11	697.47	11	564.86
staff-scheduling	2	74402.74	2	74108.37	2	73207.89
planning	4	1.69	4	1.85	4	1.82
scSequencing	19	80608.19	19	80678.55	19	80650.86
ParametricRBAC	32	59118.70	32	59603.27	32	59639.39
InterpretableClassifiers	7	36057.44	7	36062.51	7	36056.42
auc_paths	8	910.97	8	310.29	8	396.19
csg_2020	30	231.39	30	277.10	30	230.93
scp	10	953.63	10	610.38	10	496.72
preference_planning	16	236.30	16	372.53	16	403.98
pseudoBoolean	8	7201.04	8	7201.07	8	7201.07
lisbon-wedding	6	131107.84	6	131112.42	6	133287.46
frb	19	175.57	19	1501.58	19	2168.53
qcp	9	5.04	9	5.05	9	5.05
log	3	43204.83	3	43202.23	3	43201.83
miplib	2	43294.76	2	43320.13	2	43297.22
warehouses	8	217.34	8	154.79	8	69.18
haplotyping-pedigrees	29	691.74	29	733.20	29	668.02
min-width	8	240055.37	8	238353.95	8	237878.82
drmx-atmostk	17	516.92	17	675.43	17	443.45
upgradeability	19	111.14	19	126.34	19	110.18
correlation-clustering	10	96843.58	11	89478.17	11	90682.68
maxcut	0	43200.00	0	43200.00	0	43200.00
packup	5	30.83	5	30.59	5	28.12
abstraction-refinement	11	9251.50	11	10153.65	11	8655.52
railroad_sc	0	43200.00	0	43200.00	0	43200.00
security-witness	30	10481.58	30	9973.36	30	9291.70
rna-alignment	23	99.31	23	133.67	23	100.90
drmx-cryptogen	17	2432.11	17	2541.40	17	2456.42
CSG	10	294.89	10	315.85	10	329.77
css-refactoring	11	179.67	11	243.71	11	166.37
Security-CriticalCyber	39	232.85	39	262.01	39	235.87
ramsey	2	93978.71	2	94221.14	3	89300.00
railroad_scheduling	0	57600.00	0	57600.00	0	57600.00
wcsp	21	7.39	21	6.95	21	7.07
set-covering	9	14668.38	9	14795.22	9	14719.04
max-realizability	25	47187.54	26	40544.80	26	40382.86
MinWeightDomSet	0	50400.00	0	50400.00	0	50400.00
BTBNSL	7	123625.64	8	118890.94	8	118397.70
auc_regions	6	9280.05	6	9315.44	7	1749.98
mpe	24	38815.48	26	23566.10	27	17579.18
max-prob-min-cuts	30	113.40	30	131.24	30	114.82
hs-timetabling	2	83441.11	2	82579.77	2	83541.69
metro	27	3971.78	27	3713.86	27	3833.46
timetabling	14	60924.14	15	56410.41	15	55920.20
spot5	8	22237.25	9	15141.17	8	21657.60
wcnf_gz	8	64828.57	8	64847.09	8	64841.45
up-up98	6	15.90	6	16.55	6	14.97
scpnr	0	14400.00	0	14400.00	0	14400.00
dimacs_mod	1	86516.74	1	86589.22	1	86609.38
af-synthesis	17	59316.18	15	65557.61	15	66009.69
railway-transport	2	23206.20	1	28971.47	1	28948.40
RBAC	9	154220.94	7	166452.75	8	160087.41
shiftdesign	16	6726.73	16	8060.46	16	6949.01
auctions	11	34949.24	13	21297.08	15	4909.61
binaryNN	4	12465.98	4	12696.34	4	12413.52
dir	2	8171.10	2	7317.87	2	7373.82
SUM	752	2169531.16	756	2135809.26	761	2097451.06

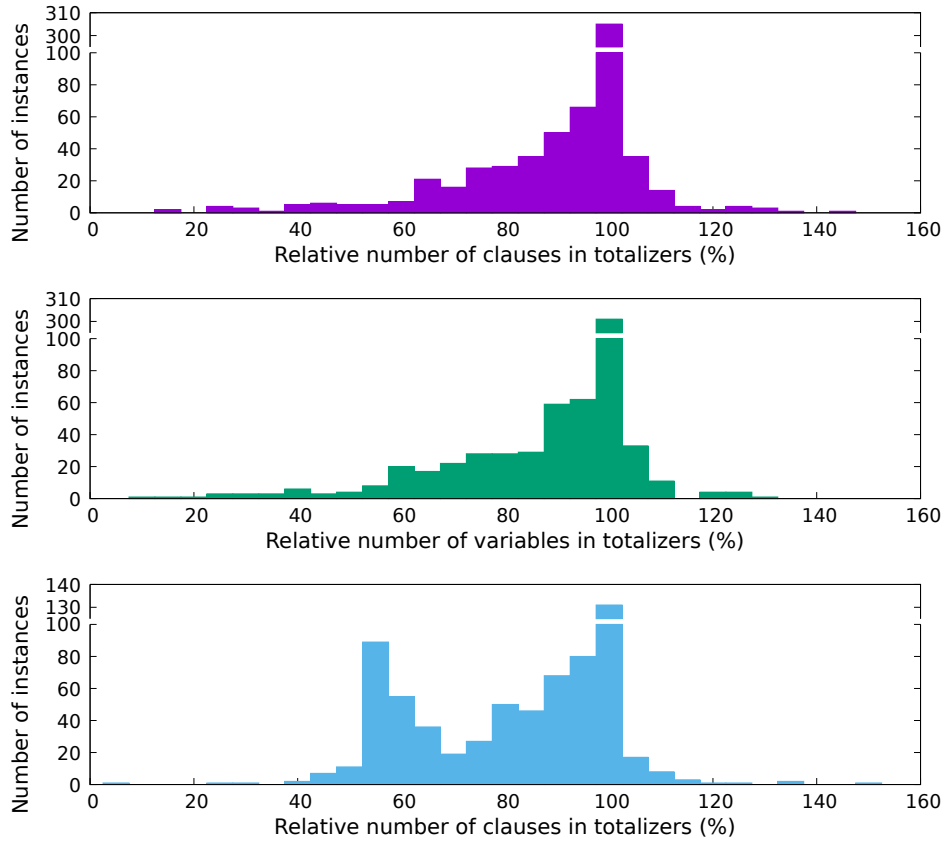


■ **Figure 3** Top: The effect of WCE and structure sharing (SS) on OLL. Bottom: Effect of equivalence constraints on the different variants of OLL.

surprisingly, adding all equivalence constraints to the totalizers in **RC2*** actually increases performance in our evaluation: **RC2*+EQ** solves 756 instances. In contrast, including all equivalence constraints into **RC2*+WCE** degrades performance, **RC2*+WCE+EQ** solves 754 instances. Furthermore, **RC2*+WCE+SS+EQ** exhibits slightly weaker performance than **RC2*+WCE+SS**, solving in total 760 instances.

Further detailed results are provided in Table 1. Here we detail for each of the 63 benchmark domains within the benchmark set, the number of instances solved and the PAR-2 score (i.e. cumulative runtimes, with timeouts penalized by a factor of 2) of **RC2***, **RC2*+WCE** and **RC2*+WCE+SS**. We observe that **RC2*+WCE+SS** solves the greatest number of instances in 5 domains and obtains the best (lowest) PAR-2 score in 22 further domains, achieving overall a PAR-2 score that is $\sim 98\%$ of the PAR-2 score of **RC2*+WCE** and $\sim 96\%$ of the PAR-2 score of **RC2***.

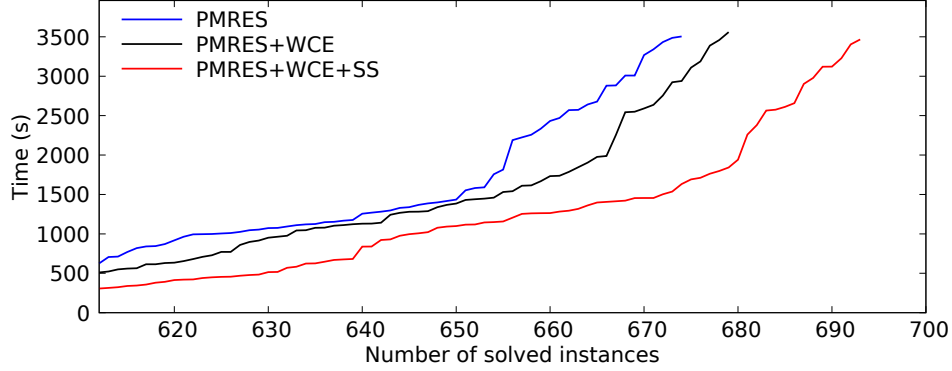
Figure 4 demonstrates the relative number of variables and clauses added during the solving process. In more detail, Figure 4 (top) provides a comparison of the the relative of implication constraints added during solving by **RC2*+WCE** and **RC2*+WCE+SS**. Here the height of the bar at a x -axis value p gives the number of instances for which the number of implication constraints introduced by **RC2*+WCE+SS** was $p\%$ of the number of implication constraints introduced by **RC2*+WCE**. We observe that structure sharing indeed decreases the number of implication constraints added on a significant number of instances. At times **RC2*+WCE+SS** adds less than 20% of the implication constraints



■ **Figure 4** Impact of structure sharing on the number of implication clauses (top) and the number of variables (middle); impact of selective addition of equivalences on the number of equivalence constraints (bottom).

added by **RC2*+WCE**. Figure 4 (middle) analogously provides the relative number of variables introduced. We again observe that for many benchmark instances the working instance of **RC2*+WCE+SS** contains fewer variables than the one of **RC2*+WCE**. Finally, Figure 4 (bottom) provides the relative number of equivalence constraints introduced by **RC2*+WCE+SS** and **RC2*+WCE+SS+EQ**. We observe that selective addition of equivalences introduces fewer clauses in many instances, which together with the runtime results presented in Figure 3 (bottom) suggests that the technique achieves the desideratum of introducing fewer equivalence clauses without sacrificing the potential benefits (for example, in the form of additional propagations).

Finally, we note that structure sharing is a general technique and not limited to the OLL algorithm. In order to demonstrate this, we reimplemented the PMRES algorithm in the same framework as RC2 (based on PySAT [17]). Figure 5 demonstrates the effect of WCE and structure sharing on the PMRES algorithm. We observe that both WCE and SS have a significant positive impact on the performance of PMRES, improving from 674 instances solved by the base algorithm to 679 when extended with WCE and further improving the number of solved instances to 693 when extended by SS.



■ **Figure 5** The effect of WCE and structure sharing (SS) on PMRES.

6 Analysis on the Impact of WCE on OLL

Finally, complementing the empirical observations on the impact of WCE on OLL, we provide theoretical insights into the effects of weight-aware core extraction on the OLL algorithm. In particular, while the original paper proposing WCE [11] demonstrated its effectiveness on the PMRES algorithm in practice, it did not provide insight into the effect that WCE can have on core-guided MaxSAT algorithms on a theoretical level. In this section, we show that WCE can both decrease, and increase the number of iterations required by the OLL algorithm, depending on the instance. (We note that these observations extend to PMRES in a relatively straightforward way.)

In the following, let OLL be Algorithm 1 with RELAX instantiated according to Definition 2 and OLL+WCE the OLL algorithm extended with WCE.

The following observation will be useful in the analysis.

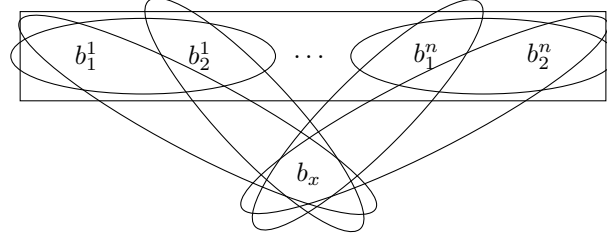
► **Observation 7.** *Invoke Algorithm 1 on instance \mathcal{F} . Assume that $(\kappa_1, \dots, \kappa_n)$ is the sequence of cores extracted after n iterations. Then $\sum_{i=1}^n \text{MINW}(\kappa_i) \leq \text{COST}(\mathcal{F})$. Furthermore, equality holds only if Algorithm 1 terminates after relaxing all of the cores.*

In other words, cores extracted by Algorithm 1 provide a lower bound on the optimal cost, and the algorithm terminates only when that lower bound equals the optimal cost.

Observation 7 forms the basis for a heuristic that is employed by most implementations of core-guided MaxSAT solvers that we are aware of. We say that Algorithm 1 uses bounds if it maintains a lower bound LB – computed using Observation 7 – on the optimal cost of the instance \mathcal{F} being solved. As soon as any solution τ is obtained, its cost $\text{COST}(\mathcal{F}, \tau)$ is compared to the lower bound. If $\text{LB} = \text{COST}(\mathcal{F}, \tau)$, the algorithm immediately terminates, without invoking RELAX. In our setting, intermediate solutions are only obtained by OLL+WCE. In practice, implementations of OLL also obtain non-optimal solutions during search via the so-called stratification [2] heuristic.

WCE can allow Algorithm 1 using bounds to terminate without relaxing any cores. Consider an instance \mathcal{F}^n with $\mathcal{B}(\mathcal{F}^n) = \{b_1, \dots, b_{n+3}\}$ that contains the clauses $(b_i \vee b_{n+2})$ for $i = 1 \dots n+1$ and $(b_{n+1} \vee b_{n+3})$. Assume that $w(b_i) = 1$ for $i = 1 \dots (n+1), (n+3)$ and $w(b_{n+2}) = n+2$. This instance has one optimal solution τ of cost $\text{COST}(\mathcal{F}^n, \tau) = \text{COST}(\mathcal{F}^n) = n+1$ that sets $\tau(b_i) = 1$ for $i = 1 \dots (n+1)$ and $\tau(b_{n+2}) = \tau(b_{n+3}) = 0$.

► **Proposition 8.** *Algorithm 1 extracts $n+1$ cores when computing an optimal solution to \mathcal{F}^n .*



■ **Figure 6** Structure of \mathcal{F}^n in Observations 12 and 13. The ellipses and rectangles illustrate the clauses.

Proof (Sketch). The result follows from the fact that any core κ extracted by Algorithm 1 on \mathcal{F}^n must have $\text{MINW}(\kappa) = 1$. This by Observation 7 implies that termination occurs only after $n + 1$ cores have been extracted. ◀

► **Proposition 9.** *OLL+WCE using bounds can terminate without relaxing any cores when invoked on \mathcal{F}^n .*

Proof. Let the first core extracted be $\{b_{n+1}, b_{n+2}\}$. The weight of both b_{n+1} and b_{n+2} is then lowered by 1 before invoking the SAT solver again. In the second iteration, the set $\mathcal{B}(\mathcal{F}^1)$ contains $\{b_1, \dots, b_n, b_{n+2}, b_{n+3}\}$. Assume that the algorithm continues in a similar manner, iteratively extracting a core of the form $\{b_i, b_{n+2}\}$ for each $i = 1 \dots (n + 1)$. Each extracted core κ will have $\text{MINW}(\kappa) = 1$, so at this point $\text{LB} = n + 1$. In the next SAT solver call $\mathcal{B}(\mathcal{F}^1) = \{b_{n+2}, b_{n+3}\}$ so the solver is invoked assuming $b_{n+2} = b_{n+3} = 0$. The instance is satisfied by an assignment τ that sets $\tau(b_i) = 1$ for $i = 1 \dots (n + 1)$ and $\tau(b_{n+2}) = \tau(b_{n+3}) = 0$. This solution has $\text{COST}(\mathcal{F}^n, \tau) = n + 1 = \text{LB}$ so OLL+WCE terminates without invoking the function RELAX. ◀

We thereby arrive at the following.

► **Theorem 10.** *There is a family of MaxSAT instances \mathcal{F}^n with $|\mathcal{B}(\mathcal{F}^n)| = \mathcal{O}(n)$ on which OLL using bounds is guaranteed to relax n cores, while OLL+WCE using bounds can terminate without relaxing any cores.*

In other words, there are instances on which the core relaxation steps of Algorithm 1 are redundant. In addition to (unnecessarily) increasing the size of the working instance, the redundant core relaxation steps can also increase the size of the MUSes of the working instance.

► **Observation 11.** *Invoke OLL on \mathcal{F}^n and assume that the first core extracted is $\kappa = \{b_{n+1}, b_{n+2}\}$, (the same as in the proof of Proposition 9). After refining the set of blocking variables and relaxing κ , the next working instance \mathcal{F}^2 has $\mathcal{B}(\mathcal{F}^2) = \{b_1, \dots, b_n, b_{n+2}, b_{n+3}, b_1^\kappa\}$, where b_1^κ is the new blocking variable introduced by RELAX. The set $\{b_1, b_{n+3}, b_1^\kappa\}$ is an MUS of \mathcal{F}^2 of size 3, i.e., larger than any of the MUSes of \mathcal{F}^n . Note that all MUSes of \mathcal{F}^n are of size 2 and that Algorithm 1 only extracts MUSes of \mathcal{F}^n in the proof of Proposition 9.*

So far we have shown that WCE can lower the number invocations of RELAX of Algorithm 1, thus decreasing the complexity of the working instance which alleviates a main bottleneck of core-guided MaxSAT solvers. However, we can also identify that WCE may also increase the number iterations. In the following, consider the instance \mathcal{F}^n with $\text{HARD}(\mathcal{F}^n) = \{(b_1^i \vee b_2^i), (b_1^i \vee b_x), (b_2^i \vee b_x) \mid i = 1 \dots n\} \cup \{(b_1^1 \vee b_2^1 \vee b_1^2 \vee b_2^2 \vee \dots \vee b_1^n \vee b_2^n)\}$, $\mathcal{B}(\mathcal{F}^n) = \{b_x\} \cup \{b_1^i, b_2^i \mid i = 1 \dots n\}$ and $w(b_1^i) = n$, $w(b_2^i) = n + 1$ for $i = 1 \dots n$ as well as

$w(b_x) = n$. The optimal cost of the instance is $\text{COST}(\mathcal{F}^n) = n^2 + n$ and an example of an optimal solution τ sets $\tau(b_1^1) = \tau(b_1^2) = \dots = \tau(b_1^n) = \tau(b_x) = 1$ and $\tau(b_2^j) = 0$ for $j = 1 \dots n$. The structure of \mathcal{F}^n is shown in Figure 6.

► **Observation 12.** *Invoke OLL using bounds on \mathcal{F}^n . Assume that the first core extracted is $\kappa_1 = \{b_1^i, b_2^i \mid i = 1 \dots n\}$ with $\text{MINW}(\kappa_1) = n$. After the first invocation of RELAX, the working instance \mathcal{F}^2 has $\mathcal{B}(\mathcal{F}^2) = \{b_2^i \mid i = 1 \dots n\} \cup \{b_x\} \cup \{b_1^{\kappa_1}, \dots, b_{2n-1}^{\kappa_1}\}$. In the next $n-1$ iterations, the algorithm can extract and relax the cores $\kappa_2 = \{b_1^{\kappa_1}\}, \dots, \kappa_n = \{b_{n-1}^{\kappa_1}\}$ and finally the core $\kappa_{n+1} = \{b_n^{\kappa_1}, b_x\}$, all having $\text{MINW}(\kappa_i) = n$. As $\sum_{i=1}^n \text{MINW}(\kappa_i) = n^2 + n = \text{COST}(\mathcal{F}^n)$ the algorithm will terminate after relaxing all $n+1$ of these cores.*

► **Observation 13.** *Invoke OLL+WCE using bounds on \mathcal{F}^n . Assume that the first core extracted is $\kappa_1 = \{b_1^i, b_2^i \mid i = 1 \dots n\}$ with $\text{MINW}(\kappa_1) = n$. After refining the blocking variables, the instance will have $\mathcal{B}(\mathcal{F}^1) = \{b_2^i \mid i = 1 \dots n\} \cup \{b_x\}$ with $w(b_2^i) = 1$ for all $i = 1 \dots n$ and $w(b_x) = n$. There are still n more cores of the form $\kappa_i = \{b_2^i, b_x\}$ with $\text{MINW}(\kappa_i) = 1$ to extract before the RELAX function is invoked. Notice that these cores can be extracted in any order. At this point, $n+1$ cores ($\kappa_1, \dots, \kappa_{n+1}$) have been extracted. Since $\sum_{i=1}^{n+1} \text{MINW}(\kappa_i) = 2n < n^2 + n$ (for $n > 1$), the algorithm does not terminate on the next SAT solver call; in particular, the algorithm needs to extract more than $n+1$ cores before terminating.*

Thereby we arrive at the following.

► **Theorem 14.** *For every $n \in \mathbb{N}$, there is a MaxSAT instance \mathcal{F}^n with $|\mathcal{B}(\mathcal{F}^n)| = \mathcal{O}(n)$, and a core κ of \mathcal{F}^n such that if κ is the first core extracted, then (i) OLL can terminate after extracting and relaxing in total $n+1$ cores, while (ii) OLL+WCE has to extract at least $n+2$ cores before terminating.*

7 Conclusions

The exact details of the transformations steps, which compile a newly extracted unsatisfiable core into the current working instance, are a key to efficient core-guided MaxSAT solving, and this is also where the various existing core-guided MaxSAT algorithms differ. We proposed a novel form of structure sharing that can be applied within the core extraction steps, aiming at speeding up runtimes of the core-guided approach as well as avoiding unnecessary introduction of additional variables and clauses to the working instances during iterations of the algorithms. In contrast to earlier approaches to lowering the number of introduced variables and clauses via incremental cardinality constraints, structure sharing is an intrinsically different approach. In particular, it builds on weight-aware core extraction, which allows for extracting multiple cores during a single iteration from weighted MaxSAT instances, and exploits shared substructures among cores for structure sharing. Putting structure sharing into practice in a state-of-the-art core-guided MaxSAT solver, we showed that structure sharing in combinations with WCE provides empirical runtime improvements. In addition to these main contributions, we also provided a theoretical analysis of the worst and best case impact of WCE on the central OLL MaxSAT algorithm.

References

- 1 Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub. Unsatisfiability-based optimization in CLASP. In Agostino Dovier and Vitor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, volume 17 of *LIPIcs*, pages 211–221. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.ICLP.2012.211.

- 2 Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving SAT-based weighted MaxSAT solvers. In Michela Milano, editor, *Principles and Practice of Constraint Programming – 18th International Conference, CP 2012, Québec City, QC, Canada, October 8–12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2012. doi:10.1007/978-3-642-33558-7_9.
- 3 Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Solving (weighted) partial MaxSAT through satisfiability testing. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing – SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 – July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 427–440. Springer, 2009. doi:10.1007/978-3-642-02777-2_39.
- 4 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.
- 5 Carlos Ansótegui and Joel Gabàs. WPM3: an (in)complete algorithm for weighted partial MaxSAT. *Artificial Intelligence*, 250:37–57, 2017. doi:10.1016/j.artint.2017.05.003.
- 6 Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins, editors. *MaxSAT Evaluation 2020: Solver and Benchmark Descriptions*, volume B-2020-2 of *Department of Computer Science Report Series B*. Department of Computer Science, University of Helsinki, Finland, 2020.
- 7 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. MaxSAT Evaluation 2018: New developments and detailed results. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):99–131, 2019. doi:10.3233/SAT190119.
- 8 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 24, pages 929–991. IOS Press BV, 2 edition, 2021. doi:10.3233/FAIA201008.
- 9 Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 – October 3, 2003. Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003. doi:10.1007/978-3-540-45193-8_8.
- 10 Jeremias Berg, Fahiem Bacchus, and Alex Poole. Abstract cores in implicit hitting set MaxSat solving. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020 – 23rd International Conference, Alghero, Italy, July 3–10, 2020. Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 277–294. Springer, 2020. doi:10.1007/978-3-030-51825-7_20.
- 11 Jeremias Berg and Matti Järvisalo. Weight-aware core extraction in SAT-based MaxSAT solving. In Christopher Beck, editor, *Principles and Practice of Constraint Programming – 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 – September 1, 2017. Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 652–670. Springer, 2017. doi:10.1007/978-3-319-66158-2_42.
- 12 Nikolaj Bjørner and Nina Narodytska. Maximum satisfiability using cores and correction sets. In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25–31, 2015*, pages 246–252. AAAI Press, 2015. URL: <http://ijcai.org/Abstract/15/041>.
- 13 Jori Bomanson, Martin Gebser, and Tomi Janhunen. Improving the normalization of weight rules in answer set programs. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence – 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24–26, 2014. Proceedings*, volume 8761 of *Lecture Notes in Computer Science*, pages 166–180. Springer, 2014. doi:10.1007/978-3-319-11558-0_12.
- 14 Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003. doi:10.1016/S1571-0661(05)82542-3.

- 15 Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In Armin Biere and Carla Gomes, editors, *Theory and Applications of Satisfiability Testing – SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006. doi:10.1007/11814948_25.
- 16 Federico Heras, António Morgado, and João Marques-Silva. Lower bounds and upper bounds for MaxSAT. In Youssef Hamadi and Marc Schoenauer, editors, *Learning and Intelligent Optimization – 6th International Conference, LION 6, Paris, France, January 16-20, 2012, Revised Selected Papers*, volume 7219 of *Lecture Notes in Computer Science*, pages 402–407. Springer, 2012. doi:10.1007/978-3-642-34413-8_35.
- 17 Alexey Ignatiev, António Morgado, and João Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018 – 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 2018. doi:10.1007/978-3-319-94144-8_26.
- 18 Alexey Ignatiev, António Morgado, and João Marques-Silva. RC2: an efficient MaxSAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):53–64, 2019. doi:10.3233/SAT190116.
- 19 Chu Min Li and Felip Manyà. MaxSAT, hard and soft constraints. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 903–927. IOS Press, 2 edition, 2021. doi:10.3233/FAIA201007.
- 20 Chu Min Li, Felip Manyà, and Jordi Planes. Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 86–91. AAAI Press, 2006. URL: <http://www.aaai.org/Library/AAAI/2006/aaai06-014.php>.
- 21 Vasco Manquinho, João Marques Silva, and Jordi Planes. Algorithms for weighted boolean optimization. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing – SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 – July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 495–508. Springer, 2009. doi:10.1007/978-3-642-02777-2_45.
- 22 João Marques-Silva and Jordi Planes. On using unsatisfiability for solving maximum satisfiability. *CoRR*, abs/0712.1097, 2007. arXiv:0712.1097.
- 23 Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental cardinality constraints for MaxSAT. In Barry O’Sullivan, editor, *Principles and Practice of Constraint Programming – 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer, 2014. doi:10.1007/978-3-319-10428-7_39.
- 24 Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. On using incremental encodings in unsatisfiability-based MaxSAT solving. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1):59–81, 2014. doi:10.3233/sat190102.
- 25 Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014 – 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer, 2014. doi:10.1007/978-3-319-09284-3_33.
- 26 António Morgado, Carmine Dodaro, and João Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In Barry O’Sullivan, editor, *Principles and Practice of Constraint Programming – 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer, 2014. doi:10.1007/978-3-319-10428-7_41.

- 27 António Morgado, Federico Heras, Mark Liffiton, Jordi Planes, and João Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013. doi:10.1007/s10601-013-9146-2.
- 28 Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided MaxSAT resolution. In Carla 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: <http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8513>.

A Linear Time Algorithm for the k -Cutset Constraint

Nicolas Isoart ✉

Université Côte d’Azur, Nice, France

Jean-Charles Régin ✉

Université Côte d’Azur, Nice, France

Abstract

In CP, the most efficient model solving the TSP is the Weighted Circuit Constraint (WCC) combined with the k -cutset constraint. The WCC is mainly based on the edges cost of a given graph whereas the k -cutset constraint is a structural constraint. Specifically, for each cutset in a graph, the k -cutset constraint imposes that the size of the cutset is greater than or equal to two. In addition, any solution contains an even number of elements from this cutset. Isoart and Régin introduced an algorithm for this constraint. Unfortunately, their approach leads to a time complexity growing with the size of the considered cutsets, i.e. with k . Thus, they introduced an algorithm with a quadratic complexity dealing with k lower or equal to three. In this paper, we introduce a linear time algorithm for any k based on a DFS checking the consistency of this constraint and performing its filtering. Experimental results show that the size of most of the k -cutsets is lower or equal than 3. In addition, since the time complexity is improved, our algorithm also improves the solving times.

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

Keywords and phrases k -Cutset, TSP, Linear algorithm, Constraint

Digital Object Identifier 10.4230/LIPIcs.CP.2021.29

Funding This work has been supported by the 3IA Côte d’Azur with the reference number ANR-19-P3IA-0002.

1 Introduction

The Traveling Salesman Problem (TSP) is a fundamental graph theory problem. It consists in finding a minimum cost cycle in a graph visiting all nodes. The TSP appeared in numerous domains such as biology with genome sequencing, industry with scan chains and electronic component drilling problems, positioning of very large telescopes, data clustering, scheduling problems and many others. The applications of TSP make it as fundamental as interesting: it is often an underlying problem of real-world problems.

Like many real-world problems, the search for the existence of a TSP is NP-Complete and finding the optimal one is NP-Hard. Thus, all classical methods designed for solving NP-Hard problems have been tried (MIP, CP, SAT, ...).

In order to solve the optimization version of the TSP without side constraints, the most efficient method is based on MIP: the so-called specialized solver Concorde [1]. It is mainly based on an LP relaxation of the TSP obtained by relaxing the integrity and the subtour constraints in combination with structural cutting planes. Indeed, a large number of cutting plane algorithms correct the structural defects of the LP relaxation lower bound. Simple ones such as imposing the 2-connectivity of a graph, and more complex ones such as the so-called Comb inequalities. Nevertheless, no polynomial time algorithm is known at this time to detect whether a graph does not violate a Comb inequality. Thus, a large number of polynomial algorithms have been developed in order to consider only particular cases of Comb inequalities [4, 6, 3, 13].



© Nicolas Isoart and Jean-Charles Régin;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

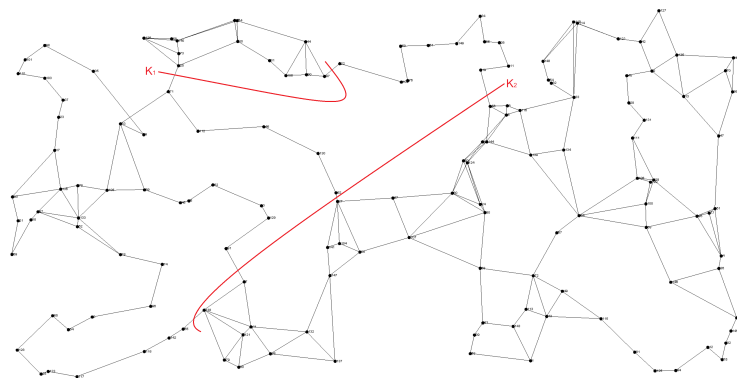
Editor: Laurent D. Michel; Article No. 29; pp. 29:1–29:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

However, it appears that a TSP is often combined with other constraints. For instance, precedence constraints, TSPTW where there is a time window to visit a node. Thus, Concorde can no longer be used for these problems and CP becomes a good candidate because it is more robust to side constraints. The most efficient method at this time solving the TSP in CP is the Weighted Circuit Constraint (WCC) [2] in combination with the structural k -cutset constraint [10]. The optimization part of the WCC is based on the Lagrangian Relaxation (LR) of Held and Karp [8, 9]. The lower bound of the LR is computed by selecting a node x with its two lowest cost neighbors and a minimum spanning tree in the graph without x , i.e. a 1-tree. If we find a minimum 1-tree such that all its nodes have exactly two neighbors, then an optimal solution is obtained. Thus, the 1-tree is derived through the LR process until an optimal solution is obtained. Unfortunately, solving optimally the TSP with a LR can be extremely slow. The WCC integrates filtering algorithms based on the edge costs, the 1-tree cost and a degree constraint on the nodes. With this model, Benchimol et al. were able to obtain a competitive method with Concorde for problems with less than 100 nodes. In addition, the integration of the k -cutset constraint improved the competitiveness of this method. The idea of this constraint is that each cutset of a graph must contain at least two edges and any solution contains an even number of elements from this cutset. Actually, the CP model is based on a single graph variable with mandatory and optional edges. Thus, the k -cutset constraint tries to detect inconsistency in mandatory edges and to filter optional edges.



■ **Figure 1** Graph kroA150 from TSPLib [14] while solving in a CP solver. K_1 is a 2-cutset and K_2 is a 4-cutset.

For instance, we show two k -cutsets in the graph of Figure 1. The 2-cutset K_1 contains 2 edges and the 4-cutset K_2 contains 4 edges.

Isoart and Régim [10] introduced a quadratic algorithm for the k -cutset constraint checking the consistency and performing some filtering operations. It is based on a 2-edge-connected subgraph and Tsin's algorithm [18]. Note that their algorithm only handle $k \leq 3$. This limitation is set because the number of k -cutsets in a graph is exponential: this corresponds to all possible partitions of the graph nodes, i.e. 2^n . However, we are not interested in all of them. The ones we are interested in are the k -cutsets containing k or $k - 1$ mandatory edges in the graph. In addition, there are at most n mandatory edges in any TSP solution.

In this paper, we show that it is sufficient to study a set of k -cutsets lower than or equal to n in order to obtain a complete filtering. Moreover, we introduce a linear time algorithm for the k -cutset constraint for any k .

This article is organized as follows: first, we recall some concepts of graph theory. Then, we introduce the TSP in CP with the k -cutset constraint. Next, we define a new linear time algorithm for the k -cutset constraint. Finally, we discuss some experiments and we conclude.

2 Preliminaries

2.1 Definitions

The definitions about graph theory are taken from Tarjan's book [17].

A **directed graph** or **digraph** $G = (X, U)$ consists of a **node set** X and an **arc set** U , where every arc (x_i, x_j) is an ordered pair of distinct nodes. We note $X(G)$ the set of nodes of G such that $n = |X(G)|$ and $U(G)$ the set of arcs of G such that $m = |U(G)|$. In addition, $U(i)$ is the set of adjacent edges of i . The **cost** of an arc is a value associated with the arc. An **undirected graph** is a digraph such that for each arc $(x_i, x_j) \in U$, $(x_i, x_j) = (x_j, x_i)$. A **multigraph** is a digraph such that it can exist arcs that are not unique. If $G_1 = (X_1, U_1)$ and $G_2 = (X_2, U_2)$ are graphs, both undirected or both directed, G_1 is a **subgraph** of G_2 if $X_1 \subseteq X_2$ and $U_1 \subseteq U_2$. A **path** from node x_1 to node x_t in G is a list of nodes $[x_1, \dots, x_t]$ such that (x_i, x_{i+1}) is an arc for $i \in [1..k-1]$. The path **contains** node x_i for $i \in [1..k]$ and arc (x_i, x_{i+1}) for $i \in [1..k-1]$. The path is **simple** if all its nodes are distinct. The path is a **cycle** if $k > 1$ and $x_1 = x_t$. A cycle is **Hamiltonian** if $[x_1, \dots, x_{k-1}]$ is a simple path and contains every node of X . The **cost** of a path p , denoted by $w(p)$, is the sum of the costs of the arcs contained in p . For a graph G , a solution to the **traveling salesman problem (TSP)** in G is a Hamiltonian cycle $HC \in G$ minimizing $w(HC)$. An undirected graph G is **connected** if there is a path between each pair of nodes, otherwise it is **disconnected**. The maximum connected subgraphs of G are its **connected components**. A **k -edge-connected graph** is a graph in which there is no edges set of cardinality strictly less than k disconnecting the graph. A **tree** is a connected graph without a cycle. A tree $T = (X', U')$ is a **spanning tree** of G if $X' = X$ and $U' \subseteq U$. The U' edges are the **tree edges** T and the $U - U'$ edges are the **non-tree edges** T . A **minimum spanning tree** $T = (X', U')$ is a spanning tree minimizing the cost of the tree edges. A **bridge** is an edge such that its removal increases the number of connected components. A partition (S, T) of the nodes of G such that $S \subseteq X$ and $T = X - S$ is a **cut**. The set of edges $(x_i, x_j) \in U$ having $x_i \in S$ and $x_j \in T$ is the **cutset** of the (S, T) cut. A **k -cutset** is a cutset of cardinality k .

2.2 TSP in CP

The current best CP method solving the TSP is a combination of the Weighted Circuit Constraint (WCC) [2] and the structural constraint k -cutset [10]. The WCC is mainly based on the 1-tree Lagrangian Relaxation (LR) of Held and Karp [8, 9]. Intuitively, the LR derives a lower bound of the TSP (here, the 1-tree) until a solution of the TSP is found. A 1-tree is a minimum spanning tree in $G = (X - \{x\}, U)$ such that $x \in X$ is connected by its two nearest neighbors to the minimum spanning tree. Thus, a 1-tree covers the whole graph with n edges and a single cycle. In addition, if the 1-tree satisfies the degree constraint (each node of the 1-tree has exactly two neighbors), then the 1-tree is an optimal solution of the TSP. Therefore, the goal is to minimize the number of nodes that violate the degree constraint in the 1-tree. To do so, this constraint is integrated into the objective function and a Lagrangian multiplier π_i is associated to each node i . Let d_i be the degree of the node i in the 1-tree. For each node i of the graph, if $d_i < 2$, then π_i is decreased. Otherwise, if $d_i > 2$, then π_i is increased. Next, the edge cost $w((i, j))$ is modified such that $w'((i, j))$ is the modified cost and $w'((i, j)) = w((i, j)) + \pi_i + \pi_j$. Finally, we obtain an optimal solution of the TSP by computing a succession of 1-trees and modifying the edge costs.

However, experiments shown a very slow convergence toward the optimal solution. Thus, the WCC integrates the following filtering algorithms based on the costs:

- If an edge e does not belong to any 1-tree with cost smaller than a given upper bound, then e can be safely deleted.
- If an edge e belongs to all 1-trees with cost smaller than a given upper bound, then e is mandatory.

In addition, the WCC integrates a structural constraint imposing that each node has exactly two neighbors, i.e. the degree constraint.

Next, for each cutset of size k , the k -cutset constraint imposes that an even number of edges is mandatory. In practice, the study is limited to $k \leq 3$ because the given algorithm has a complexity growing with k . In addition, the interaction of the filtering algorithms and the convergence of the Lagrangian relaxation is not straightforward. Thus, Isoart and Régim [11] introduced an adaptive method in order to improve overall solving times.

About the search strategy, it consists in making a binary search where a left branch is an edge assignment and a right branch is an edge removal. More precisely, we use the search strategy LCFfirst of Fages et al. [5] which is an interpretation of Last Conflict heuristics [7, 12] for graph variables. It selects one edge in the graph according to a heuristic and keeps branching on one extremity of this edge until the extremity is exhausted. Note that it keeps branching even if a backtrack occurs. Thus, it is a highly dynamic search strategy that learns from previous choices. Moreover, most of the search strategies are much more efficient (up to an order of magnitude) when LCFfirst is used. In practice, we observe that using LCFfirst strongly interferes with the Lagrangian relaxation and filtering algorithms.

In addition, the WCC uses a single undirected graph variable $G = (X, M, O)$ where all nodes are mandatory. Without loss of generality, we note O the set of optional edges, M the set of mandatory edges such that $O \cup M = U$ and $O \cap M = \emptyset$. In addition, M is a growing set and O is a shrinking set. Since we search for a solution of the TSP, when a solution is found, $|M| = n$ and $O = \emptyset$.

2.3 The k -cutset constraint

We previously said that the purpose of the k -cutset constraint is to ensure for each cutset in $G = (X, M, O)$ that a strictly positive and even number of edges are mandatory in any solution. To do so, Isoart and Régim [10] have shown the following proposition:

► **Proposition 1.** *Given K a k -cutset, then any Hamiltonian cycle C contains an even and strictly positive number of edges from K .*

Since G contains mandatory and optional edges, a k -cutset of G can be partitioned into two disjoint subsets of O and M . Therefore, we note a k -cutset $K = (M', O')$ of G such that $M' \subset M$ and $O' \subset O$.

► **Definition 2.** *For each k -cutset $K = (M', O')$, the k -cutset constraint ensure that $|M'| + |O'| \geq 2$ and $|M'|$ is even if $O' = \emptyset$.*

From Definition 2, we can therefore define the following consistency checks:

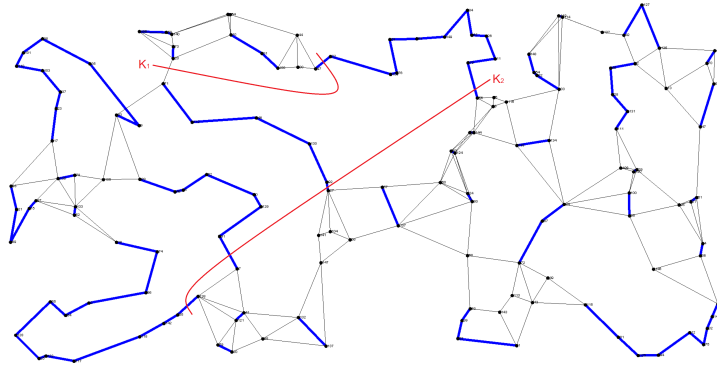
► **Corollary 3.** *If there is a k -cutset in G such that $k < 2$, then there is no solution for $TSP(G)$.*

► **Corollary 4.** *If there is a k -cutset in G containing k mandatory edges such that k is odd, then there is no solution for $TSP(G)$.*

In addition, we can define from Definition 2 the following filtering rules:

- **Corollary 5.** *Given a 2-cutset $K = (M', O')$ in G . Then, the edges of O' must become mandatory ($M \leftarrow M + O'$).*
- **Corollary 6.** *If there is a k -cutset K in G containing $k - 1$ mandatory edges such that k is odd, then the non-mandatory edge e of K can be safely deleted ($O \leftarrow O - \{e\}$).*
- **Corollary 7.** *If there is a k -cutset K in G containing $k - 1$ mandatory edges such that k is even, then the non-mandatory edge e of K must become mandatory ($M \leftarrow M + \{e\}$).*

Therefore, Corollary 3 and 4 allow checking the consistency, Corollary 5 and 7 allow finding mandatory edges, Corollary 6 allows removing edges.



■ **Figure 2** Representation of the graph from Figure 1 with mandatory edges (blue) and optional edges (dark). The two k -cutsets K_1 and K_2 are displayed in red.

For instance, we can deduce in Figure 2 that for the 2-cutset K_1 , all the K_1 edges become mandatory by Corollary 5 or 7. Moreover, the 4-cutset K_2 is valid because the k -cutset constraint is consistent (K_2 contains only mandatory edges and $|K_2| = 4$ is even).

- **Definition 8.** *A k -cutset $K = (M', O')$ is failing if $|M'| = k$ and k is odd.*
- **Definition 9.** *A k -cutset $K = (M', O')$ is prunable if $k > 1$ and $|M'| = k - 1$ and $|O'| = 1$.*

In order to find failing and prunable k -cutsets, Isoart and Régin [10] have developed an algorithm in $O(n(n + m))$. It finds all the k -cutsets such that $k \leq 2$ and all the 3-cutsets $K = (M', O')$ such that $|M'| > 0$. For $k = 1$ and $k = 2$, they use the non-trivial Tsin's algorithm [18] finding all cutsets of size smaller than or equal to 2 in a graph using a DFS in $O(n + m)$. For 3-cutsets $K = (M', O')$ such that $|M'| > 0$ in $G = (X, U)$, the main idea is the following: for each mandatory edge e_m , we look for the 2-cutsets in $G = (X, M - \{e_m\}, O)$. Since there are at most n mandatory edges in a TSP solution, this leads to a time complexity in $O(n(n + m))$. In addition, they give some practical improvements greatly reducing the number of considered mandatory edges e_m . For instance, they suggest using a 2-edge-connected subgraph of G minimizing the number of mandatory edges since all the 3-cutsets have at least 2 edges in this subgraph. Thus, it leads to the following algorithm: for each mandatory edge e_m in the 2-edge-connected subgraph, we look for the 2-cutsets in $G = (X, M - \{e_m\}, O)$.

Therefore, Corollary 3 and 5 are checked with Tsin's algorithm in $O(n + m)$. Corollary 4, 6 and 7 are checked for $k \leq 3$ in $O(n(n + m))$ with the algorithm described above. Finally, they have shown that the use of the k -cutset constraint allows reducing the number of backtracks by an order of magnitude with static strategies. Moreover, a gain of about a factor of 2 in solving times is obtained.

In the next section, we will introduce an algorithm enforcing the k -cutset constraint (i.e. checking Corollary 3, 4, 5, 6 and 7) for all k in $O(n + m)$.

3 A linear time algorithm

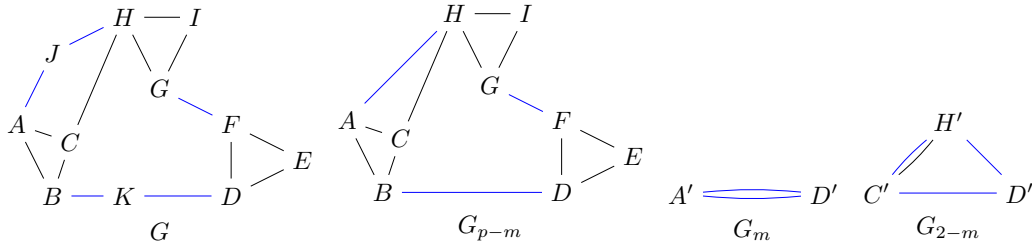
► **Definition 10.** A **mandatory path** of G is a path $p = [x_1, \dots, x_k]$ in G such that for each $i \in [1, k - 1]$, the edge (x_i, x_{i+1}) is mandatory.

► **Definition 11.** A **path-merged graph** G_{p-m} of G is the graph G such that for each mandatory path $p = [x_1, \dots, x_k]$ of G and $k > 2$, the nodes from x_2 to x_{k-1} are removed and a mandatory edge (x_1, x_k) is added.

► **Definition 12.** Given X' a set of nodes. The **merge** of X' is a mapping from all nodes of X' to a single node.

► **Definition 13.** Given $G_{p-m} = (X_{p-m}, M_{p-m}, O)$ a path-merged graph. A **merged graph** G_m of G_{p-m} is the multigraph G_{p-m} such that each connected component of the subgraph $G_{opt} = (X_{p-m}, \emptyset, O)$ of G_{p-m} are merged.

► **Definition 14.** Given $G_{p-m} = (X_{p-m}, M_{p-m}, O)$ a path-merged graph. A **2-merged graph** G_{2-m} of G_{p-m} is the multigraph G_{p-m} such that each 2-edge-connected component of the subgraph $G_{opt} = (X_{p-m}, \emptyset, O)$ of G_{p-m} are merged.



■ **Figure 3** Given G a 2-edge-connected graph. G_{p-m} is the path-merged graph of G such that the mandatory paths are $p_1 = [A, J, H]$ and $p_2 = [B, K, D]$. G_m is the merged graph of G_{p-m} . G_{2-m} is the 2-merged graph of G_{p-m} .

For instance, Figure 3 shows an example of Definition 11, 13 and 14.

Without loss of generality, we will consider that G is connected. In addition, we will use $G_{p-m} = (X_{p-m}, M_{p-m}, O)$ the path merged graph of G , $G_m = (X_m, M_m, \emptyset)$ the merged graph of G_m and $G_{opt} = (X_{p-m}, \emptyset, O)$ the subgraph G_{p-m} containing only optional edges. Each removed nodes of G in G_{p-m} are connected with exactly two mandatory edges (thanks to the degree constraint of the WCC) and each path is replaced by a single mandatory edge. Therefore, we will often consider G_{p-m} instead of G .

3.1 Consistency Check

In order to determine whether G is consistent with the k -cutset constraint, we must check Corollary 3 and Corollary 4. We can check Corollary 3 with Tarjan's algorithm [16]. Using a DFS, it finds all the bridges of a graph (i.e. the 1-cutsets) in $O(n + m)$. Checking Corollary 4 requires to check for the existence of a failing k -cutset. We will show that it can be done in linear time with Proposition 16.

► **Definition 15.** Given X' a set of nodes. We note $M^+(X') = \{(i, j) | (i, j) \in M, i \in X', j \notin X'\}$ the set of outgoing mandatory edges of X' in M .

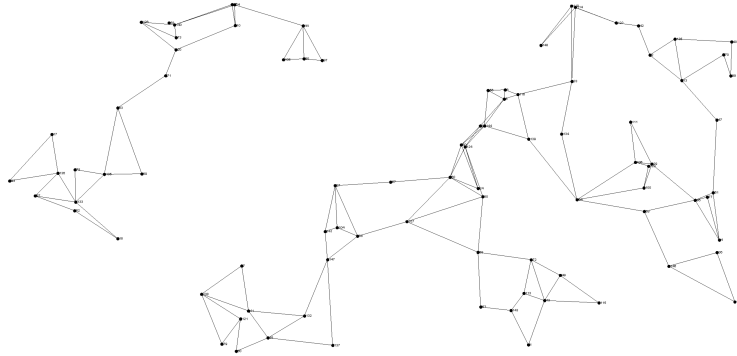
► **Proposition 16.** There is no failing k -cutset in G_{p-m} if and only if there is no connected component X' in G_{opt} such that $|M^+(X')|$ is odd.

Proof. We note (i) there is no failing k -cutset in G_{p-m} and (ii) there is no connected component X' in G_{opt} such that $|M^+(X')|$ is odd.

(i) \Rightarrow (ii) By definition, if there is no failing k -cutset in G , then there is no connected component X' in G_{opt} such that $|M^+(X')|$ is odd.

(i) \Leftarrow (ii) If a non-empty k -cutset K cuts a connected component of G_{opt} , then K contains optional edges since G_{opt} is the graph of optional edges. Therefore, a k -cutset containing only mandatory edges cannot cut a connected component of G_{opt} . Then, the failing k -cutsets are obtained by partitioning the connected components of G_{opt} in G_{p-m} , i.e. all the k -cutsets of the merged graph.

For each $S \subset X_m$, the cutset size of the cut $(S, X_m - S)$ can be computed as follows: (1) make the sum of the number of adjacent edges of each node of S , then (2) subtract twice the number of edges (i, j) such that i and j belong to S (an edge connecting two nodes of S is counted twice in (1)). Since we consider that (ii) is true, the sum obtained by (1) is even. (2) subtract an even number to the sum obtained by (1). Therefore, the cutset size is even and there is no failing k -cutset in G_{p-m} . ◀



■ **Figure 4** Representation of G_{opt} such that G is the graph of Figure 2.

For instance, in Figure 4, we notice that there are two connected components connected by 4 mandatory paths in Figure 2, so the k -cutset constraint is consistent with the graph of Figure 2. In addition, if we consider G_m of Figure 3, there is no failing k -cutset and each node has an even number of adjacent edges.

Then, we can describe an algorithm. First, compute the connected components of G_{opt} . Then, for each mandatory edge of M having its two endpoints in two different connected components of G_{opt} , we increase the number of mandatory outgoing edges for these connected components. Finally, we iterate on the connected components. If there is a connected component with an odd number of mandatory outgoing edges, then there is a failing k -cutset in G . The computation of the connected components of G_{opt} can be done in $O(n + m)$ with a DFS. The iteration over the mandatory edges of M can be done in $O(n)$. The check of the number mandatory outgoing edges for the connected components can be done in $O(n)$. Thus, we can test the consistency of the k -cutset constraint in $O(n + m)$.

3.2 Pruning

Corollary 5, 6 and 7 define filtering rules for the k -cutset constraint. First, Corollary 5 can be enforced with Tsin's algorithm [18]. It performs a single DFS in order to find all the 2-cutsets in a given graph, so it has a time complexity in $O(n + m)$.

In order to enforce Corollary 6 and 7, we need a method finding all k -cutsets having exactly $k - 1$ mandatory edges, i.e. the prunable k -cutsets. Given a prunable k -cutset $K = (M', O')$ of G . If M' is removed, then the edge of O' is a bridge of G since $|O'| = 1$. Formally, we define it in Proposition 17. We will exploit those bridges in order to enforce Corollary 6 and 7.

► **Proposition 17.** *If $K = (M', O')$ is a prunable k -cutset of G , then the edge of O' is a bridge in $G' = (X, M - M', O)$.*

Proof. A prunable k -cutset $K = (M', O')$ contains exactly $k - 1$ mandatory edges and 1 optional edge. Thus, removing the $k - 1$ mandatory edges in G transform K in a 1-cutset, i.e. in a bridge. ◀

► **Corollary 18.** *If $K = (M', O')$ is a prunable k -cutset of G , then the edge of O' is a bridge in G_{opt} .*

Proof. For each prunable k -cutset $K = (M', O')$, $M' \subseteq M$. Thus, from Proposition 17, the edge of O' is a bridge in $G' = (X, \emptyset, O)$. Therefore, it is a bridge in $G_{opt} = (X_{p-m}, \emptyset, O)$. ◀

From Corollary 18, we find the edges belonging to some prunable k -cutsets in G by searching for bridges in G_{opt} . It can be done with Tarjan's DFS algorithm [16] in $O(n + m)$. Next, we must determine for each bridge whether it should be deleted or become mandatory. We therefore need to retrieve the set of prunable k -cutsets that contain each bridge.

Without loss of generality, we will consider that G is 2-edge-connected, i.e. G is connected and bridgeless. Note that it can be checked in $O(n + m)$ with Tarjan's algorithm [16]. In addition, we note $X_i(G')$ the connected component of G' containing the node i .

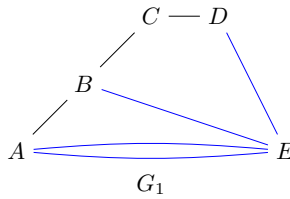
► **Proposition 19.** *Given $e \in O$ a bridge in the connected component X' of G_{opt} connecting $(X_1, X' - X_1)$. Then, the k -cutset $K = (M^+(X_1), \{e\})$ is a prunable k -cutset of G .*

Proof. In order to disconnect X_1 in G , the edges having exactly one end in X_1 must be removed, i.e. the k -cutset $K = (M', O')$ of $(X_1, X - X_1)$. It means $M' = M^+(X_1)$ and $O' = O^+(X_1)$. Since $e \in O$ is the bridge in X' of G_{opt} connecting $(X_1, X' - X_1)$, $O' = \{e\}$. Otherwise, e is not a bridge in G_{opt} . Finally, G is 2-edge connected, then $|M'| > 0$. Thus, K is a prunable k -cutset of G . ◀

► **Proposition 20.** *Given $e \in O$ a bridge in G_{opt} and a prunable k -cutset $K' = (M', \{e\})$. If there are no failing k -cutsets in G , then there are no prunable k -cutsets K'' such that the pruning of e with K' and K'' is different.*

Proof. For any k -cutset $K'' = (M'', \{e\})$ such that $M' \neq M''$, we can build a k -cutset $K''' = (M' \cup M'', \emptyset)$ containing only mandatory edges. In addition, if K''' is not a cutset, then e is not a bridge for M' or M'' . If K''' is not a failing k -cutset, then $M' \cup M''$ is even. Therefore, M' and M'' are either even or odd. Thus, if there are no failing k -cutsets in G , then there are no prunable k -cutsets K'' such that the pruning of e with K' and K'' is different. ◀

From Proposition 19 and 20, we can describe a first algorithm: for each bridge of the connected component X' of G_{opt} connecting $(X_1, X' - X_1)$, count the number of mandatory edges having one end in X_1 and the other in $X - X_1$. If there is an even number of mandatory edges, then delete e . Otherwise, add e to the mandatory edges. Thus, for each bridge, we parse at most all the mandatory edges. There is at most $n - 1$ bridge in a graph and at most n mandatory edges. Therefore, this algorithm finds all the prunable k -cutsets in $O(n^2)$. Next, we will show how to improve this algorithm in order to obtain a linear time complexity. However, note that this algorithm is already much better than the one of the state of the art [10] because we find the k -cutsets for all k with a better time complexity.



■ **Figure 5** G_1 represents the 2-merged graph of the path-merged graph of Figure 2. Blue edges are mandatory paths and dark edges are bridges.

In Figure 5, the 2-merged graph of Figure 2, we notice that some prunable k -cutsets are much simpler than others to find. Indeed, for (A, B) there are two prunable k -cutsets: $K_1 = (M_1, \{(A, B)\})$ where $M_1 = \{(A, E), (A, E)\}$ and $K_2 = (M_2, \{(A, B)\})$ where $M_2 = \{(B, E), (D, E)\}$. In order to find M_1 , we can simply search for the mandatory edges having one end in A and the other in $\{B, C, D, E\}$. In order to find M_2 , we have to search for the mandatory edges having one end in $\{B, C, D\}$ and the other in $\{A, E\}$. The difference between M_1 and M_2 is that we can simply find M_1 by considering the mandatory edges with exactly one end in a single component. Thus, if for each bridge there is such a prunable k -cutset, then we simply have to count the number of outgoing mandatory edges of each 2-edge-connected components of G_{opt} in G . It leads to an algorithm with a linear time complexity. Unfortunately, this algorithm may not handle all the prunable k -cutsets. For instance, there are two prunable k -cutsets containing (B, C) : $K_3 = (M_3, \{(B, C)\})$ such that $M_3 = \{(D, E)\}$ and $K_4 = (M_4, \{(B, C)\})$ such that $M_4 = \{(B, E), (A, E), (A, E)\}$. In that case, we cannot simply look at the neighbors of B or C to find the prunable k -cutsets containing (B, C) . Indeed, B and C have more than one optional neighbor. It means that the bridge (B, C) disconnect G_{opt} in $(\{A, B\}, \{C, D\})$. Therefore, both connected components are not 2-edge-connected. Thus, we show in Corollary 21 that for a bridge e connecting (X_1, X_2) in G_{opt} , if X_1 (resp. X_2) is 2-edge-connected, then it exists a prunable k -cutset containing e and all the mandatory edges having exactly one end in X_1 (resp. X_2).

► **Corollary 21.** *Given X' a connected component of G_{opt} . If $e \in O$ is a bridge in X' of G_{opt} connecting $(X_1, X' - X_1)$ such that X_1 is 2-edge-connected, then there is a prunable k -cutset $K = (M^+(X_1), \{e\})$.*

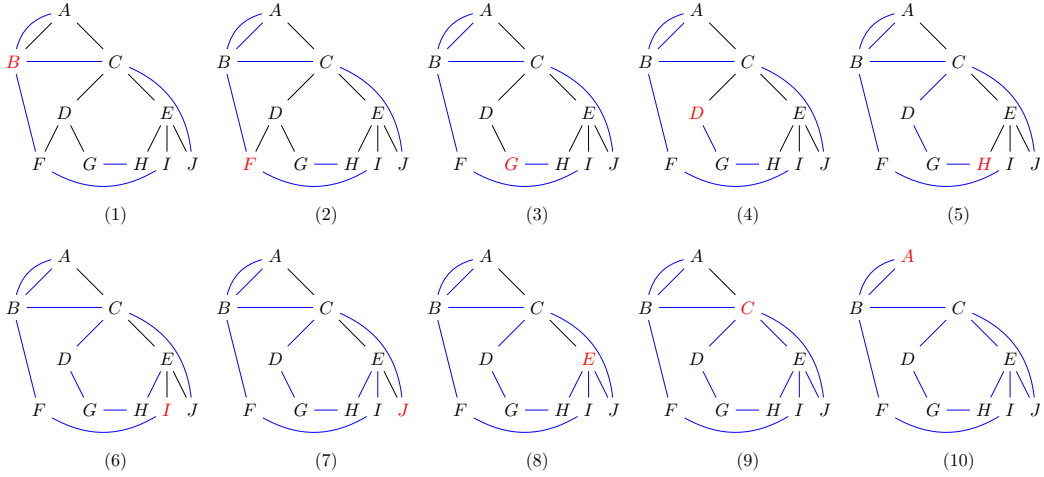
Proof. Immediate from Proposition 19. ◀

The advantage of Corollary 21 over Proposition 19 is that the 2-edge-connected component X_1 of Corollary 21 are disjoint nodes sets. Thus, parsing the neighbors of the 2-edge-connected components consider at most twice the total number of mandatory edges whereas Proposition 19 can reconsider for each component all the mandatory edges of the 2-merged graph. We will use this idea in order to obtain a linear time algorithm.

In Figure 5, all the k -cutsets formed by the neighborhood of a component are: $K_A = (\{(A, E), (A, E)\}, \{(A, B)\})$, $K_B = (\{(B, E)\}, \{(B, A), (B, C)\})$, $K_C = (\emptyset, \{(C, B), (C, D)\})$ and $K_D = (\{(D, E)\}, \{(D, C)\})$. Among them, K_A and K_D are prunable k -cutsets, then we can immediately deduce for K_1 that (A, B) is deleted (by Corollary 6) and for K_D that (D, C) becomes mandatory (by Corollary 5 or 7). If we update the k -cutsets we have: $K_A = (\{(A, E), (A, E)\}, \emptyset)$, $K_B = (\{(B, E)\}, \{(B, C)\})$, $K_C = (\{(C, D)\}, \{(C, B)\})$ and $K_D = (\{(D, E), (D, C)\}, \emptyset)$. We notice that both K_B and K_C become prunable k -cutsets. Thus, (C, B) becomes mandatory for both K_B and K_C (by Corollary 5 or 7). Finally, all bridges of G_1 have been solved.

Note that the subgraph of optional edges of the 2-merged graph can be more sophisticated than a simple path of bridges edges: it can be a tree. However, it cannot exist a cycle in this subgraph since the 2-edges-connected components are merged in the 2-merged graph. For example, (1) of Figure 6 shows a 2-merged graph such that the subgraph of optional edges is not a single path. We note that (1) is rooted in A and each node of the set of nodes $S = \{B, F, G, H, I, J\}$ has exactly one optional neighbor. Thus, we can start by applying Corollary 21 on S , i.e. the leaves of the tree. Leaves are always valid candidates for Corollary 21 because they have no optional child and a single optional parent. Then, either there are no more leaves and therefore there are no more prunable k -cutsets or there are leaves and we can apply Corollary 21 to these leaves. Finally, we suggest to recursively apply this process until there are no more leaves in the tree. A sketch of the algorithm is:

- Find bridges of G with the DFS-based Tarjan's algorithm.
- Mark the 2-edge-connected components in postorder, i.e. the order of a node is set when it is backtracked in the DFS.
- For each mandatory edge (i, j) , increase the number of outgoing mandatory edges of the 2-edge-connected component of i and j .
- Iterate over the 2-edge-connected components C_i of the 2-merged graph with the defined postorder and prune the bridge connected to C_i .



■ **Figure 6** Example of an execution of the k -cutset pruning algorithm on the graph (1). Blue edges are mandatory paths, dark edges are bridges. The red node is the current 2-edge-connected component of the algorithm step.

For instance, Figure 6 shows an execution of the algorithm in a 2-merged graph. Tarjan's algorithm allows us to create the 2-merged graph where black edges are bridges and blue edges are mandatory edges. In this tree, the postorder traversal is $\{B, F, G, D, H, I, J, E, C, A\}$.

Note that the postorder is used to guarantee that for each node considered in the execution of the algorithm, all its children have already been pruned. From (1) to (10), we show the iteration over the 2-edge-connected components C_i marked as red nodes. Finally, finding the bridges is performed in $O(n + m)$, parsing the mandatory edges is performed in $O(n)$ and parsing the 2-edge-connected components is performed in $O(n)$. Thus, our algorithm finds the prunable k -cutsets for all k in $O(n + m)$. Algorithm 1 is a possible implementation.

It takes as input $G = (X, M, O)$. We note CC the set of connected components in G_{opt} and $2CC$ the set of 2-edge-connected components in G_{opt} such that $2CC_i$ is the 2-edge-connected component containing the node i . First, we start by searching the bridges with Tarjan's algorithm based on a DFS in order to build CC and $2CC$. Within the DFS, $2CC$ is constructed with respect to the postorder tree traversal. Thus, iterating on $2CC$, we obtain the postorder tree traversal of the 2-merged graph of G . In Tarjan's algorithm, a 2-edge connected component C is found when a bridge e is found. Thus, we associate e with C by setting $C.bridge = e$. For instance, in (1) of Figure 6, each node is a 2-edge-connected component knowing its parent, i.e. a bridge. Thus, the only node having no parent is the root node and all the other nodes have a single parent that is an optional edge.

Secondly, we count the number of outgoing mandatory edges for each connected component of CC and for each 2-edge-connected component of $2CC$. We then consider the connected components $C \in CC$ such that $|C| > 1$. If $|C| = 1$, then C has two adjacent mandatory edges and C contains a single node. Thus, considering C such that $|C| > 1$ is equivalent of considering the path-merged graph of G . In addition, we know that each node i of C has at most one adjacent mandatory edge (i, j) . Therefore, $|M(i)| \leq 1$. Otherwise, the node i would not belong to C . We note $M(i).firstEdge()$ the first edge in the list of the adjacent mandatory edges of i . Since we are looking for the outgoing mandatory edges, we only consider the nodes with $M(i) = 1$. If j is an outgoing edge of C , then we increase the number of outgoing mandatory edges of C noted $M^+(C)$. If i and j do not belong to the same 2-edge-connected component, then we increase the number of outgoing mandatory edges of $2CC_i$ noted $M^+(2CC_i)$ ($M^+(2CC_j)$ is increased when the node j is considered by the foreach). Note that we can check if the node i belongs to the nodes set C' with the function $C'.isIn(i)$. In (1) of Figure 6, there is a single connected component C so $M^+(C) = 0$. However, there are 10 2-edge-connected components ($\{A, B, C, D, E, F, G, H, I, J\}$). For example, $M^+(2CC_A) = 1$ and $M^+(2CC_B) = 3$. Afterwards, we check the consistency. If there is $C \in CC$ such that $M^+(C)$ is odd, then there is a failing k -cutset and we return False. In (1) of Figure 6, there is no $C \in CC$ such that $M^+(C)$ is odd, so (1) is consistent with the k -cutset constraint.

Thirdly, we perform the pruning step. We iterate on all the 2-edge connected components C of $2CC$. We note (i, j) the bridge associated to C . If $M^+(C)$ is odd, then (i, j) becomes mandatory (i.e. it is added to M) and $M^+(2CC_i)$ and $M^+(2CC_j)$ are increased by 1. Whether $M^+(C)$ is even or odd, (i, j) is removed from O . Indeed, if the edge becomes mandatory, then it must not be in the set of the optional edges. For instance, in (1) of Figure 6, we consider the node B and the bridge (B, A) . Note that in (1), $M^+(2CC_A) = 1$ and $M^+(2CC_B) = 3$. Since $M^+(2CC_B)$ is odd, (B, A) becomes mandatory and $M^+(2CC_A) = 2$ and $M^+(2CC_B) = 4$. Next, we consider the node F and its bridge (F, D) . $M^+(2CC_F) = 2$ so (F, D) is removed and $M^+(2CC_F)$ remains unchanged. Then, we repeat this process until (10). We note that the nodes are chosen in the postorder.

■ **Algorithm 1** Perform the consistency check and the pruning of k -cutset constraint.

```

1 k-cutset ( $G = (X, M, O)$ )
  Input: A graph  $G=(X,M,O)$ .
  Output: A boolean specifying whether  $G$  contains a failing  $k$ -cutset
  //  $CC$ : connected components of  $G - M$  ( $G_{opt}$ )
  //  $2CC$ : postorder 2-edge-connected components of  $G - M$  ( $G_{opt}$ )
2 computeBridgesDFS( $G - M, CC, 2CC$ ) ;
3 foreach connected components  $C \in CC$  do
4   if  $|C| > 1$  then
5     foreach node  $i \in C$  do
6       if  $|M(i)| = 1$  then
7          $(i, j) \leftarrow M(i).firstEdge()$ ;
8         if not  $C.isIn(j)$  then
9            $M^+(C) \leftarrow M^+(C) + 1$ ;
10        if not  $2CC_i.isIn(j)$  then
11           $M^+(2CC_i) \leftarrow M^+(2CC_i) + 1$ ;
12
13      // Consistency check
14      foreach connected components  $C \in CC$  do
15        if  $M^+(C)$  is odd then return False ;
16
17      // Pruning
18      foreach 2-edge-connected components  $C \in 2CC$  do
19        if  $C.bridge \neq nil$  then
20           $(i, j) \leftarrow C.bridge$ ;
21          if  $M^+(C)$  is odd then
22             $M^+(2CC_i) \leftarrow M^+(2CC_i) + 1$ ;
23             $M^+(2CC_j) \leftarrow M^+(2CC_j) + 1$ ;
24             $M \leftarrow M + (i, j)$ ;
25             $O \leftarrow O - (i, j)$ ;
26
27      return True;

```

4 Experiments

The algorithms have been implemented in Java 11 in a locally developed constraint programming solver. The experiments were performed on Clear Linux with an Intel Xeon E5-2696v2 and 64 GB of RAM. The instances are from the TSPLib [14], a library of reference graphs for the TSP. We rerun experiments ran in Isoart and Régim [10] and exclude instances solved in less than two seconds. We also include some harder instances. The name of each instance is suffixed by its number of nodes. In our implementation, the TSP is modeled by the WCC using the CP-based LR configuration introduced in Isoart and Régim [11]. We note “state of the art” the TSP model with the state of the art k -cutset algorithm and “linear full k -cutset” the TSP model with our algorithm. The search strategy used is LCFfirst with the heuristic minDeltaDeg [5] which is also the state of the art. Given $e = (i, j)$ an edge, minDeltaDeg selects the edge with the minimum difference between the sum of the number of optional neighbors of i and j and the sum of the number of mandatory neighbors of i and j . Thus, we

compare linear full k -cutset and the state of the art. We give the number of backtracks (#bk) and the solving time in seconds in arrays for the k -cutset constraint with different search strategies. In addition, we set a timeout *t.o.* of 100,000 seconds. All considered instances are symmetric graphs.

■ **Table 1** General results comparing the state of the art and linear full k -cutset.

Instance	(1) State of the art		(2) Linear full k -cutset		Ratio (1)/(2)	
	time(s)	#bk	time(s)	#bk	time(s)	#bk
kroB100	5.4	4,816	3.6	3,308	1.5	1.5
kroE100	2.3	1,804	2.4	2,212	0.9	0.8
pr124	2.8	1,856	3.5	2,482	0.8	0.7
pr136	20.4	18,684	20.0	21,446	1.0	0.9
kroA150	6.1	4,164	4.1	2,798	1.5	1.5
kroB150	262.6	247,574	153.6	154,002	1.7	1.6
si175	288.5	301,102	280.8	358,676	1.0	0.8
rat195	38.5	24,274	37.8	27,512	1.0	0.9
d198	14.0	7,192	8.6	4,782	1.6	1.5
kroA200	401.0	237,806	323.7	200,392	1.2	1.2
kroB200	127.8	87,296	135.9	109,322	0.9	0.8
tsp225	121.5	65,002	139.3	89,946	0.9	0.7
gr229	227.0	166,378	139.8	114,434	1.6	1.5
gil262	5,230.2	2,254,728	2,970.7	1,711,410	1.8	1.3
pr264	4.7	690	4.9	642	0.9	1.1
a280	7.0	2,372	6.6	2,484	1.1	1.0
lin318	32.9	7,834	11.0	3,456	3.0	2.3
gr431	1,724.8	265,698	1,358.6	247,090	1.3	1.1
pcb442	15,081.5	4,130,580	16,490.1	5,555,756	0.9	0.7
d493	95,916.6	13,478,616	69,247.1	11,346,180	1.4	1.2
mean	5,975.8	1,065,423.3	4,567.1	997,916.5	1.3	1.1

Table 1 shows the solving times and backtrack numbers for the state of the art and the linear full k -cutset. A ratio column display both solving times and backtrack numbers gain for the linear full k -cutset. For most instances, we observe a gain in backtrack numbers and solving times. Otherwise, the results are quite close to the state of the art results. Indeed, we observe an average gain in solving time of 30% and 10% in backtracks. On average, the state of the art runs in 178 backtracks per seconds while linear full k -cutset runs in 219 backtracks per seconds. The low backtrack number gain suggests that most of the cutsets are in fact k -cutsets with $k \leq 3$, and therefore the state of the art algorithm already finds most of the cutsets.

Nevertheless, the TSP model includes a Lagrangian relaxation and the relation between the filtering algorithms and the Lagrangian relaxation is not clear [15, 11]. Moreover, the LCFirst minDeltaDeg search strategy is extremely dynamic. Thus, in order to have a better understanding of the impact of the linear full k -cutset, we compare it with the static search strategy maxCost, i.e. edges are selected by decreasing costs.

In Table 2, we observe a gain on all instances that are both solved by (1) and (2), the solving is therefore much more stable. In addition, we observe that the instance gil262 timeout in the state of the art. Excluding it, we obtain an average gain of 80% in solving time

■ **Table 2** General results of the static search strategy maxCost comparing the state of the art (1) with our algorithm (2).

Instance	(1) State of the art		(2) Linear full k -cutset		Ratio (1)/(2)	
	time(s)	#bk	time(s)	#bk	time(s)	#bk
kroB100	7.7	7,640	6.1	7,056	1.3	1.1
kroE100	10.6	11,328	6.3	6,136	1.7	1.8
pr124	1.3	316	1.1	294	1.1	1.1
pr136	101.5	86,860	73.4	86,094	1.4	1.0
kroA150	57.7	55,940	46.7	49,016	1.2	1.1
kroB150	408.1	344,440	314.2	299,728	1.3	1.1
si175	4,168.6	4,661,506	3,190.8	4,305,208	1.3	1.1
rat195	486.1	372,438	364.0	358,936	1.3	1.0
d198	71.9	52,618	50.2	41,432	1.4	1.3
kroA200	3,111.1	1,944,312	1,813.5	1,209,136	1.7	1.6
kroB200	491.3	333,440	376.7	303,318	1.3	1.1
tsp225	15,252.7	11,579,074	10,224.3	9,209,292	1.5	1.3
gr229	2,349.7	1,955,538	1,591.3	1,607,300	1.5	1.2
gil262	<i>t.o.</i>	<i>t.o.</i>	70,450.6	36,507,156	≥ 1.2	
pr264	7.5	902	6.7	748	1.1	1.2
a280	46.9	20,380	9.6	4,024	4.9	5.1
lin318	65.0	21,292	14.4	5,934	4.5	3.6

and 60% in backtracks. Thus, this shows that the impact of considering k -cutsets for any k is useful for the performance. In addition, these results suggest that LCFfirst minDeltaDeg fills a part of the lack of structural constraints in the TSP model in CP.

In Table 3, we study the size of the founded k -cutsets in linear full k -cutset with LCFfirst minDeltaDeg. We can observe that the mean size the founded k -cutsets is 3. It confirms the fact that the state of the art algorithm already finds a large part of the k -cutsets. However, larger k -cutsets exist: the average number of maximum size of the k -cutsets is 14.1. This is why we obtain a more interesting backtrack gain than the state of the art whereas the average k -cutset size is 3.

Finally, our linear full k -cutset algorithm is simple to implement and allows us to obtain an improvement of the solving times and the number of backtracks.

5 Conclusion

In this paper, we have introduced a new linear time algorithm checking the k -cutset constraint for any k . Experiments have shown that our algorithm leads to an improvement of solving times. Moreover, we have shown that on average most of the cutsets are of size 3 even if we found some much larger cutsets. We hope that other structural constraints will be integrated into the WCC: they make the CP competitive in the same way that Comb inequalities make the MIP efficient.

■ **Table 3** Comparison of mean and max k -cutsets size.

Instance	k -cutsets size	
	mean	max
kroB100	2.63	7
kroE100	2.72	7
pr124	2.50	10
pr136	2.57	14
kroA150	2.86	8
kroB150	2.21	11
si175	2.63	13
rat195	4.77	26
d198	2.40	13
kroA200	2.79	10
kroB200	2.96	13
tsp225	2.93	13
gr229	3.08	14
gil262	2.55	17
pr264	2.48	25
a280	3.31	9
lin318	5.47	14
gr431	2.45	16
pcb442	3.53	24
d493	2.43	26
mean	3.0	14.5

References

- 1 David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2006.
- 2 Pascal Benchimol, Willem-Jan Van Hoeve, Jean-Charles Régin, Louis-Martin Rousseau, and Michel Rueher. Improved filtering for weighted circuit constraints. *Constraints*, 17(3):205–233, 2012. URL: <https://hal.archives-ouvertes.fr/hal-01344070>.
- 3 Václav Chvátal. Edmonds polytopes and weakly hamiltonian graphs. *Math. Program.*, 5(1):29–40, 1973. doi:10.1007/BF01580109.
- 4 Jack Edmonds. Paths, Trees, and Flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965. doi:10.4153/CJM-1965-045-4.
- 5 Jean-Guillaume Fages, Xavier Lorca, and Louis-Martin Rousseau. The salesman and the tree: the importance of search in CP. *Constraints*, 21(2):145–162, 2016.
- 6 Martin Grötschel and Manfred Padberg. On the symmetric travelling salesman problem i: Inequalities. *Mathematical Programming*, 16:265–280, December 1979. doi:10.1007/BF01582116.
- 7 Robert Haralick and Gordon Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, January 1979.
- 8 Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.
- 9 Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees: Part ii. *Mathematical Programming*, 1(1):6–25, 1971.

- 10 Nicolas Isoart and Jean-Charles Régin. Integration of structural constraints into tsp models. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming*, pages 284–299, Cham, 2019. Springer International Publishing.
- 11 Nicolas Isoart and Jean-Charles Régin. Adaptive CP-Based Lagrangian Relaxation for TSP Solving. In Emmanuel Hebrard and Nysret Musliu, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 300–316, Cham, 2020. Springer International Publishing.
- 12 Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.
- 13 Adam N. Letchford and Andrea Lodi. Polynomial-Time Separation of Simple Comb Inequalities. In William J. Cook and Andreas S. Schulz, editors, *Integer Programming and Combinatorial Optimization*, pages 93–108, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
- 14 Gerhard Reinelt. TSPLIB – A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3(4):376–384, 1991.
- 15 Meinolf Sellmann. Theoretical Foundations of CP-Based Lagrangian Relaxation. In *Principles and Practice of Constraint Programming – CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 – October 1, 2004, Proceedings*, pages 634–647, 2004.
- 16 Robert E. Tarjan. A note on finding the bridges of a graph. *Inf. Process. Lett.*, 2:160–161, 1974.
- 17 Robert E. Tarjan. *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics, 1983.
- 18 Yung H. Tsin. Yet another optimal algorithm for 3-edge-connectivity. *Journal of Discrete Algorithms*, 7(1):130–146, 2009. Selected papers from the 1st International Workshop on Similarity Search and Applications (SISAP).

A k -Opt Based Constraint for the TSP

Nicolas Isoart ✉

Université Côte d’Azur, Nice, France

Jean-Charles Régim ✉

Université Côte d’Azur, Nice, France

Abstract

The LKH algorithm based on k -opt is an extremely efficient algorithm solving the TSP. Given a non-optimal tour in a graph, the idea of k -opt is to iteratively swap k edges of this tour in order to find a shorter tour. However, the optimality of a tour cannot be proved with this method. In that case, exact solving methods such as CP can be used. The CP model is based on a graph variable with mandatory and optional edges. Through branch-and-bound and filtering algorithms, the set of mandatory edges will be modified. In this paper, we introduce a new constraint to the CP model named mandatory Hamiltonian path constraint searching for k -opt in the mandatory Hamiltonian paths. Experiments have shown that the mandatory Hamiltonian path constraint allows us to gain on average a factor of 3 on the solving time. In addition, we have been able to solve some instances that remain unsolved with the state of the art CP solver with a 1 week time out.

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

Keywords and phrases TSP, k -opt, 1-tree, Constraint

Digital Object Identifier 10.4230/LIPIcs.CP.2021.30

Funding This work has been supported by the 3IA Côte d’Azur with the reference number ANR-19-P3IA-0002.

1 Introduction

The Traveling Salesman Problem (TSP) is a widely studied graph theory problem with a simple statement: find a minimum cost cycle in a graph visiting all nodes. Unfortunately, solving a TSP is not as easy as stating it: finding the optimal solution of the TSP is NP-Hard.

Heuristics allow one to find non-proved optimal solutions of the TSP in reasonable solving times. The most efficient heuristic solving the TSP is the Lin-Kernighan-Helsgaun (LKH) algorithm [17, 12]. It starts from a tour that is not optimal and iteratively improves the tour with one of the most popular tour improvement algorithms: the local search algorithm k -opt [18]. It consists in finding k edges in a given tour such that swapping them create a cheaper tour. Unfortunately, the k -opt algorithm has a time complexity in $O(n^k)$ such that n is the number of nodes in a graph. In order to obtain an efficient algorithm, they suggest many improvements such as using a variable k and not considering all the swaps of size k but only the most “promising” swaps.

Exact algorithms allow one to find optimal solutions of the TSP. In practice, they are usually much slower than heuristics because of the optimality proof. The most efficient method solving the “pure” TSP is the specialized solver Concorde [1] based on MIP methods. It is mainly based on the relaxation of the integrity and subtour constraints of the TSP model. In addition, the cutting plane method [5] is used in order to correct structural defects of the intermediate solutions obtained by this relaxation. It proceeds by iteratively generating constraints that are violated by the solution of the relaxed problem. Among them, there are the well-known Comb inequalities. However, no polynomial time algorithm is known at this time to detect whether a solution of the relaxed problem violates a Comb inequality. Therefore, many polynomial time algorithms have been developed in order



© Nicolas Isoart and Jean-Charles Régim;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 30; pp. 30:1–30:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to handle particular cases [6, 8, 4, 16]. In addition, Concorde embed other sophisticated techniques such as local cuts. Note that Concorde outperforms the other exact solving methods when considering large graphs. However, CP method is competitive with Concorde for small-medium size graphs [2]. In addition, a TSP is often combined with other constraints. For instance, precedence constraints, TSPTW where there is a time window to visit a node. For these problems, Concorde is not well suited whereas the CP method is a good candidate because it is more robust to side constraints. Nowadays, the most efficient method solving the TSP in CP is the Weighted Circuit Constraint (WCC) [2] in combination with the structural k -cutset constraint [13]. The optimization part of the WCC is based on the Lagrangian Relaxation (LR) of Held and Karp [10, 11]. The lower bound of the LR is computed by selecting a node x with its two lowest cost neighbors and a minimum spanning tree in the graph without x , *i.e.* it is a 1-tree. If a minimum 1-tree is found such that all its nodes have exactly two neighbors, then an optimal solution is obtained. Thus, the 1-tree is derived through the LR process until an optimal solution is obtained. However, optimally solving the TSP with a LR only can be extremely slow. Thus, the WCC integrates filtering algorithms based on the cost of the edges, the 1-tree cost and a degree constraint on the nodes. In addition, the k -cutset constraint is based on the graph structure. It considers the cutsets of the graph containing k mandatory edges and deduces structural filtering. In contrast to heuristic methods, the CP method does not improve a tour but builds an optimal tour. Indeed, the CP model imposes some edges through a branch and bound. Those edges, named mandatory edges, can form paths. Therefore, the purpose of the CP method is to find a tour going through these edges. However, finding an optimal solution can be impossible. For instance, it happens when a path is not itself optimal. Thus, it finds solutions that are suboptimal.

In this paper, we define the mandatory Hamiltonian path constraint that uses the k -opt algorithm on the mandatory paths. More precisely, let us define p , a path composed of mandatory edges going from s to t through a set of nodes X' . If p can be improved by another path p' going from s to t through X' , then p cannot belong to an optimal solution. In addition, we define a filtering algorithm removing edges: if a path can be improved when an edge is added to it, then it cannot exist an optimal solution simultaneously containing that path and that edge.

This article is organized as follows: first, we recall some concepts of graph theory. Then, we introduce the TSP in CP with the k -cutset constraint and the tour improvement algorithms. Next, we define the mandatory Hamiltonian path constraint and its incremental version. Finally, we discuss some experiments and we conclude.

2 Preliminaries

2.1 Definitions

The definitions of graph theory are taken from Tarjan's book [21].

A **directed graph** or **digraph** $G = (X, U)$ consists of a **node set** X and an **arc set** U , where every arc (x_i, x_j) is an ordered pair of distinct nodes. We note $X(G)$ the set of nodes of G such that $n = |X(G)|$ and $U(G)$ the set of arcs of G such that $m = |U(G)|$. In addition, $U(i)$ is the set of adjacent edges of i . The **cost** of an arc is a value associated with the arc. An **undirected graph** is a digraph such that for each arc $(x_i, x_j) \in U$, $(x_i, x_j) = (x_j, x_i)$. If $G_1 = (X_1, U_1)$ and $G_2 = (X_2, U_2)$ are graphs, both undirected or both directed, G_1 is a **subgraph** of G_2 if $X_1 \subseteq X_2$ and $U_1 \subseteq U_2$. A **path** from node x_1 to node x_t in G is a list of nodes $[x_1, \dots, x_t]$ such that (x_i, x_{i+1}) is an arc for $i \in [1..k-1]$. The path **contains** node

x_i for $i \in [1..k]$ and arc (x_i, x_{i+1}) for $i \in [1..k-1]$. The path is **simple** if all its nodes are distinct. The path is a **cycle** if $k > 1$ and $x_1 = x_k$. A cycle is **Hamiltonian** if $[x_1, \dots, x_{k-1}]$ is a simple path and contains every node of X . The **cost** of a path p , denoted by $w(p)$, is the sum of the costs of the arcs contained in p . For a graph G , a solution to the **traveling salesman problem (TSP)** in G is a Hamiltonian cycle $HC \in G$ minimizing $w(HC)$. An undirected graph G is **connected** if there is a path between each pair of nodes, otherwise it is **disconnected**. The maximum connected subgraphs of G are its **connected components**. A **tree** is a connected graph without a cycle. A tree $T = (X', U')$ is a **spanning tree** of G if $X' = X$ and $U' \subseteq U$. The U' edges are the **tree edges** T and the $U - U'$ edges are the **non-tree edges** T . A **minimum spanning tree** $T = (X', U')$ is a spanning tree minimizing the cost of the tree edges. A partition (S, T) of the nodes of G such that $S \subseteq X$ and $T = X - S$ is a **cut**. The set of edges $(x_i, x_j) \in U$ having $x_i \in S$ and $x_j \in T$ is the **cutset** of the (S, T) cut. A **k -cutset** is a cutset of cardinality k .

2.2 TSP in CP

The current best CP method solving the TSP is a combination of the Weighted Circuit Constraint (WCC) [2] and the structural constraint k -cutset [13]. The WCC is mainly based on the 1-tree Lagrangian Relaxation (LR) of Held and Karp [10, 11]. Intuitively, the LR derives a lower bound of the TSP (here, the 1-tree) until a solution of the TSP is found. A 1-tree is a minimum spanning tree in $G = (X - \{x\}, U)$ such that $x \in X$ is connected by its two nearest neighbors to the minimum spanning tree. Thus, a 1-tree covers the whole graph with n edges and a single cycle. In addition, if the 1-tree satisfies the degree constraint (each node of the 1-tree has exactly two neighbors), then the 1-tree is an optimal solution of the TSP. Therefore, the goal is to minimize the number of nodes that violate the degree constraint in the 1-tree. To do so, this constraint is integrated into the objective function and a Lagrangian multiplier π_i is associated to each node i . Let d_i be the degree of the node i in the 1-tree. For each node i of the graph, if $d_i < 2$, then π_i is decreased. Otherwise, if $d_i > 2$, then π_i is increased. Next, the edge cost $w((i, j))$ is modified such that $w'((i, j))$ is the modified cost and $w'((i, j)) = w((i, j)) + \pi_i + \pi_j$. Finally, we obtain an optimal solution of the TSP by computing a succession of 1-trees and modifying the edge costs.

However, experiments shown a very slow convergence toward the optimal solution. Thus, the WCC integrates the following filtering algorithms based on the costs:

- If an edge e does not belong to any 1-tree with cost smaller than a given upper bound, then e can be safely deleted.
- If an edge e belongs to all 1-trees with cost smaller than a given upper bound, then e is mandatory.

Moreover, the WCC integrates a structural constraint imposing that each node has exactly two neighbors (the degree constraint).

Next, for each cutset of size k , the k -cutset constraint imposes that an even number of edges is mandatory. In practice, the study is limited to $k \leq 3$ since the given algorithm has a complexity growing with k . In addition, the interaction of the filtering algorithms and the convergence of the Lagrangian relaxation is not straightforward. Thus, Isoart and Régin [14] introduced an adaptive method in order to improve the overall solving times.

About the search strategy, it consists in making a binary search where a left branch is an edge assignment and a right branch is an edge removal. More precisely, we use the search strategy LCFfirst of Fages et al. [7] which is an interpretation of Last Conflict heuristics [9, 15] for graph variables. It selects one edge in the graph according to a heuristic and keeps branching on one extremity of this edge until the extremity is exhausted. Note that it keeps

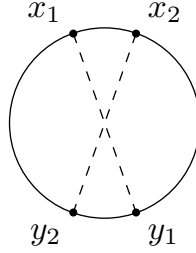
branching even if a backtrack occurs. Thus, it is a highly dynamic search strategy that learns from previous choices. Moreover, most of the search strategies are much more efficient (up to an order of magnitude) when LCFfirst is used. In practice, we observe that using LCFfirst strongly interferes with the Lagrangian relaxation and filtering algorithms.

In addition, the WCC uses a single undirected graph variable where all nodes are mandatory. Without loss of generality, we note O the set of optional edges, M the set of mandatory edges and D the set of deleted edges such that $O \cup M \cup D = U$, $O \cap M = \emptyset$, $O \cap D = \emptyset$ and $M \cap D = \emptyset$. Thus, the purpose of the CP is to find a TSP in the input graph $G_{init} = (X, M, O)$ such that M is a growing set and O is a shrinking set. When a solution is found, $|M| = n$ and $O = \emptyset$.

For the sake of clarity, we define $G_{solve} = (X, M', O')$ the current graph such that $M \subseteq M' \subseteq (O \cup M)$ and $O' \subseteq O$. In addition, we define G_{solve} the graph G_{init} modified by the search strategy and the filtering algorithms and $G_{mand} = (X, M', \emptyset)$ the graph of mandatory edges. If not specified, we will use these notations and data structures in the next sections.

2.3 Tour improvement algorithms

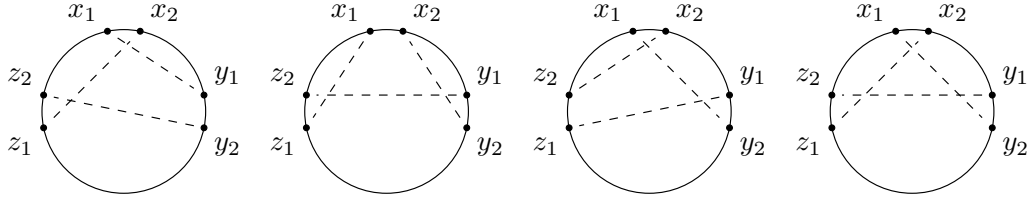
In order to find a TSP, a tour improvement algorithm takes as input a tour and iteratively tries to improve it. The most popular tour improvement algorithms are the local search algorithms 2-opt and 3-opt.



■ **Figure 1** An example of 2-opt. The circle represents a tour and the dashed lines are the suggested move for the pair of edges $((x_1, x_2), (y_1, y_2))$.

The idea of 2-opt is pretty simple. Given a tour T , for each pair of edges (e_1, e_2) in T , if replacing (e_1, e_2) by another pair of edges (e_3, e_4) of T leads to a connected and shorter tour, then we can replace (e_1, e_2) with (e_3, e_4) in T . We name such a replacing procedure a move. Note that some heuristics are looking for the best improving move before applying the replacement of a move. In addition, for each pair of edges, there is only one move reconnecting the graph that is not the null move. The iteration on pairs leads to a time complexity in $O(n^2)$. Figure 1 shows an example where $e_1 = (x_1, x_2)$, $e_2 = (y_1, y_2)$ and the move is $e_3 = (x_1, y_1)$, $e_4 = (x_2, y_2)$.

For the 3-opt algorithm, instead of choosing pair of edges, we choose a triplet of edges and, as for 2-opt, we search for moves reducing the overall cost of the tour. In that case, there are seven ways to reconnect the graph. Note that three of them are simple 2-opt (that is a combination with one edge of the triplet not moved). Thus, 3-opt allows checking more sophisticated combination than 2-opt and then can potentially find better moves. However, it leads to an algorithm with a time complexity in $O(n^3)$. Figure 2 shows an example of all 3-opt moves that are not 2-opt.



■ **Figure 2** An example of 3-opt. The circle represents a tour and the dashed lines are the suggested move for the triplet of edges $((x_1, x_2), (y_1, y_2), (z_1, z_2))$.

Naturally, the 2-opt and the 3-opt algorithms can be generalized to the k -opt algorithm with a time complexity in $O(n^k)$. Experiments have shown that increasing the value of k improves the quality of the tours but slows down solving times. Thus, some methods [19, 3] consider some 3-opt and/or 4-opt, but not all, in order to reduce the time complexity and speed up the solving times.

Moreover, Lin and Kernighan suggested to use a variable k while solving [17] in order to include larger moves. The algorithm is therefore more complex but it greatly improves the results (tour quality and solving times). To do so, they suggested several rules. First, they are looking for the most promising permutations only. Next, they allow improving k -opt moves that can be built from a sequence of 2-opt moves such that some moves do not improve the tour. These moves are much more complex and provide better moves than a simple run of the 2-opt algorithm. In order to make this algorithm extremely efficient, Helsgaun [12] has remarkably refined most of the rules given by Lin and Kernighan [17]. Nowadays, the Lin-Kernighan-Helsgaun algorithm is considered as one of the most efficient heuristic solving the TSP and therefore it is embedded in most of the exact methods.

In this paper, we integrate 2-opt and 3-opt concepts into CP. Unlike tour improvement algorithms, the CP model does not have a tour to improve. However, the CP model has mandatory edges that can form paths and try to find a tour going through these paths. We then search for 2-opt and 3-opt in the paths of mandatory edges.

3 Mandatory Hamiltonian path constraint

We note $M'(i)$ (resp. $O'(i)$) the set of mandatory (resp. optional) edges having i for extremity in M' (resp. O'). For each node i , $|M'(i)| \leq 2$ because of the degree constraint. Thus, the mandatory edges form disjoint paths. Without loss of generality, we assume that the current assignment of the G_{solve} is consistent with the degree constraint.

► **Definition 1.** A *mandatory Hamiltonian path* p is a path such that p is a Hamiltonian path in a subgraph of G_{solve} and for each edge $e = (x_i, x_{i+1})$ of p , $e \in M'$.

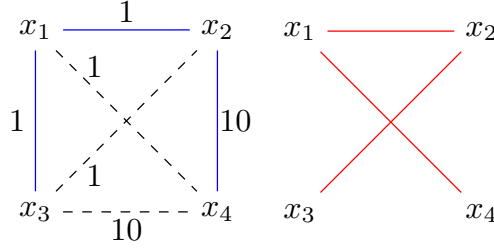
We note $p_1 = [x_1, x_2, \dots, x_t]$ a mandatory Hamiltonian path of G_{solve} .

3.1 Consistency Check

In this section, we will study the existence of optimal solutions in G_{solve} .

► **Definition 2.** An *alternative path* $p_2 = [x'_1, x'_2, \dots, x'_t]$ of p_1 is a permutation of the nodes of p_1 such that $p_1 \neq p_2$, $x_1 = x'_1$, $x_t = x'_t$ and for each $i \in [1, t-1]$, $(x'_i, x'_{i+1}) \in U$.

Figure 3 shows an example of an alternative path. Thus, an alternative path can be composed of edges in $M \cup O \cup D$, i.e. in U .



■ **Figure 3** The left graph is a subgraph of G_{init} . The blue edges are from M , they form a mandatory Hamiltonian path going from x_3 to x_4 . The dashed edges are from D (the deleted edges). The right graph is an alternative path of the left graph.

► **Definition 3.** *The mandatory Hamiltonian path p_1 is minimal if and only if there is no alternative path p_2 of p_1 such that $w(p_2) < w(p_1)$.*

In Figure 3, the mandatory Hamiltonian path is not minimal. The right graph represents an alternative path with a cost of 4 whereas the mandatory Hamiltonian path has a cost of 12. The idea is to search for a non-minimal mandatory Hamiltonian path p_2 in the connected components of $G_{mand} = (X, M', \emptyset)$. In Proposition 4, we show that if such a path p_2 exists, then the cost of the TSP in $G_{solve} = (X, M', O')$ is greater than the cost of the TSP in $G_{init} = (X, M, O)$.

► **Proposition 4.** *If there is a mandatory Hamiltonian path p_1 that is not minimal, then p_1 cannot belong to any solution of $TSP(G_{init})$.*

Proof. Given $w(TSP(G_{init} + p_1))$ the cost of $TSP(G_{init})$ such that p_1 is in the solution. If there is no solution for $TSP(G_{init} + p_1)$, then p_1 cannot belong to any solution $TSP(G_{init})$. Otherwise, if p_1 is not minimal, then there is an alternative path p_2 of p_1 such that $w(p_2) < w(p_1)$. Thus, $w(TSP(G_{init} + p_2)) < w(TSP(G_{init} + p_1))$ and therefore p_1 cannot belong to any $TSP(G_{init})$. ◀

In the context of a CP solver, if there is a mandatory Hamiltonian path p that is not minimal, then from Proposition 4 we can trigger a failure because the current solution is not minimal. Moreover, it raises a question: how do we verify if a mandatory Hamiltonian path is minimal? A first algorithm consists in checking all the possible permutations for each mandatory Hamiltonian path. Unfortunately, checking all the permutations leads to an impractical algorithm. However, a large number of heuristics improving tours have been designed. Among them, there are the ones introduced in Subsection 2.3. Thus, we can use any of these heuristics on the mandatory Hamiltonian paths. If it finds an improvement, then a mandatory Hamiltonian path is not minimal and therefore we can trigger a failure.

In this paper, we use k -opt heuristics as tour improvement since they are very efficient and easy to implement. In Section 4, we will show that 2-opt and 3-opt are enough in order to obtain good results.

► **Definition 5.** *Given the set of mandatory Hamiltonian paths P and an integer k . For each mandatory Hamiltonian path $p \in P$, the mandatory Hamiltonian path constraint ensure that there is no alternative path p' obtained by swapping k edges of p such that $w(p') < w(p)$.*

Therefore, we define the mandatory Hamiltonian path constraint in Definition 5 such that if its consistency is not verified, then we trigger a failure.

■ **Algorithm 1** Consistency check of the mandatory Hamiltonian paths.

```

1 ConsistencyCheck ( $G_{init}, G_{mand}, k$ )
   Input: The initial graph  $G_{init}$ , the graph of mandatory edges  $G_{mand}$  and an
           integer  $k$ .
   Output: A Boolean specifying whether  $G_{mand}$  contains a mandatory
           Hamiltonian path that is not minimal.
2    $P \leftarrow \text{computeMandatoryHamiltonianPaths}(G_{mand})$  ;
3   foreach  $path\ p \in P$  do
4     if  $k\text{-optPath}(G_{init}, p)$  then
5       return False ;
6   return True ;

```

In Algorithm 1, we introduce an implementation of the algorithm checking the consistency of the mandatory Hamiltonian path constraint. We assume that $k\text{-optPath}(G_{init}, p)$ returns true if and only if the mandatory Hamiltonian path constraint with the given k is consistent. Internally, $k\text{-optPath}(G_{init}, p)$ uses a $k\text{-opt}$ heuristic. Then, for each mandatory Hamiltonian path p , we run $k\text{-optPath}(G_{init}, p)$ in $O(|p|^k)$.

► **Proposition 6.** *Given P the set of mandatory Hamiltonian paths. Then, $\sum_{p \in P} |p| \leq n$ and $|P| \leq n$.*

Proof. By definition, each node can only be contained in one mandatory Hamiltonian path and there are n nodes in G_{solve} . Thus, $\sum_{p \in P} |p| \leq n$. In addition, if each node is contained in a different path, then there are n paths and then therefore $|P| \leq n$. ◀

Each p of P are disjoint. From Proposition 6, the sum of $|p|$ for all $p \in P$ is lower or equal to n . Finally, the time complexity of Algorithm 1 is in $O(\sum_{p \in P} |p|^k) < O(n^k)$.

3.2 Filtering algorithm

In this section, we will consider that the consistency has been checked. An edge $e = (x_t, x_i)$ is a successor of p_1 and an edge $e = (x_i, x_1)$ is a predecessor of p_1 . In addition, we note $x_i + p_1 = [x_i, x_1, x_2, \dots, x_t]$ and $p_1 + x_i = [x_1, x_2, \dots, x_t, x_i]$.

From Proposition 4, we have the two following corollaries:

► **Corollary 7.** *For each edge $e \in O'$ such that e is a predecessor of p_1 , if $i + p_1$ is not a minimal mandatory Hamiltonian path, then e cannot belong to any solution of $TSP(G_{solve})$.*

► **Corollary 8.** *For each edge $e \in O'$ such that e is a successor of p_1 , if $p_1 + i$ is not a minimal mandatory Hamiltonian path, then e cannot belong to any solution of $TSP(G_{solve})$.*

Thus, in order to define a filtering algorithm we are interested in the minimality of $p_1 + i$ and $i + p_1$. If the minimality of p_1 has already been checked, then we can avoid all permutations only containing the elements of p_1 . We impose i to be in the considered permutations and we look for permutations of size $(k - 1)$ in p_1 . Then, for a mandatory Hamiltonian path p and a single successor or predecessor, we can filter the edge in $O(|p|^{k-1}) < O(n^{k-1})$. Performing the filtering for all predecessors and successors of p can be done in $O(|O'(p)| |p|^{k-1}) < O(n^k)$. From Proposition 6, there can be at most n paths and the sum of the size of all paths is smaller than or equal to n . Thus, from Corollary 7 and 8 we can filter the edges of all paths with a complexity in $O(n^{k+1})$. Note that the number of checked permutations in practice is much smaller.

A mandatory Hamiltonian path p_1 can have a successor or a predecessor e connecting another mandatory Hamiltonian path p_2 . Then, adding e to the solution leads to a minimality check in $p_1 + p_2$. We can then extend the two previous corollaries:

► **Corollary 9.** *For each edge $e = (x_i, x_1) \in O'(x_1)$, if it exists $p_2 = [x'_1, x'_2, \dots, x'_t]$ a mandatory Hamiltonian path of G_{solve} such that $x_i = x'_t$ and $p_2 + p_1$ is not a minimal mandatory Hamiltonian path, then e cannot belongs to any $TSP(G_{solve})$.*

► **Corollary 10.** *For each edge $e = (x_i, x_t) \in O'(x_t)$, if it exists $p_2 = [x'_1, x'_2, \dots, x'_t]$ a mandatory Hamiltonian path of G_{solve} such that $x_i = x'_1$ and $p_1 + p_2$ is not a minimal mandatory Hamiltonian path, then e cannot belongs to any $TSP(G_{solve})$.*

Given a mandatory Hamiltonian path p_2 of G_{solve} connected to p_1 with $e \in O'$. Then, we have to check if $p_1 + p_2$ is minimal in order to determine whether e can be in a solution of $TSP(G_{solve})$. It can be done with Corollary 9 and 10 in $O((|p_1| + |p_2|)^k - |p_1|^k - |p_2|^k) < O(n^k)$. The number of predecessors and successors of p_1 is at most $2n$. If $P(p_1)$ is the set of mandatory Hamiltonian paths such that each path of $P(p_1)$ is connected to p_1 with a successor or a predecessor of p_1 , then the filtering on p_1 can be done in $O(\sum_{p_2 \in P(p_1)} (|p_1| + |p_2|)^k - |p_1|^k - |p_2|^k) < O(n^{k+1})$. Given P the set of mandatory Hamiltonian paths. The filtering for all paths of P can be done in $O(\sum_{p_1 \in P} \sum_{p_2 \in P(p_1)} (|p_1| + |p_2|)^k - |p_1|^k - |p_2|^k) < O(n^{k+2})$. Algorithm 2 is a possible implementation.

For the sake of clarity, we will use the following notations in the algorithms:

- P : contains all the mandatory Hamiltonian paths of the graph G_{solve} .
- $P[i]$: if there is a path p containing the node i , then it returns p . Otherwise, it returns i .
- $p.first()$: returns the first node of the path p .
- $p.last()$: returns the last node of the path p .

■ **Algorithm 2** Filtering algorithm for the mandatory Hamiltonian paths.

```

1 Filter ( $G_{init}, G_{solve} = (X, M', O'), P, k$ )
   Input: The initial graph  $G_{init}$ , a graph  $G_{solve}$ , the set of mandatory
           Hamiltonian paths  $P$  and an integer  $k$ .
2 foreach  $p_1 = [x_1, x_2, \dots, x_t] \in P$  do
3   foreach edge  $e = (x_1, j) \in O'(x_1)$  do
4      $p_2 \leftarrow P(j)$  ;
5     if  $p_2.last() \neq j$  then reverse( $p_2$ );
6     if ( $p_1 = p_2$  and  $|M'| \neq n - 1$ ) or  $k\text{-optPath}(G_{init}, p_2, p_1)$  then
7        $O' \leftarrow O' - e$  ;
8   foreach edge  $e = (x_t, j) \in O'(x_t)$  do
9      $p_2 \leftarrow P(j)$  ;
10    if  $p_2.first() \neq j$  then reverse( $p_2$ );
11    if ( $p_1 = p_2$  and  $|M'| \neq n - 1$ ) or  $k\text{-optPath}(G_{init}, p_1, p_2)$  then
12       $O' \leftarrow O' - e$  ;

```

Given P the set of the mandatory Hamiltonian paths. For each path $p_1 \in P$, we perform the filtering on all the predecessor and successor e of p_1 . We note p_2 the path connected to p_1 by e (p_2 can be a single node). In addition, we note $k\text{-optPath}(graph, p_1, p_2)$ the $k\text{-optPath}$ algorithm considering the permutations of $p_1 + p_2$ such that each permutation contains at

least one element of p_1 and at least one element of p_2 . When two paths are merged, they must be in the right order. If p_2 must be inserted in front of p_1 , then the node j must be the last node of p_2 . Otherwise, j must be the first node of p_2 . Thus, p_2 is reversed if needed. Note that we can save the reversed path in order to avoid redundant computations. If an improvement is found when p_1 and p_2 are merged, then from Corollary 9 or 10 the edge e cannot belong to a solution of $TSP(G_{solve})$ and therefore e is removed from the optional edges of G_{solve} . In addition, if $p_1 = p_2$ and $|M'| \neq n - 1$, then it exists an edge $e = (i, j)$ such that i and j belong to the same mandatory Hamiltonian path and therefore the edge close a cycle with a size lower than n . Thus, adding e to the solution creates a sub-cycle and then e is removed from the optional edges of G_{solve} .

3.3 Maintenance during the search

In this section, we will consider the incremental aspect of this constraint, *i.e.* the consistency of this constraint or its filtering when some edges become mandatory or deleted. Moreover, we will consider the restoration of the data structures introduced for the incremental aspect when a backtrack occurs. In this study, an edge can be deleted or an edge becomes mandatory.

► **Proposition 11.** *Given $G' = (X, M', O'')$ such that $O'' \subseteq O'$. If p_1 is minimal, then p is minimal in G' .*

Proof. The graph G' is the graph G_{solve} such that some edges are deleted. By definition, the deleted edges are in D and the alternative paths can contain edges of D . Thus, if p_1 is minimal, then p is minimal in G' . ◀

From Proposition 11, if we know that all the mandatory Hamiltonian paths of G_{solve} are minimal and then some edges are removed, then the mandatory Hamiltonian paths of G_{solve} remain minimal. In addition, removing some edges does not change the result of the filtering algorithm since new alternative paths cannot be created from removal. Thus, the consistency test and the filtering algorithm are only triggered when there are new mandatory edges.

In the following algorithms, we use the following data structures:

- candidates: a stack of graph nodes such that the nodes are adjacent to edges that can be filtered.
- deltaMand: a set of the new mandatory edges since the last call of the constraint for the current search node.

3.3.1 Consistency check

When an edge e becomes mandatory, there are three cases:

- e is not connected to any path and therefore e creates a new path only containing its two endpoints. Note that a mandatory Hamiltonian path with two nodes is necessarily minimal.
- e is connected to a mandatory Hamiltonian path p and therefore p and e are merged in a not necessarily minimal mandatory Hamiltonian path because new alternative paths may exist.
- e is connected to two mandatory Hamiltonian paths p_1 and p_2 and therefore p_1 and p_2 are merged in a not necessarily minimal mandatory Hamiltonian path because new alternative paths may exist.

Thus, for consistency check, we only consider the paths that must be merged. In addition, given a new mandatory edge e connecting p_1 and p_2 , we note p_3 the merged path of p_1 and p_2 . When two paths are merged, we assume that the minimality check has been performed

on the two paths. Therefore, when the k -optPath algorithm is checking the minimality for the path p_3 , it can avoid the permutations containing either only elements of p_1 or only elements of p_2 . Then, we consider the permutations that contain at least one element of p_1 and at least one element of p_2 .

In Algorithm 3, we give a possible implementation of the incremental algorithm checking the minimality of the mandatory Hamiltonian paths. For each edge (i, j) newly mandatory, we have p_1 and p_2 the mandatory Hamiltonian paths such that i and j are respectively an extremity of p_1 and p_2 . Therefore, the edge (i, j) merge p_1 and p_2 and p_1 and/or p_2 are accordingly reversed. Note that the *candidates* stack is filled for the filtering algorithm. Then, we run the k -optPath algorithm in order to find alternative paths in $p_1 + p_2$. Note that we only consider permutations such that each permutation contains at least one element of p_1 and at least one element of p_2 . Finally, if no alternative path is found, $p_1 + p_2$ is a minimal mandatory Hamiltonian path and we merge p_1 and p_2 in P . Otherwise, we return False and a failure is triggered.

■ **Algorithm 3** Incremental minimality check of the mandatory Hamiltonian paths.

```

1 IncrementalConsistencyCheck ( $G_{init}, P, \text{deltaMand}, \text{candidates}, k$ )
   Input: The initial graph  $G_{init}$ , the set of mandatory Hamiltonian paths  $P$ , the
           set of new mandatory edges  $\text{deltaMand}$ ,  $\text{candidates}$  a filtering used stack
           and an integer  $k$ .
   Output: A Boolean specifying whether  $P$  contains a mandatory Hamiltonian
           path that is not minimal.
2 foreach  $(i, j) \in \text{deltaMand}$  do
3    $p_1 \leftarrow P[i]$  ;
4    $p_2 \leftarrow P[j]$  ;
5   if  $p_1.\text{last}() \neq i$  then  $\text{reverse}(p_1)$ ;
6   if  $p_2.\text{first}() \neq j$  then  $\text{reverse}(p_2)$ ;
7    $\text{candidates.push}(p_1.\text{first}())$  ;
8    $\text{candidates.push}(p_2.\text{last}())$  ;
9   if  $k\text{-optPath}(G_{init}, p_1, p_2)$  then
10    return False ;
    // merge  $p_1$  and  $p_2$  in  $P$ 
11     $\text{merge}(P, p_1, p_2)$  ;
12 return True ;

```

The overall time complexity of Algorithm 3 is $O(\sum_{(i,j) \in \text{deltaMand}} (|P[i]| + |P[j]|)^k - |P[i]|^k - |P[j]|^k) < O(n^k)$. Note that Algorithm 1 has a time complexity in $O(\sum_{p \in P} |p|^k) < O(n^k)$ when all paths are already merged which is equivalent to $O(\sum_{(i,j) \in \text{deltaMand}} (|P[i]| + |P[j]|)^k)$ if paths are not merged. Thus, the incremental algorithm improves the time complexity for checking the minimality of the mandatory Hamiltonian paths.

3.3.2 Filtering algorithm

When an edge e becomes mandatory, we have the same three cases as for the consistency check. Thus, we will only consider the merged mandatory Hamiltonian paths in the previous consistency check. More precisely, we will only consider the neighborhood of the first node and the last node of these paths.

Algorithm 4 is a possible implementation of an incremental algorithm performing the filtering. First, we iterate on *candidates*. Every time two paths are merged in Algorithm 3, the first and last nodes of the merged path are pushed in *candidates*. Thus, *candidates* contains the first node and last nodes of all merged paths. In addition, *candidates* may contain some nodes that are “intermediate” merged paths. For example, merging p_1 and p_2 results in p_3 such that $p_3.first() = x$ and $p_3.last() = y$. Then, x and y are pushed in *candidates*. Merging p_3 with p_4 results in p_5 such that $p_5.first() = x'$ and $p_5.last() = y'$. Then, x' and y' are pushed in *candidates*. However, x and y still are in *candidates* while x or y is no longer the first node or the last node of a merged path. Then, while iterations on candidates, we need to avoid these nodes. Finally, if a node i is an endpoint of a mandatory Hamiltonian path, then we check in the neighborhood of the node i (same as for Algorithm 2).

■ **Algorithm 4** Incremental filtering of the mandatory Hamiltonian paths.

```

1 IncrFiltering ( $G_{init}, G_{solve} = (X, M', O'), P, candidates, k$ )
   Input: The initial graph  $G_{init}$ , a graph  $G_{solve}$ , the set of mandatory
           Hamiltonian paths  $P$ , the stack of nodes to consider for the filtering
           candidates and an integer  $k$ .
2   while candidates.isNotEmpty() do
3        $i \leftarrow candidates.pop()$  ;
4        $p_1 \leftarrow P[i]$  ;
5       if  $i = p_1.first()$  or  $i = p_1.last()$  then
6           foreach edge  $e = (i, j) \in O'(i)$  do
7                $p_2 \leftarrow P[j]$  ;
8               if  $p_1.last() \neq i$  then  $reverse(p_1)$ ;
9               if  $p_2.first() \neq j$  then  $reverse(p_2)$ ;
10              if  $(p_1 = p_2 \text{ and } |M'| \neq n - 1) \text{ or } k\text{-optPath}(G_{init}, p_1, p_2)$  then
11                   $O' \leftarrow O' - e$  ;

```

If $P(i)$ is the set of mandatory Hamiltonian paths such that each path of $P(i)$ is connected with a successor or a predecessor of $P[i]$, then the time complexity of Algorithm 4 is in $O(\sum_{i \in candidates} \sum_{p_2 \in P(i)} (|P[i]| + |p_2|)^k - |P[i]|^k - |p_2|^k)) < O(n^{k+2})$.

3.3.3 Restoration

In order to save more computations, we maintain the set P of mandatory Hamiltonian paths. When a backtrack occurs, the difference between the backtracked state and the current state is that some mandatory edges could have been found and therefore some mandatory Hamiltonian paths of P could have been merged. Thus, in order to restore P , the merged mandatory Hamiltonian paths should be split. To do so, we define a stack S such that S contains the added mandatory edges from the root to the current state. In addition, we save the size of the stack for each open search node. Then, when a backtrack occurs, we iteratively pop the mandatory e edges of S until the wanted size is obtained. For each e , we split the mandatory Hamiltonian path in P containing e .

4 Experiments

The algorithms have been implemented in Java 11 in a locally developed constraint programming solver. The experiments were performed on Clear Linux with an Intel Xeon E5-2696v2 and 64 GB of RAM. The instances are from the TSPLib [20], a library of reference graphs for the TSP. We rerun experiments ran in Isoart and Régim [13] and exclude instances solved in less than two seconds by the state of the art. In addition, we tried some harder instances from the TSPLib and selected those that did not have reached a time out *t.o.* by both the state of the art and our method. Note that we set *t.o.* to 1 week, that is 604,800 seconds. The name of each instance is suffixed by its number of nodes. In our implementation, the TSP is modeled by the WCC using the CP-based LR configuration introduced in Isoart and Régim [14]. We note “state of the art” the TSP model introduced in Subsection 2.2, “MHP 2-opt” the state of the art combined with the mandatory Hamiltonian path constraint searching for 2-opt and “MHP 3-opt” the state of the art combined with the mandatory Hamiltonian path constraint searching for 3-opt. The search strategy used is LCFfirst with the heuristic minDeltaDeg [7] which is also the state of the art. Given $e = (i, j)$ an edge, minDeltaDeg selects the edge with the minimum difference between the sum of the number of optional neighbors of i and j and the sum of the number of mandatory neighbors of i and j . Thus, we compare our constraint and the state of the art through the number of backtracks (#bk) and the solving times in seconds in arrays. All considered instances are symmetric graphs. If not specified, we use the implementation given in Algorithm 3 and 4.

Table 1 shows the solving times and the number of backtracks for the state of the art solving method and with 2-opt and 3-opt added to it. In addition, we display a ratio column in order to show the gain factor for each instance by using 2-opt and 3-opt.

For the state of the art, we notice that 4 instances over 32 have reached the time out. For the mandatory Hamiltonian path constraint combined with 2-opt, we notice that only 2 of the 4 instances have reached the time out. Indeed, pr299 is solved in 9,640s and rd400 is solved in 28,122s with 2-opt whereas they remain unsolved in 604,800s with the state of the art.

Most of the time, we notice that the use of 2-opt allows us to improve the solving times. For example, ali535 is improved by a factor of 3.5 in solving time and by a factor of 3.7 in backtracks. Some problems have higher gain factors: d493 gains a factor 6 in solving time and a factor 4.9 in backtracks. Moreover, only pr124 has a degraded solving time when using 2-opt: 2.8s vs 3.3s. Note that there is gain in backtracks 1856 vs 1700.

The mandatory Hamiltonian path constraint combined with 3-opt allow us to obtain an additional improvement to the use of 2-opt only. Indeed, 3-opt can be slower than 2-opt in terms of backtracks/second but it greatly reduce the number of backtracks. Note that this configuration solve all the considered instances. Indeed, pr299 is solved in 3,039s, pr493 is solved in 170,127s, rd400 is solved in 12,352s and u574 is solved in 198,693s with the mandatory Hamiltonian path constraint combined with 3-opt whereas they remain unsolved in 604,800s with the state of the art. We thus obtain great improvement factors on the solving times: > 199 for pr299, > 3.6 for pr493, > 49 for rd400 and > 3 for u574. In addition, some instances are solved much faster with 3-opt than with 2-opt: gr666 is solved in 303.293s with the state of the art, 64,391s with 2-opt and 21,853s with 3-opt. Some other instances are solved with almost the same number of backtracks for 2-opt and 3-opt: for ali535 with 2-opt there is 3,148,626bk and there is 3,178,482bk with 3-opt. However, it has a slower solving time with 2-opt than with 3-opt: 24,367s vs 35,026s. This can be due to several reasons: the extra cost of using an algorithm in $O(n^3)$ compared to an algorithm in $O(n^2)$.

■ **Table 1** General results comparing the mandatory Hamiltonian path constraint combined with 2-opt or 3-opt and the state of the art.

	State of the art (1)		MHP 2-opt (2)		ratio (1)/(2)		MHP 3-opt (3)		ratio (1)/(3)	
	time(s)	#bk	time(s)	#bk	time	#bk	time(s)	#bk	time(s)	#bk
a280	7.0	2,372	6.5	2,182	1.1	1.1	10.8	3,134	0.6	0.8
ali535	84,929.1	11,747,704	24,367.5	3,148,626	3.5	3.7	35,026.0	3,178,482	2.4	3.7
ch150	2.9	1,526	2.1	644	1.4	2.4	1.7	392	1.7	3.9
d198	14.0	7,192	12.6	6,062	1.1	1.2	10.4	3,694	1.4	1.9
d493	95,916.6	13,478,616	15,931.0	2,778,780	6.0	4.9	31,162.1	1,877,298	3.1	7.2
gil262	5,230.2	2,254,728	3,804.8	1,710,410	1.4	1.3	3,833.3	1,501,756	1.4	1.5
gr137	3.4	1,910	1.7	706	2.0	2.7	1.5	518	2.2	3.7
gr202	2.4	886	2.0	600	1.2	1.5	2.3	448	1.0	2.0
gr229	227.0	166,378	64.6	44,696	3.5	3.7	59.1	33,336	3.8	5.0
gr431	1,724.8	265,698	494.8	68,432	3.5	3.9	556.3	65,100	3.1	4.1
gr666	303,293.4	28,432,754	64,390.7	5,168,402	4.7	5.5	24,853.1	1,721,794	12.2	16.5
kroA100	2.0	1,270	1.2	438	1.7	2.9	1.3	458	1.6	2.8
kroA150	6.1	4,164	5.2	3,374	1.2	1.2	3.9	1,814	1.6	2.3
kroA200	401.0	237,806	63.8	33,166	6.3	7.2	68.2	34,058	5.9	7.0
kroB100	5.4	4,816	2.8	2,164	1.9	2.2	1.7	972	3.2	5.0
kroB150	262.6	247,574	30.5	23,296	8.6	10.6	21.7	16,012	12.1	15.5
kroB200	127.8	87,296	34.5	21,060	3.7	4.1	15.8	8,140	8.1	10.7
kroC100	2.0	1,470	1.1	346	1.8	4.2	1.1	334	1.8	4.4
kroE100	2.3	1,804	1.6	782	1.4	2.3	1.6	824	1.4	2.2
lin318	32.9	7,834	8.7	1,944	3.8	4.0	10.1	2,018	3.3	3.9
pr124	2.8	1,856	3.3	1,700	0.8	1.1	2.3	1,142	1.2	1.6
pr136	20.4	18,684	16.2	13,886	1.3	1.3	13.1	8,598	1.6	2.2
pr144	2.3	1,036	1.8	628	1.3	1.6	1.9	594	1.2	1.7
pr264	4.7	690	4.9	508	1.0	1.4	5.1	524	0.9	1.3
pr299	<i>t.o.</i>	<i>t.o.</i>	9,640.5	2,710,230	> 62.7	-	3,038.7	805,344	> 199.0	-
pr439	<i>t.o.</i>	<i>t.o.</i>	<i>t.o.</i>	<i>t.o.</i>	-	-	170,127.1	35,750,706	> 3.6	-
rat195	38.5	24,274	17.9	10,286	2.2	2.4	17.8	8,560	2.2	2.8
rd400	<i>t.o.</i>	<i>t.o.</i>	28,121.1	6,524,576	> 21.5	-	12,351.9	2,507,272	> 49.0	-
si175	288.5	301,102	204.5	197,968	1.4	1.5	342.6	275,870	0.8	1.1
tsp225	121.5	65,002	116.8	59,688	1.0	1.1	51.4	24,042	2.4	2.7
u574	<i>t.o.</i>	<i>t.o.</i>	<i>t.o.</i>	<i>t.o.</i>	-	-	198,962.7	28,269,058	> 3.0	-

The Lagrangian relaxation can also be impacted by the filtered edges. However, since 3-opt solves more problems than 2-opt and that on average (if we do not consider the instances that have reached the time out) we obtain a gain of a factor of 2.5 for 2-opt and 3 for 3-opt over the state of the art. Thus, we will consider the version with 3-opt. Note that we also could use some other heuristics such as 2.5-opt that compute 2-opt and some 3-opt. The improvement over the number of backtracks is not as much important as for the 3-opt method but the number of backtracks per second is higher. In practice, we have observed on average a 10% difference on the solving times between 3-opt and 2.5-opt.

In Table 2, we show the impact of the use of the incremental version of the mandatory Hamiltonian path constraint on some instances of Table 1. On this instance set, the incremental version is on average 33% faster than the non-incremental one. With the incremental version, the solving times of some instances such as d198 are improved of 8% whereas for other instances such as gr229 the solving times are improved of 50%. Thus, the benefit of avoiding recalculations may be interesting for this constraint due to the time complexity of the k -opt algorithm.

In Table 3, we are interested in the use of k -opt algorithms with k greater than 3. For the number of backtracks, we notice that on average 4-opt is more efficient than 3-opt which is more efficient than 2-opt. In addition, 4-opt and 5-opt achieve similar results. However,

■ **Table 2** Comparison of solving times for the non-incremental and the incremental version of the mandatory Hamiltonian path constraint.

	(1) 3-opt not incremental time(s)	(2) 3-opt incremental time(s)	ratio (1) / (2) time
a280	16.2	10.8	1.49
d198	11.2	10.4	1.08
gr229	90.8	59.1	1.53
kroA200	75.1	68.2	1.10
pr136	19.5	13.1	1.49
rat195	22.5	17.8	1.26
mean	39.22	29.92	1.33

■ **Table 3** Comparison of solving times for mandatory Hamiltonian path constraint with 2-opt, 3-opt, 4-opt and 5-opt.

	2-opt		3-opt		4-opt		5-opt	
	time(s)	#bk	time(s)	#bk	time(s)	#bk	time(s)	#bk
a280	6.5	2,182	10.8	3,134	132.0	3,386	41,884.8	3,386
ch150	2.1	644	1.7	392	3.0	392	246.9	392
d198	12.6	6,062	10.4	3,694	32.3	3,694	4,839.9	3,694
gr229	64.6	44,696	59.1	33,336	97.4	23,930	11,001.2	26,036
kroA200	63.8	33,166	68.2	34,058	71.1	31,050	1,450.7	31,050
pr136	16.2	13,886	13.1	8,598	54.5	6,496	8,183.5	6,496
rat195	17.9	10,286	17.8	8,560	56.9	10,192	5,816.3	10,192
mean	26.2	15,846.0	25.9	13,110.3	63.9	11,305.7	10,489.0	11,606.6

the use of 4-opt and 5-opt degrades the solving times compared to 2-opt and 3-opt. Indeed, for 4-opt we observe a loss of a factor greater than 2. For 5-opt, we observe a loss of a factor greater than 400. Thus, the solving times and number of backtracks trade-off is not good when $k > 3$.

In Table 4, we show the gap between the MIP solver Concorde [1] and the state of the art CP solving method with the mandatory Hamiltonian path constraint with 3-opt. Note that the results for Concorde are obtained on our machine. For the small sized instances, we notice that our method is competitive with Concorde. Indeed, small sized instances such as att48 are solved in 0.14s with Concorde whereas we solved it in 0.03s. For medium sized instances such as rat195, we can see a slight degradation of the results: Concorde solved it in 8.73s whereas we solved it in 17.82s. However, the solving times are still comparable. Unfortunately, our method starts to slowing down for larger instances. For example, rd400 is solved in 20.6s with Concorde whereas it is solved in 12,351.85s with our method. Nevertheless, in [2], the solving time ratio with Concorde of kroC100 is about 1000, here it is only 3.1. Thus, we hope that same improvement factors will be obtained for larger instances in future works.

5 Conclusion

In this paper, we introduced a new constraint based on the k -opt algorithm, named mandatory Hamiltonian path constraint, into to the TSP model in CP. We also introduced an incremental version of this constraint. Experiments have shown that the use of this constraint leads to an

■ **Table 4** Comparison of the solving times and the number of backtracks for mandatory Hamiltonian path constraint with 3-opt and Concorde.

	Concorde		3-opt		ratio time
	time(s)	#bk	time(s)	#bk	
gr24	0.02	0	0.00	2	0.0
att48	0.14	0	0.03	6	0.2
eil51	0.07	0	0.05	32	0.8
st70	0.12	0	0.14	70	1.1
kroC100	0.35	0	1.08	334	3.1
bier127	0.31	0	0.28	60	0.9
gr137	1.32	0	1.55	518	1.2
ch150	0.93	0	1.73	392	1.9
si175	3.58	2	342.64	275,870	95.8
rat195	8.73	6	17.82	8,560	2.0
gr202	2.92	0	2.30	448	0.8
lin318	2.59	0	10.12	2,018	3.9
ali535	6.72	0	35,025.96	3,178,482	5215.3
d493	47.17	4	31,162.09	1,877,298	660.6
rd400	20.60	8	12,351.85	2,507,272	599.7

improvement of at least a factor of 3 in solving times. In addition, it shown that the use of 3-opt is well suited for our constraint. Moreover, we have been able to solve some instances that remains unsolved with the state of the art CP model. The k -opt algorithm is embedded in most of the solving methods of the TSP and therefore now in the CP. In future work, we will study an extension of this constraint not only considering the mandatory Hamiltonian paths but the mandatory cutsets in the graph.

References

- 1 David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2006.
- 2 Pascal Benchimol, Willem-Jan Van Hoeve, Jean-Charles Régim, Louis-Martin Rousseau, and Michel Rueher. Improved filtering for weighted circuit constraints. *Constraints*, 17(3):205–233, 2012. URL: <https://hal.archives-ouvertes.fr/hal-01344070>.
- 3 Jon Louis Bentley. Fast algorithms for geometric traveling salesman problems. *ORSA Journal on computing*, 4(4):387–411, 1992.
- 4 Václav Chvátal. Edmonds polytopes and weakly hamiltonian graphs. *Math. Program.*, 5(1):29–40, 1973. doi:10.1007/BF01580109.
- 5 George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.
- 6 Jack Edmonds. Paths, Trees, and Flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965. doi:10.4153/CJM-1965-045-4.
- 7 Jean-Guillaume Fages, Xavier Lorca, and Louis-Martin Rousseau. The salesman and the tree: the importance of search in CP. *Constraints*, 21(2):145–162, 2016.
- 8 Martin Grötschel and Manfred Padberg. On the symmetric travelling salesman problem i: Inequalities. *Mathematical Programming*, 16:265–280, December 1979. doi:10.1007/BF01582116.
- 9 Robert Haralick and Gordon Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, January 1979.

- 10 Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.
- 11 Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees: Part ii. *Mathematical Programming*, 1(1):6–25, 1971.
- 12 Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000. doi:10.1016/S0377-2217(99)00284-2.
- 13 Nicolas Isoart and Jean-Charles Régin. Integration of structural constraints into tsp models. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming*, pages 284–299, Cham, 2019. Springer International Publishing.
- 14 Nicolas Isoart and Jean-Charles Régin. Adaptive CP-Based Lagrangian Relaxation for TSP Solving. In Emmanuel Hebrard and Nysret Musliu, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 300–316, Cham, 2020. Springer International Publishing.
- 15 Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.
- 16 Adam N. Letchford and Andrea Lodi. Polynomial-Time Separation of Simple Comb Inequalities. In William J. Cook and Andreas S. Schulz, editors, *Integer Programming and Combinatorial Optimization*, pages 93–108, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
- 17 S. Lin and B. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Oper. Res.*, 21:498–516, 1973.
- 18 Shen Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44:2245–2269, 1965.
- 19 Ilhan Or. Traveling salesman type combinatorial problems and their relation to the logistics of regional blood banking, 1977.
- 20 Gerhard Reinelt. TSPLIB – A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3(4):376–384, 1991.
- 21 Robert E. Tarjan. *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics, 1983.

The Seesaw Algorithm: Function Optimization Using Implicit Hitting Sets

Mikoláš Janota   

Czech Technical University in Prague, Czech Republic

António Morgado   

INESC-ID Lisbon, Portugal

José Fragoso Santos  

INESC-ID/IST, University of Lisbon, Portugal

Vasco Manquinho  

INESC-ID/IST, University of Lisbon, Portugal

Abstract

The paper introduces the Seesaw algorithm, which explores the Pareto frontier of two given functions. The algorithm is complete and generalizes the well-known implicit hitting set paradigm. The first given function determines a cost of a hitting set and is optimized by an exact solver. The second, called the oracle function, is treated as a black-box. This approach is particularly useful in the optimization of functions that are impossible to encode into an exact solver. We show the effectiveness of the algorithm in the context of static solver portfolio selection.

The existing implicit hitting set paradigm is applied to cost function and an oracle predicate. Hence, the Seesaw algorithm generalizes this by enabling the oracle to be a function. The paper identifies two independent preconditions that guarantee the correctness of the algorithm. This opens a number of avenues for future research into the possible instantiations of the algorithm, depending on the cost and oracle functions used.

2012 ACM Subject Classification Computing methodologies → Optimization algorithms

Keywords and phrases implicit hitting sets, minimal hitting set, MaxSAT, optimization

Digital Object Identifier 10.4230/LIPIcs.CP.2021.31

Funding The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902. This scientific article is part of the RICAIP project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306. This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020, the project INFOCOS with reference PTDC/CCI-COM/32378/2017 and the project DOME with reference PTDC/CCI-COM/31198/2017.

1 Introduction

Given a set of constraints, solving MaxSAT means finding a subset of given constraints under two different criteria: 1) the set must be the smallest possible 2) removing these constraints makes the whole set satisfiable. These two criteria go against each other because the fewer constraints we remove, the less likely we are to obtain satisfiability. In their seminal work, Davies and Bacchus [4] observe that MaxSAT can be solved by gradually enumerating the sets that any solution must intersect with, i.e., the solution is a hitting set of the enumerated sets. To guarantee that the smallest possible set is found, the algorithm only considers the smallest hitting sets. This style of solving is called the *implicit hitting set* algorithm. Implicit Hitting Set solving approaches have their roots in the 1980s in the theory of diagnosis [32, 33]. Since then, implicit hitting set solving and hitting set duality



© Mikoláš Janota, António Morgado, José Fragoso Santos, and Vasco Manquinho;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 31; pp. 31:1–31:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

have been successfully applied to many different problems including problems that are not necessarily in NP [6, 8, 15–17, 26, 27, 31, 35, 38]. Implicit hitting set approaches follow the common pattern: find a set of the minimal cost that satisfies a certain predicate. In the case of MaxSAT, the cost is the cardinality of the removed set and the predicate is the satisfiability of its complement.

This paper makes a crucial observation: *It is possible to extend the implicit hitting set algorithm beyond predicates.*

We show that it is possible to generalize the algorithm to minimize the cost under an objective function. We call this function the *oracle function*. This means that we are facing a *multi-objective optimization problem* with two objectives: cost and oracle. Since the objectives typically go against each other, it is generally impossible to point to a single best solution. However, we focus on *Pareto-optimal* solutions, which are solutions where improving either of the objectives requires worsening the other. The existing framework defined over predicates [16, 17, 27, 35] is subsumed by our approach because a predicate can be cast as a function that only returns either 0 or 1. Note that in the predicate-setting there are at most two Pareto-optimal solutions.

Why is this generalization useful? Consider the problem of selecting a good set of solvers, henceforth a *solver-portfolio*, for a given set of benchmarks. The selection of this set is guided by two criteria:

1. The solver-portfolio must be as small as possible, i.e., we want to use the minimum possible set of solvers.
2. The solver-portfolio's runtime in the benchmarks must be the best possible, i.e., the inclusion of more solvers means better runtime of the portfolio.

Again we have two optimization criteria going against each other and that is why our framework becomes useful in this context. In practice, we are mainly interested in the best portfolio of fixed size k ; this can be tackled by the Seesaw algorithm. The fact that the oracle function in the algorithm is treated in a black-box fashion is also important in this context. This is because the possible ways of measuring the runtime of a solver-portfolio can typically be complex and inconvenient, or even impossible, to encode in traditional exact solvers within available resources. While this alone is already an important example, similar problems often appear in practice, e.g. selecting a good set of tests for a particular software system.

The paper has the following main contributions.

- The **Seesaw Algorithm** is introduced, which extends the implicit hitting set paradigm to calculate Pareto-optimal solutions over given cost and oracle functions.
- Two independent preconditions for the algorithm's correctness are identified.
- The algorithm is implemented and evaluated on optimization of solver portfolios and strategies of real-world portfolio systems.

2 Preliminaries

Standard notions and notation for propositional logic are assumed [37]. A *literal* is a Boolean variable (x) or its negation (denoted $\neg x$); a *clause* is a disjunction of literals. A formula is in *conjunctive normal form (CNF)* if it is a conjunction of clauses.

For a CNF ϕ a subset of its clauses $\psi \subseteq \phi$ is called a *maximal satisfiable set (MSS)* of ϕ if ψ is satisfiable and there is no ψ' such that $\psi \subsetneq \psi'$ and ψ' is satisfiable. Conversely, $\psi \subseteq \phi$ is called a *minimal correction set (MCS)* of ϕ if $\phi \setminus \psi$ is satisfiable and there is no ψ' such

that $\psi' \subsetneq \psi$ and $\phi \setminus \psi'$ is satisfiable. For a CNF ϕ a subset of its clauses $\psi \subseteq \phi$ is called a *minimal unsatisfiable set (MUS)* of ϕ if ψ is unsatisfiable and there is no ψ' such that $\psi' \subsetneq \psi$ and ψ' is unsatisfiable.

The Maximum Satisfiability (MaxSAT) problem is the task of finding the smallest possible correction set, or, equivalently finding the largest maximum satisfiable set, of a given ϕ .

2.1 Functions and Predicates

► **Definition 1** (monotone). A function $f : 2^{\mathcal{U}} \rightarrow \mathbb{R}$ is monotone if and only if for any $S \subseteq S' \subseteq \mathcal{U}$ it holds that $f(S) \leq f(S')$.

► **Definition 2** (anti-monotone). A function $f : 2^{\mathcal{U}} \rightarrow \mathbb{R}$ is anti-monotone if and only if for any $S \subseteq S' \subseteq \mathcal{U}$ it holds that $f(S) \geq f(S')$.

► **Definition 3** (strictly (anti-)monotone). A function $f : 2^{\mathcal{U}} \rightarrow \mathbb{R}$ is strictly (anti-)monotone if and only if for any $S \subsetneq S' \subseteq \mathcal{U}$ it holds that $f(S) < f(S')$. (respectively, $f(S) > f(S')$).

For the purpose of this paper, we treat a *predicate* as a special case of a function that always returns either 0 or 1 representing false and true, respectively. In the context of propositional satisfiability, two predicates have special importance. Given a set of clauses S , the predicate $\text{SAT}(S)$ is true, if and only if S is satisfiable. Conversely, the predicate $\text{UNSAT}(S)$ is true, if and only if S is unsatisfiable. The predicate SAT is monotone and the predicate UNSAT is anti-monotone, but the predicates are not strictly monotone/anti-monotone. The cardinality function, $|S|$ is strictly monotone.

► **Definition 4** (Hitting sets). Let $\Gamma \subseteq 2^{\mathcal{U}}$ be a set of sets over some universe \mathcal{U} . A set $H \subseteq \mathcal{U}$ is called a *hitting set* of Γ if H has a nonempty intersection with every set of Γ . We write $\text{HS}(\Gamma)$ for the set of all hitting sets of Γ .

For an objective function $f : 2^{\mathcal{U}} \rightarrow \mathbb{R}$, a hitting set is *f-minimal* if it minimizes f over the set of all hitting sets.

2.2 Multi-objective Optimization

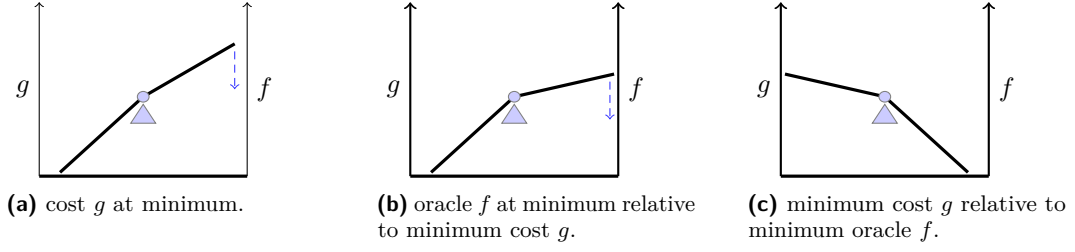
Throughout the paper we assume that any optimization function should be minimized. Under multiple optimization criteria, Pareto optimal solutions are such that improving any criterion means worsening some other. This idea is formalized by the following definitions.

► **Definition 5** (dominates). Let $O = \{f_1, \dots, f_l\}$ be a set of functions with domain D and range \mathbb{R} . We say that $x \in D$ dominates $x' \in D$, if and only if for all $f \in O$ it holds that $f(x) \leq f(x')$, and, there exists an $f \in O$ such that $f(x) < f(x')$. If x dominates x' , we write $x \prec x'$.

► **Definition 6** (Pareto-optimal). Let $O = \{f_1, \dots, f_l\}$ be a set of functions with domain D and range \mathbb{R} . We say that $x \in D$ is Pareto optimal if and only if there does not exist any other $x' \in D$ dominating x .

► **Definition 7** (Pareto frontier). For some set of optimization functions O , the Pareto frontier is the set of all Pareto-optimal individuals.

Note that even though this section refers to multi-objective optimization on a set of l functions, in this paper we consider optimizing 2 objective functions.



■ **Figure 1** Phases of the seesaw movement.

3 The Seesaw Algorithm

We are given two objective functions *cost* and *oracle*, denoted as g and f , respectively. Both functions are defined over sets of sets of some universe \mathcal{U} . We assume that we are able to minimize the cost function g using a dedicated solver, such as MaxSAT, or Integer Linear Programming (ILP) solver. The oracle function f is used as a black box.

The overall objective is to find the Pareto frontier of $\{g, f\}$. For this purpose we introduce the *Seesaw Algorithm*. The algorithm enables exploring the Pareto frontier using the implicit hitting set paradigm, as long as the functions fulfill certain properties, which are investigated later on. The algorithm is *any-time* in the sense that it may be stopped before the whole Pareto frontier is explored. The stopping criterion depends on the concrete problem at hand.

As a visual aid consider a seesaw where the middle is not completely rigid (Figure 1). This means that it is sometimes possible to push down one of the ends without the other one going immediately up. The height of each of the two ends represents the value of the two respective objective functions. Our algorithm traces the movement of this seesaw and we are at the liberty of stopping whenever we like.

Since the preference is to minimize both functions, we can imagine that there is a child sitting on either of the ends being pulled down towards the optimum by gravity. The movement is such that first there is only a child on the cost-end (g), and then someone places a heavier child on the oracle-end (f).

Figure 1 illustrates some notable phases of the movement. In the beginning, the cost function g is all the way down at its absolute minimum (Figure 1a). After that, the oracle function f starts being pushed down, which eventually causes f to reach its minimum point provided that g is maintained at its minimum (Figure 1b). This is the first Pareto-optimal point that the algorithm visits. After this phase, the cost function g leaves the ground and starts increasing, while the oracle function f continues to decrease. The movement terminates once f reaches its absolute minimum, meaning the seesaw hits the ground on the right-hand side (Figure 1c). This is the last Pareto-optimal point that the algorithm visits.

Let us look at a concrete example. Consider a MaxSAT problem defined by some unsatisfiable CNF ϕ . The objective is to find a smallest $S \subseteq \phi$ such that $\phi \setminus S$ becomes satisfiable. On the left-hand side of the seesaw we have the cost function $g = |S|$, and on the right-hand side we have the oracle function as the unsatisfiability predicate over $\phi \setminus S$ (i.e., $f = \text{UNSAT}(\phi \setminus S)$). This means that if the cost g is all the way down, S is necessarily the empty set, which sends the f side to its maximum value, value 1, because $\text{UNSAT}(\phi \setminus \emptyset)$ is true. Since the oracle f can only give two possible values (0 and 1), the initial phase coincides with the middle phase. The last phase corresponds to the solution of the MaxSAT problem: The unsatisfiability predicate became false and the size of S is the smallest possible under this condition, i.e. for this problem, the solution is obtained once Phase 3 is reached.

■ **Algorithm 1** The Seesaw Algorithm.

```

1  $H_{\text{best}} \leftarrow \perp$  // best candidate so far
2  $\Gamma \leftarrow \emptyset$  // set of collected cores
3 while true do
4    $H \leftarrow \operatorname{argmin}_{H \in \text{HS}(\Gamma)} g(H)$  // find  $g$ -minimal hitting set
5   if  $H = \perp$  or stopping criterion then
6     return  $H_{\text{best}}$ 
7   if  $f(H) < f(H_{\text{best}})$  then //  $f$  upper bound bound improvement
8      $H_{\text{best}} \leftarrow H$ 
9    $\Gamma \leftarrow \Gamma \cup \{\operatorname{extractCore}(H, f(H_{\text{best}}), f)\}$  // calculate new core

```

Let us highlight several important properties of this concrete example. The cost-side g of the seesaw is easy to optimize (push down), because we have good solvers to optimize for cardinality. However, the oracle-side f represents some complex problem (satisfiability) over which we have lesser control. We follow this pattern for the rest of the paper, the cost function g may be optimized by a dedicated solver, e.g. MaxSAT, whereas the oracle function f is only queried for its value in a black-box fashion. In practice, however, for concrete applications, it of course makes sense to take advantage of whatever we know about the problem and try to steer the algorithm towards a faster improvement of the value of the oracle.

3.1 Formalizing the Algorithm

Algorithm 1 shows the pseudocode for the algorithm. The algorithm goes through a sequence of sets called *candidates*, which determine the current values of the objectives g and f . The essence of the algorithm is to improve the value of the oracle f while at the same time maintaining the smallest possible cost g . This is done by adding *necessary conditions* for the value of the oracle f to improve. Each of these conditions is recorded in the form of a set called *core*.

Throughout the course of the algorithm, all the cores are being accumulated in the variable Γ and the candidate is always chosen to be a hitting set of Γ . Hence, a core is defined as a set that any candidate improving on the value of f must necessarily intersect with, i.e., we say that the candidate “hits” all the accumulated cores.

► **Definition 8** (core). *Given an upper bound $v \in \mathbb{R}$, a set $\kappa \subseteq \mathcal{U}$ is called a core if all $H' \subseteq \mathcal{U}$ with $f(H') < v$ intersect with κ .*

In each iteration of the algorithm, first a new candidate (a hitting set of Γ) is calculated and a new core is added by invoking a dedicated function `extractCore`. New candidates are calculated by taking some g -minimal hitting set of Γ . Since this alone is NP-hard, a dedicated solver for the task is applied.

The concrete implementation of the function `extractCore` depends on the task at hand. However, two important properties need to be guaranteed by the function: 1) The returned set must prevent the current candidate solution from being found again. 2) The returned set must be a core. The first property guarantees termination and the second property guarantees that the algorithm does not exclude from search any candidates that might improve on the current value of the oracle function f .

Let us introduce definitions formalizing these properties. For the current candidate be excluded from further search, the set returned by `extractCore` must be a complement of the current candidate; otherwise, the current candidate continues to hit all cores of Γ .

► **Definition 9** (*H*-blocking set). *Given $H \subseteq \mathcal{U}$ a set $\kappa \subseteq \mathcal{U}$ is called *H*-blocking if $\kappa \subseteq (\mathcal{U} \setminus H)$.*

Whenever the current candidate is being blocked, we have to take some care as not to also block future candidates that improve the value of the oracle f . How can it happen that a future candidate is excluded from future search even if it improves on the value of f ? Each candidate is a hitting set of the current set of cores Γ and once a candidate *stops* being a hitting set of Γ , immediately, any subset of the candidate also stops being a hitting set of Γ . This is an inherent property of hitting sets and it fundamentally influences the properties and requirements of the algorithm and therefore we note this in the following observation.

► **Observation 10.** *If H is not a hitting set of some set Γ , then any subset $H' \subseteq H$ is also not a hitting set of Γ .*

Due to this property of hitting sets, we may only block a candidate if we are sure that none of its subsets improve on the value of the oracle function f . We refer to this as careful blocking, anchored in the following definition.

► **Definition 11** (careful blocking). *Given a candidate $H \subseteq \mathcal{U}$, with $v = f(H)$, a set κ carefully blocks H , if and only if, $\kappa \subseteq (\mathcal{U} \setminus H)$ and for all $H' \subseteq \mathcal{U} \setminus \kappa$ it holds that $f(H') \geq v$.*

► **Observation 12.** *Given a candidate $H \subseteq \mathcal{U}$, any set κ that carefully blocks H is a core.*

To summarize the discussion so far, in each iteration of Algorithm 1 the new set added to Γ must be chosen as a complement of the current candidate but at the same time, this can only be done if it is guaranteed that no subset of the current candidate improves on the value of the oracle function. In general, such a core may not exist. The following section investigates two separate sufficient conditions for this existence.

3.2 Conditions for Careful Blocking

We identify two independent and sufficient conditions.

1. g is strictly monotone, or
2. f is anti-monotone

This means that the algorithm behaves correctly if the cost function g strictly prefers smaller sets, or, if the oracle function f prefers larger sets (non-strictly). We remark that Saikko et al. [35] only identifies the first of the two conditions in the context of oracles being predicates; since our framework is more general, both conditions apply also to predicates.

While either of the condition is sufficient, both may hold in some instantiations of the framework. For instance, in the case of MaxSAT, the function $|H|$ is strictly monotone, and $\text{UNSAT}(\phi \setminus H)$ is anti-monotone.

In contrast, let us consider the task of finding an unsatisfiable set of some fixed cardinality $k \in \mathbb{N}$. This implies minimizing the cost function $g(H) \triangleq (k \neq |H|)$ (effectively maximizing $k = |H|$). The oracle function is $\text{SAT}(H)$ (effectively maximizing $\text{UNSAT}(H)$). In this situation, the cost function (predicate) is neither monotone nor anti-monotone, whereas the oracle function is anti-monotone.

Under either of the conditions, the weakest possible way of extracting cores is to calculate the complement of the current candidate hitting set, i.e.,

$$\text{extractCore}_w(H, v, f) \triangleq \mathcal{U} \setminus H.$$

Such a set is blocking the candidate H and we will also see that it is blocking the candidate carefully (Definition 11) whenever either of the two above conditions is satisfied. However, this default version of core extraction is way too weak since it effectively enumerates all possible sets. In the case of anti-monotone oracles, we show that we can significantly improve on that. Let us now look at these two conditions separately.

3.2.1 Strictly monotone cost g

► **Proposition 13.** *If g is strictly monotone and H is some candidate calculated throughout the course of the algorithm as a hitting set of Γ . Then any $H' \subsetneq H$ is not a hitting set of Γ .*

Proof. By contradiction assume that H' is a hitting set of Γ . Since g is strictly monotone, $g(H') < g(H)$, which is a contradiction because H is a g -minimal hitting set of Γ . ◀

► **Corollary 14.** *If g is strictly monotone, then any H can be carefully blocked.*

We observe that the requirement of *strict* monotonicity is tight. The requirement of g being monotone does not constitute a sufficient condition as shown by the following example.

► **Example 15.** Let $\mathcal{U} = \{a\}$ and $f(\emptyset) = 1$, $f(\{a\}) = 3$. Let g be constantly 0. Hence, both g and f are monotone. Since the algorithm may choose the first candidate arbitrarily, let us assume that it is $\{a\}$, the complement of which is the empty set and therefore the algorithm terminates even though the optimum of g has not been reached.

3.2.2 Anti-Monotone oracle f

For an anti-monotone f core extraction extractCore_{am} is calculated as follows:

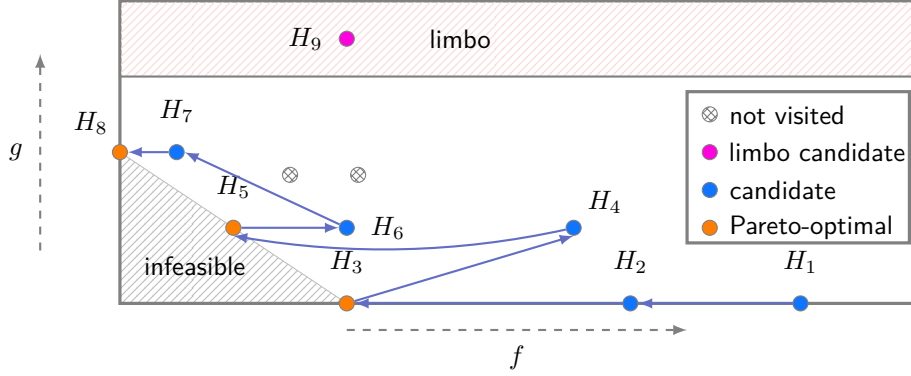
1. Non-deterministically choose H' a subset-maximal such that $H \subseteq H'$ and $f(H') \geq v$
2. **return** $\kappa \triangleq \mathcal{U} \setminus H'$.

To calculate subset maximal H' we refer to the *monotone predicates* framework [34].

► **Proposition 16.** *Let f be anti-monotone and $\kappa \triangleq \text{extractCore}_{am}(H, v, f)$. Then κ carefully blocks H .*

Proof. Since f is anti-monotone, any subset of $D \subseteq H'$ will either have the same value of f or worse. Hence, it cannot possibly improve f . This means that any solution strictly improving f is *not* be a subset of H' , i.e. it must have an intersection with $\mathcal{U} \setminus H'$. ◀

► **Corollary 17.** *If f is anti-monotone, then any H can be carefully blocked for any given upper bound v .*



■ **Figure 2** Behavior of the algorithm with respect to the Pareto frontier.

3.3 Properties

Figure 2 illustrates how the algorithm behaves in relation to the Pareto frontier of the two functions g and f . Most notably, the algorithm visits all the Pareto-optimal points in the order of the g function. However, some non-Pareto points may be visited in between. After the search has reached the optimal value of the oracle function, there are no guarantees on the obtained values. All candidates obtained after this stage are said to be in *limbo*. Ideally, we stop the algorithm before this stage, i.e., ideally, the limbo is empty.

Let us highlight some important properties of Figure 2. It starts with g being at the absolute minimum and f at some arbitrary value. After two iterations the first Pareto-optimal point is reached, where f is optimal under the condition that the cost is still minimal. Once the cost worsens, the process starts again, by looking for the next Pareto-optimal point with the smallest cost.

A special situation arises when the absolute minimum of f has been reached in the eighth iteration. In terms of the seesaw paradigm, this means that the oracle-end of the seesaw has hit the ground. After this happens, all bets are off because there are no more candidates that could possibly improve on the value of the oracle function. This means that in fact, any core is valid at this point. In particular, the algorithm may add the empty core and immediately terminate. However, for general implementation of the function `extractCore`, we do not know if such property is guaranteed. Hence, the algorithm may hopelessly try to improve on the best possible value until it runs out of possible candidates (candidates must necessarily run out because the space is finite). We show that for core extraction `extractCoream`, for anti-monotone oracle, the limbo is empty.

In the remainder of the section we focus on proving these properties. The first observation that we make is that the value of g cannot possibly improve over time because the set of possible hitting sets gradually diminishes. This means that the value of g criterion behaves anti-monotonically even if the function itself is not.

► **Proposition 18** (worsening of g). *Let H, H' be two candidates so that H was found in an earlier iteration than H' . Then, $g(H') \geq g(H)$.*

Proof. Let Γ, Γ' be the sets of cores used to calculate H, H' , respectively. Since Γ only grows over time, the set of possible hitting sets of Γ only diminishes. More precisely, since $\Gamma \subsetneq \Gamma'$, it must necessarily hold that $\text{HS}(\Gamma') \subseteq \text{HS}(\Gamma)$. From which, necessarily, $\min_g(\text{HS}(\Gamma')) \geq \min_g(\text{HS}(\Gamma))$. ◀

► **Proposition 19.** *The Seesaw algorithm visits all the Pareto-optimal points of $\{g, f\}$. Further, the points are visited in the increasing value of the function g .*

Proof sketch. Let \mathcal{P} be the set of all Pareto-optimal points that have not yet been visited and let H_P be the most recent Pareto-optimal point found. By induction we show that all elements of \mathcal{P} are hitting sets of the current Γ and we have $g(H'_P) > g(H_P)$ for all $H'_P \in \mathcal{P}$. The hypothesis is trivially true at the beginning because Γ is empty.

From the induction hypothesis and Proposition 18, all the points in \mathcal{P} have a larger or equal value of g than $g(H_P)$. Consequently, they also have a lower value of f as otherwise they would be dominated by H_P , i.e., we have $g(H'_P) > g(H_P), f(H'_P) < f(H_P)$ for $H'_P \in \mathcal{P}$. Since the algorithm only carefully blocks candidates (Definition 11), none of the hitting sets from \mathcal{P} will be blocked from future search unless visited. Since candidates are always chosen to be g -minimal, the point from \mathcal{P} to be first visited must be the one with the lowest value of g . ◀

► **Proposition 20.** *For anti-monotone f and $\text{extractCore}_{\text{am}}$ defined as above, there are no candidates in limbo.*

Proof. Let H be such that the value of f reached its maximum, meaning that $f(H) \leq f(H')$ for any $f(H')$. The procedure $\text{extractCore}_{\text{am}}$ goes on adding elements to H while the value of f does not improve but that never happens and therefore the procedure results in calculating H' as \mathcal{U} , whose complement is the empty core. Once the empty core is added to Γ , the algorithm terminates because there are no more hitting sets. ◀

4 Experimental Evaluation

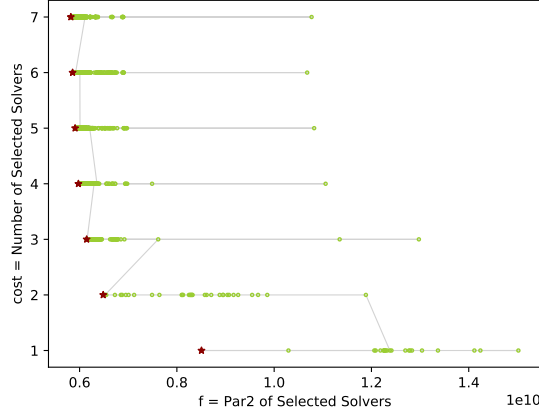
The presented Seesaw algorithm is particularly suitable for problems where the optimization function (oracle) is difficult to encode into an exact optimization solver. The problem we choose here is the selection of a portfolio of solvers, or configurations of solvers, out of a large set. There exist approaches that apply machine learning to predict the best solver, or a collection of solvers, per instance, e.g., SATzilla [42] or CPHydra [29], cf. [24]. In practice, however, solvers often employ a *static* portfolio and this is also the setting we consider for our experimental evaluation.

The problem is specified as a set of solvers $\mathcal{U} = S_1, \dots, S_n$ and a set of instances \mathcal{I} on which the solvers were run. For an instance $i \in \mathcal{I}$, we write $S(i)$ for the runtime of the solver S on this instance. If the solver does not solve the instance, $S(i)$ is some fixed penalty. In competitions, the penalty is typically chosen according to the PAR2 definition, which gives a constant penalty equal to the double of the time limit [1]. We refer to the sum of the values $S(i)$ across all instances as the *PAR2 score* of the solver. The aim is to calculate a subset of the solvers so that their PAR2 score is the smallest, when the solvers are run in parallel. More specifically, for a set of solvers $H \subseteq \mathcal{U}$ the oracle value is defined as follows.

$$\sum_{i \in \mathcal{I}} \min_{S \in H} S(i) \tag{1}$$

The oracle function is anti-monotone. Indeed, since solvers are run in parallel, when a new solver is added, the total score can only decrease or remain the same. The problem is a generalization of the classical set-cover problem [9]. For the cost function, there are two natural choices.

1. Define cost as the cardinality of the candidate, i.e., $g(H) \triangleq |H|$.
2. Define cost as the predicate $|H| = k$ for some fixed integer $k \in 1..|\mathcal{U}|$, i.e., $g(H) \triangleq (1 \text{ if } |H| = k \text{ else } 0)$.



■ **Figure 3** Pairs of values of g , f , for the SMT data set outlining a portion of the Pareto front.

In the case of cost function 1, the algorithm will explore all the Pareto front, given enough resources (see Figure 2). In the case of cost function 2, the algorithm will search for the optimal portfolio of cardinality k with respect to the PAR2 score as defined by equation (1).

Experimental Data

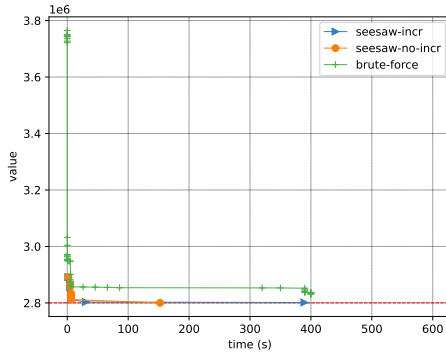
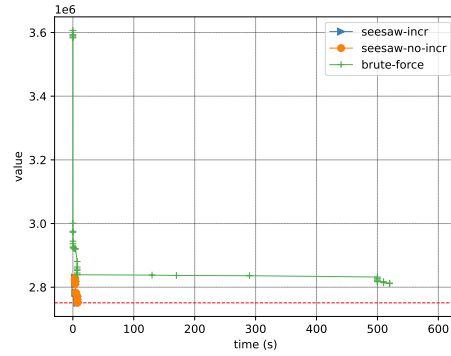
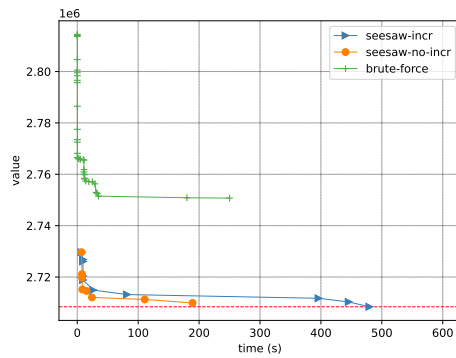
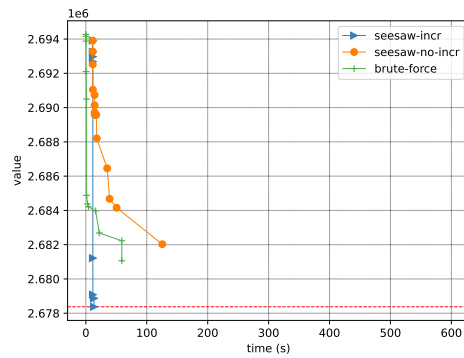
We use two data sets kindly provided by researchers in the corresponding field. The first is collected when exploring strategies for quantifier handling in the satisfiability modulo theories (SMT) solver CVC4 [2]. This exploration is motivated by the search for a set of strategies that the solver is to use in the SMT competition. This data set counts 50 different solvers and 75815 instances. We call this the *SMT dataset*.

The second data set is from the research on automated theorem provers (ATP), where a large body of strategies was considered. These were collected as follows. The various E Prover [36] configurations were invented specifically for first-order translation of Mizar Mathematical Library [41] by various methods. Specifically, they come: (1) from the E's auto-schedule mode, (2) from the system BliStrTune [19] for targeted invention of theorem proving strategies, and finally (3) from various experiments with clause selection guidance system ENIGMA [18, 20] which is based on machine learning. This data set counts 156 different solvers and 57880 instances. We call this the *ATP dataset*.

Observe that both data sets have a large number of instances, which incurs a large encoding of the oracle function if encoded explicitly. However, in the Seesaw framework, the function is only calculated on demand programmatically.

Evaluation

We have implemented Seesaw (Algorithm 1) in C++, using the Gurobi [11] solver to solve the integer linear programming subproblems. We considered two versions of the algorithm, in one of the versions Gurobi is used incrementally and in the other non-incrementally, which means all constraints need to be reloaded each time the solver is called. We also implemented a *brute-force solution* of the problem, which simply enumerates all possible subsets of cardinality k . Further, we implemented a *direct encoding* into Gurobi. However, the direct encoding into Gurobi reached memory limit in all the benchmarks and therefore was not able to solve any of the considered problems.

(a) Portfolio size 7, i.e. $k = 7$.(b) Portfolio size 16, i.e. $k = 16$.(c) Portfolio size 39, i.e. $k = 39$.(d) Portfolio size 78, i.e. $k = 78$.

■ **Figure 4** Example runs of the Seesaw algorithm for fixed portfolio size for the ATP data set.

As an additional optimization, when minimizing cores, we shuffle the elements randomly as to increase diversity in cores and therefore increase the size of the minimal hitting set. All the experiments were performed on servers with Intel(R) Xeon(R) CPU at 2.60GHz, 24 cores, 64GB RAM.

We first consider the setting where the cost function is simply the cardinality of the selected portfolio, for the first data set (SMT). The algorithm was run for 5 hours, and the worst cost reached was 7, i.e., the algorithm found all Pareto optimal points for cardinality 1..7. The run of Seesaw is plotted in Figure 3. Each pair cost (g), oracle (f) is represented by a point. Points marked with a star in dark red, correspond to Pareto points; Pareto sub-optimal points are rendered as green circles. The faded grey line, connects the points by the order in which the points are discovered (starting with a point in bottom, and stops in a point in the upper side of the plot). Observe that this figure is analogous to Figure 2.

We consider the ATP data set for fixed cardinality of the portfolio. The time limit for the algorithm was set to 600 seconds.

Since we were only able to prove optimal values for cardinalities 2 and 3, we focus here on the best value obtained by the different approaches. Figures 4a–4d show the evolution of the objective value in time for 4 different cardinalities of the portfolio ($7 \approx 5\%|\mathcal{U}|$, $16 \approx 10\%|\mathcal{U}|$, $39 \approx 25\%|\mathcal{U}|$, $78 \approx 50\%|\mathcal{U}|$). The red dashed line delimits the best value achieved amongst the solvers.

■ **Table 1** Best values achieved on samples of ATP for different cardinalities.

sample	cardinality (k)	seesaw-no-incr	seesaw-incr	brute-force
1	$k = 5$	2814287	2814191	2822009
1	$k = 13$	2764882	2764196	2760229
1	$k = 25$	2735018	2740535	2738386
2	$k = 5$	2821306	2823459	2824569
2	$k = 13$	2765332	2763383	2771172
2	$k = 25$	2730132	2729146	2732681
3	$k = 5$	2818798	2818174	2819918
3	$k = 13$	2765479	2763187	2767774
3	$k = 25$	2730116	2732724	2728360
4	$k = 5$	2827634	2830806	2825306
4	$k = 13$	2773371	2768368	2774381
4	$k = 25$	2737996	2738524	2745392
5	$k = 5$	2827598	2823750	2833257
5	$k = 13$	2756893	2763900	2775798
5	$k = 25$	2732700	2732124	2738979

First thing to observe is that there is a tendency of finding a good solution at the beginning and only improve it a little bit in the long run. From an user perspective, it means that running the algorithm longer has drastically diminishing returns.

This is quite noticeable in $k = 78$, where none of the approaches is able to improve on the value found in the first 2 minutes. Even though the non-incremental version spends much more time on reloading the constraints into Gurobi, it is not necessarily worse (see for instance $k = 39$). This suggests that the right choice of cores is the crucial ingredient in the algorithm. Indeed, in the case of $k = 78$, the incremental version found a better solution in the first 50 iterations then the non-incremental version after 5000 iterations.

In $k = 7$ and $k = 16$ all the approaches are gravitating to the same value and therefore it is possible that we are getting close to the optimal value. However, proving that the value is optimal appears to be extremely hard. Note that already $\binom{156}{16} \approx 3 \times 10^{21}$ and $\binom{156}{78} \approx 6 \times 10^{45}$.

To complement these results, we also sample the ATP dataset into 5 different subsets of size 50. The obtained results are presented in Table 1. For each considered sample and cardinality of the portfolio, the table shows the best value found by the different approaches. In the majority of cases, the Seesaw algorithm finds the best value. However, in some cases it is outperformed by brute-force. This can be explained by the fact that brute-force is able to explore much more many candidates and in some cases it may hit a good one by chance. This suggests that it would be beneficial to design a hybrid approach in order to preserve the intelligence of Seesaw but also cover more ground.

5 Related Work

5.1 Implicit Hitting sets

The implicit hitting set (IHS) approach has been successfully used in the highly competitive MaxSAT solver MaxHS [4, 5]. The idea here stems from the fact that any minimal correction set is a hitting set of all MUSes. Hence, if we enumerated all MUSes, the smallest corrections set would be obtained by calculating the minimum hitting set of those. However, the number

of MUSes may be exponential and therefore rather than enumerating all of them, MaxHS enumerates them one by one and it tests whether a correction set is obtained by picking one of the minimum hitting sets of the MUSes enumerated so far.

From the point of view of complexity theory, solving the minimum hitting set problem (MHSP) is as difficult as solving MaxSAT. However, in practice, it has been observed that state-of-the-art integer linear programming solvers perform well on MHSP. This is supported by theoretical results that show that MHSP is intractable for propositional resolution [23].

Moreno-Centeno and Karp introduce a general framework for solving NP-complete problems by the implicit hitting set paradigm and apply it to a number of problems [27]. Saikko et al. extend this framework further and observe that it is not limited to problems in NP [35]. In both aforementioned frameworks, the search for the minimal hitting set is guided by an *oracle predicate*, which effectively determines if the search should stop. The framework presented here generalizes all of the above by considering an *oracle function* rather than just a predicate. Predicates are seen as functions that either return 0 or 1. We further generalize the precondition of the algorithm by demonstrating that the algorithm does not require the cost function to be strictly monotone, as required by Saikko et al., as long as the oracle predicate/function is anti-monotone.

There is a large body of research on the use of oracles and problem decomposition. We highlight the most relevant works. The IHS-based approaches bear similarity with the *Bender's decomposition* [3] in the sense that the problem is decomposed in two different functions. Moreover, Bender's decomposition is not restricted to linear programming and it has been generalized to other optimization problems [10, 12]. Monotone predicates play a special role in our framework, which have been studied extensively in the context of SAT [34]. Our generalization of the IHS paradigm is analogous to the generalization of decision problems to function problems, such as NP to FNP [30]. Nadel employs SAT solver as an oracle in approximate optimization, similar to Walk-SAT but using SAT in each step [28].

5.2 Quantified Boolean Formulas (QBF)

In parallel to IHS, similar approaches were developed in QBF based on the AReQS paradigm [21, 22], which was further extended to QBF under optimization [14, 17]. It can be shown that this approach in fact reduces to IHS for certain QBF problems.

For illustration, consider the *smallest MUS problem*. The problem is specified by a set of clauses C_1, \dots, C_n on variables from some set X . The QBF formulation introduces *selector variables* s_1, \dots, s_n and the formula $\exists s_1, \dots, s_n \forall X \bigvee_{i \in [n]} (s_i \wedge \neg C_i)$. The formula expresses that there is a selection of clauses, determined by the selector variables, so that for any assignment to the X variables there is at least one clause that is falsified.

The task is to find a satisfying assignment for this QBF that sets to true the fewest selector variables. In the optimization AReQS paradigm, the solver collects assignments to X , each of these assignments is substituted into the formula, which results in the disjunction of selector variables of falsified clauses. Minimization is then performed gradually on the set of these disjunctions, which in fact correspond to cores in IHS. This was observed by Ignatiev et al., who carefully implement a specialized algorithm for the smallest MUS problem [16].

5.3 Multi-Objective Optimization

A large body of work exists on *Multi-Objective Combinatorial Optimization (MOCO)* and *Multi-Objective Mathematical Programming (MOMP)* [13, 40]. Multi-objective optimization is typically concerned with different strategies of navigating the Pareto frontier based on user preferences. In our setting, the way the Pareto frontier is navigated is implicit since we treat the oracle function in a black-box fashion.

Multi-objective optimization has also been studied in the context of SAT. Dedicated algorithms exist for solving MaxSAT under lexicographic preferences [25]. Pareto frontier has been explored by calculating minimal correction sets (MCSes) [39]. These approaches, however, are not immediately applicable if the functions are not encodable as propositional constraints.

A large body of work exists on the invention of good solver portfolios. These techniques rely on the combination of machine learning and constraints solving [24].

6 Conclusions and Future Work

The paper introduces and studies the Seesaw algorithm, which enables exploring the Pareto frontier using the implicit hitting set paradigm. The main strength of the algorithm is that it enables combining exact and black-box optimization. The algorithm receives as input two functions (cost and oracle), where cost is optimized exactly by a dedicated solver (e.g. ILP, MaxSAT), whereas the oracle is used in a black-box fashion.

The algorithm generalizes the existing implicit hitting set paradigm on predicates to functions. Implicit hitting sets on predicates were extensively studied in the last decade [4, 14, 16, 27, 35]. Interestingly, the framework is known to be applicable in problems that go beyond NP, which immediately implies that Seesaw algorithm also goes beyond NP; it is an open question of how to precisely characterize the complexity of the algorithm depending on the oracle used.

In our implementation we have used the commercial Gurobi solver [11]; in the future we aim to evaluate a larger set of solvers, such as MaxSAT or modern pseudo-Boolean solvers [7].

The novel Seesaw framework opens a number of opportunities for exact specialized optimization in various domains. We have demonstrated that the framework is readily usable in the context of optimizing solver portfolios and strategies. Furthermore, we aim to apply the algorithm to systematically explore the efficiency of test-quality measures.

References

- 1 Adrian Balint, Anton Belov, Matti Järvisalo, and Carsten Sinz. Overview and analysis of the SAT challenge 2012 solver competition. *Artif. Intell.*, 223, 2015. doi:10.1016/j.artint.2015.01.002.
- 2 Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification – 23rd International Conference, CAV*. Springer, 2011. doi:10.1007/978-3-642-22110-1_14.
- 3 Jacques F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Comput. Manag. Sci.*, 2(1):3–19, 2005. doi:10.1007/s10287-004-0020-y.
- 4 Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Principles and Practice of Constraint Programming - CP*. Springer, 2011. doi:10.1007/978-3-642-23786-7_19.
- 5 Jessica Davies and Fahiem Bacchus. Exploiting the power of MIP solvers in MaxSAT. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 7962. Springer, 2013. doi:10.1007/978-3-642-39071-5_13.
- 6 Erin Delisle and Fahiem Bacchus. Solving weighted CSPs by successive relaxations. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 2013.
- 7 Jan Elffers and Jakob Nordström. A cardinal improvement to pseudo-boolean solving. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence*. AAAI Press, 2020. URL: <https://aaai.org/ojs/index.php/AAAI/article/view/5508>.

- 8 Katalin Fazekas, Fahiem Bacchus, and Armin Biere. Implicit hitting set algorithms for maximum satisfiability modulo theories. In *International Joint Conference on Automated Reasoning*. Springer, 2018.
- 9 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- 10 Arthur M Geoffrion. Generalized Benders decomposition. *Journal of optimization theory and applications*, 10(4), 1972.
- 11 LLC Gurobi Optimization. Gurobi optimizer reference manual, 2021. URL: <http://www.gurobi.com>.
- 12 John N. Hooker and Greger Ottosson. Logic-based benders decomposition. *Mathematical Programming*, 96(1), 2003.
- 13 Ching-Lai Hwang and Abu Syed Md. Masud. *Multiple Objective Decision Making — Methods and Applications*. Springer Berlin Heidelberg, 1979. doi:10.1007/978-3-642-45511-7.
- 14 Alexey Ignatiev, Mikoláš Janota, and João Marques-Silva. Quantified maximum satisfiability: A core-guided approach. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 7962. Springer, 2013. doi:10.1007/978-3-642-39071-5_19.
- 15 Alexey Ignatiev, António Morgado, and João Marques-Silva. Propositional abduction with implicit hitting sets. In *ECAI 2016 - 22nd European Conference on Artificial Intelligence*. IOS Press, 2016. doi:10.3233/978-1-61499-672-9-1327.
- 16 Alexey Ignatiev, Alessandro Previti, Mark H. Liffiton, and João Marques-Silva. Smallest MUS extraction with minimal hitting set dualization. In *Principles and Practice of Constraint Programming*. Springer, 2015. doi:10.1007/978-3-319-23219-5_13.
- 17 Alexey Ignatiev, Mikoláš Janota, and João Marques-Silva. Quantified maximum satisfiability. *Constraints An Int. J.*, 21(2), 2016. doi:10.1007/s10601-015-9195-9.
- 18 Jan Jakubův, Karel Chvalovský, Miroslav Olšák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: Symbol-independent inference guiding machine (system description). In *IJCAR (2)*, volume 12167 of *Lecture Notes in Computer Science*. Springer, 2020. doi:10.1007/978-3-030-51054-1_29.
- 19 Jan Jakubův and Josef Urban. BliStrTune: hierarchical invention of theorem proving strategies. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP*. ACM, 2017. doi:10.1145/3018610.3018619.
- 20 Jan Jakubuv and Josef Urban. Hammering Mizar by learning clause guidance. In *ITP*, volume 141 of *LIPICs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
- 21 Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund Clarke. Solving QBF with counterexample guided refinement. *Artificial Intelligence*, 234, 2016. doi:10.1016/j.artint.2016.01.004.
- 22 Mikoláš Janota and Joao Marques-Silva. Abstraction-based algorithm for 2QBF. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT*. Springer, 2011. doi:10.1007/978-3-642-21581-0_19.
- 23 Stasys Jukna. Exponential lower bounds for semantic resolution. In *Proof Complexity and Feasible Arithmetics, Proceedings of a DIMACS*, volume 39. DIMACS/AMS, 1996. doi:10.1090/dimacs/039/10.
- 24 Marius Lindauer, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Selection and configuration of parallel portfolios. In *Handbook of Parallel Constraint Reasoning*. Springer, 2018. doi:10.1007/978-3-319-63516-3_15.
- 25 João Marques-Silva, Josep Argelich, Ana Graça, and Inês Lynce. Boolean lexicographic optimization: algorithms & applications. *Ann. Math. Artif. Intell.*, 62(3-4), 2011. doi:10.1007/s10472-011-9233-2.
- 26 Joao Marques-Silva, Alexey Ignatiev, and Antonio Morgado. Horn maximum satisfiability: Reductions, algorithms and applications. In *EPIA Conference on Artificial Intelligence*. Springer, 2017.

- 27 Erick Moreno-Centeno and Richard M. Karp. The implicit hitting set approach to solve combinatorial optimization problems with an application to multigenome alignment. *Oper. Res.*, 61(2), 2013. doi:10.1287/opre.1120.1139.
- 28 Alexander Nadel. On optimizing a generic function in SAT. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020*. IEEE, 2020. doi:10.34727/2020/isbn.978-3-85448-042-6_28.
- 29 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*, 2008.
- 30 Christos H. Papadimitriou. *Computational complexity*. Academic Internet Publ., 2007.
- 31 Mark Parker and Jennifer Ryan. Finding the minimum weight IIS cover of an infeasible system of linear inequalities. *Annals of Mathematics and Artificial Intelligence*, 17(1), 1996.
- 32 James A Reggia, Dana S Nau, and Pearl Y Wang. Diagnostic expert systems based on a set covering model. *International journal of man-machine studies*, 19(5), 1983.
- 33 Raymond Reiter. A theory of diagnosis from first principles. *Artificial intelligence*, 32(1), 1987.
- 34 Mikoláš Janota and João Marques-Silva. On the query complexity of selecting minimal sets for monotone predicates. *Artif. Intell.*, 233, 2016. doi:10.1016/j.artint.2016.01.002.
- 35 Paul Saikko, Johannes Peter Wallner, and Matti Järvisalo. Implicit hitting set algorithms for reasoning beyond NP. In *Principles of Knowledge Representation and Reasoning (KR)*. AAAI Press, 2016. URL: <http://www.aaai.org/ocs/index.php/KR/KR16/paper/view/12812>.
- 36 Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2-3), 2002.
- 37 João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-131.
- 38 Roni Stern, Meir Kalech, Alexander Feldman, and Gregory Provan. Exploring the duality in conflict-directed model-based diagnosis. In *AAAI Conference on Artificial Intelligence*, volume 26, 2012.
- 39 Miguel Terra-Neves, Inês Lynce, and Vasco M. Manquinho. Introducing Pareto minimal correction subsets. In *Theory and Applications of Satisfiability Testing - SAT*. Springer, 2017. doi:10.1007/978-3-319-66263-3_13.
- 40 E. L. Ulungu and J. Teghem. Multi-objective combinatorial optimization problems: A survey. *Journal of Multi-Criteria Decision Analysis*, 3(2), 1994. doi:10.1002/mcda.4020030204.
- 41 Josef Urban. MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reason.*, 37(1-2), 2006.
- 42 Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, 32, 2008. doi:10.1613/jair.2490.

Reasoning Short Cuts in Infinite Domain Constraint Satisfaction: Algorithms and Lower Bounds for Backdoors

Peter Jonsson ✉

Department of Computer and Information Science, Linköping University, Sweden

Victor Lagerkvist ✉

Department of Computer and Information Science, Linköping University, Sweden

Sebastian Ordyniak ✉

Algorithms Group, University of Sheffield, UK

Abstract

A backdoor in a finite-domain CSP instance is a set of variables where each possible instantiation moves the instance into a polynomial-time solvable class. Backdoors have found many applications in artificial intelligence and elsewhere, and the algorithmic problem of finding such backdoors has consequently been intensively studied. Sioutis and Janhunen (KI, 2019) have proposed a generalised backdoor concept suitable for infinite-domain CSP instances over binary constraints. We generalise their concept into a large class of CSPs that allow for higher-arity constraints. We show that this kind of infinite-domain backdoors have many of the positive computational properties that finite-domain backdoors have: the associated computational problems are fixed-parameter tractable whenever the underlying constraint language is finite. On the other hand, we show that infinite languages make the problems considerably harder.

2012 ACM Subject Classification Mathematics of computing → Discrete mathematics; Theory of computation → Complexity theory and logic

Keywords and phrases Constraint Satisfaction Problems, Parameterised Complexity, Backdoors

Digital Object Identifier 10.4230/LIPIcs.CP.2021.32

Funding *Peter Jonsson*: Partially supported by the Swedish Research Council (VR) under grant 2017-04112.

Victor Lagerkvist: Partially supported by the Swedish Research Council (VR) under grant 2019-03690.

Sebastian Ordyniak: Partially supported by the Engineering and Physical Sciences Research Council under grant EP/V00252X/1.

Acknowledgements We thank the anonymous reviewers for several useful comments.

1 Introduction

The *constraint satisfaction problem* (CSP) is the widely studied combinatorial problem of determining whether a set of constraints admits at least one solution. It is common to parameterise this problem by a set of relations (a *constraint language*) which determines the allowed types of constraints, and by choosing different languages one can model different types of problems. Finite-domain languages e.g. makes it possible to formulate Boolean *satisfiability* problems and *coloring* problems while infinite-domain languages are frequently used to e.g. model classical qualitative reasoning problems in artificial intelligence such as *Allen's interval algebra* and the *region-connection calculus* (RCC). Under the lens of classical complexity a substantial amount is known: every finite-domain CSP is either tractable or is NP-complete [5, 34], and for infinite domains there exists a wealth of dichotomy results separating tractable from intractable cases [1].



© Peter Jonsson, Victor Lagerkvist, and Sebastian Ordyniak;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 32; pp. 32:1–32:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The vast expressibility of infinite-domain CSPs makes the search for efficient solution methods extremely worthwhile. While worst-case complexity results indicate that many interesting problems should be insurmountably hard to solve, they are nevertheless solved in practice on a regular basis. The discrepancy between theory and practice is often explained by the existence of “hidden structure” in real-world problems [15]. If such a hidden structure exists in CSPs, then it may be exploited and offer a way of constructing improved constraint solvers. To this end, *backdoors* have been proposed as a concrete way of exploiting this structure. A backdoor represents a “short cut” to solving a hard problem instance and may be seen as a measurement for how close a problem instance is to being polynomial-time solvable [23]. The existence of a backdoor then allows one to solve a hard problem by brute forcing solutions to the (hopefully small) backdoor and then solving the resulting problems in polynomial time. This approach has been highly successful: applications can be found in e.g. (quantified) propositional satisfiability [29, 30], abductive reasoning [28], argumentation [8], planning [24], logic [26], and answer set programming [10]. Williams et al. [33] argue that backdoors may explain why SAT solvers occasionally fail to solve randomly generated instances with only a handful of variables but succeed in solving real-world instances containing thousands of variables. This argument appears increasingly relevant since modern SAT solvers frequently handle real-world instances with *millions* of variables. Might it be possible to make similar headway for infinite-domain CSP solvers? For example, can solvers in qualitative reasoning (see, e.g., Section 3.3 in [9]) be analysed in a backdoor setting? Or are the various problems under consideration so different that a general backdoor definition does not make sense?

We attack the problem from a general angle and propose a backdoor notion applicable to virtually all infinite-domain CSPs of practical and theoretical interest. Our departure is a recent paper by Sioutis and Janhunen [32] where backdoors are studied for qualitative constraint networks (which corresponds to CSPs over certain restricted sets of binary relations). We begin in Section 3 by showing why the finite-domain definition of backdoors is inapplicable in the infinite-domain setting and then continue by presenting our alternative definition, based on the idea of defining a backdoor with respect to *relationships* between variables rather than individual variables (which is the basis for the finite-domain definition [13]). We consider CSPs with respect to a fixed set of binary¹ basic relations, e.g. the basic relation in RCC-5, and then consider constraint languages definable by (not necessarily binary) first-order formulas over the basic relations. In this setting we then define a backdoor as a set of tuples of variables so that once the relationship between these variables are fixed, the resulting problem belongs to a given tractable class. If we contrast our approach with that of Sioutis and Janhunen [32], then our method is applicable to CSPs over relations of arbitrarily high arity, and we require only mild, technical assumptions on the set of binary basic relations. Crucially, Sioutis and Janhunen [32] do not consider the computational complexity of any backdoor related problems, and thus do not obtain any algorithmic results.

One of the most important properties of finite-domain backdoors is that they have desirable computational properties. Unsurprisingly, backdoor detection is NP-hard under the viewpoint of classical complexity, even for severely restricted cases. However, the situation changes if we adopt a *parameterized* complexity view. Here, the idea is to approach hard computational problems by characterizing problem parameters that can be expected to be small in applications, and then design algorithms polynomial in input size combined with a super-polynomial dependence on the parameter. We say that a problem is *fixed-*

¹ The generalisation to higher-arity relations is straightforward.

parameter tractable if its running time is bounded by $f(p) \cdot n^{O(1)}$ where n is the instance size, p the parameter, and f is a computable function. The good news is then that the backdoor detection/evaluation problem for finite-domain CSPs with a fixed finite and tractable constraint language, is *fixed-parameter tractable* (fpt) when parameterized by the size p of the backdoor [14], i.e. solvable in $f(p) \cdot n^{O(1)}$ time where n is instance size. However, if the constraint language is not finite, the basic computational problems become W[2]-hard [6]. Thus, if the backdoor size is reasonably small, which we expect for many real-world instances with hidden structure, backdoors can both be found efficiently and be used to simplify the original problem. So-called XP algorithms with a running time bounded by $n^{\text{poly}(p)}$ are polynomial-time when p is fixed, too. However, since p appears in the exponent, they become impractical when large instances are considered, and fpt algorithms are thus considered significantly better. If the constraint language is infinite, then the detection problem is not fixed-parameter tractable in general and the complexity landscape becomes more complex. Note that if the detection problem is not efficiently solvable, then the complexity of the evaluation problem is of minor importance.

While there are profound differences between finite- and infinite-domain CSPs, many important properties of backdoors fortunately remain valid when switching to the world of infinite domains. We construct algorithms (Section 4) for backdoor detection and evaluation showing that these problems are fixed-parameter tractable (with respect to the size of the backdoor) for infinite-domain CSPs based on *finite* constraint languages. Many CSPs studied in practice fulfill this condition and our algorithms are directly applicable to such problems. Algorithms for the corresponding finite-domain problems are based on enumeration of domain values. This is clearly not possible when handling infinite-domain CSPs, so our algorithms enumerate other kinds of objects, which introduces certain technical difficulties. Once we leave the safe confinement of finite languages the situation changes drastically (Section 5). We prove that the backdoor detection problem is W[2]-hard for infinite languages, making it unlikely to be fixed-parameter tractable. Importantly, our W[2]-hardness result is applicable to *all* infinite-domain CSPs where constraints are represented by first-order formulas over a fixed relational structure, meaning that it is not possible to circumvent this difficulty by targeting other classes of problems. Hence, while some cases of hardness are expected, given earlier results for satisfiability and finite-domain CSPs [15], it is perhaps less obvious that essentially all infinite-domain CSPs exhibit the same source of hardness. Again, these negative results are *not* restricted to specific problems and instead show a general difficulty in applying backdoors over infinite constraint languages. We conclude the paper with a discussion concerning future research directions (Section 6).

Throughout, some proofs have been moved to the appendix, and the affected statements are marked with an asterisk (*).

2 Preliminaries

2.1 Relations and Formulas

A *relational structure* over a set of values D (a *domain*) is a tuple $(D; R_1, \dots, R_m)$ where each R_i is a (finitary) relation over D . For simplicity we do not distinguish between the signature of a relational structure and its relations. Assume that \mathcal{R} contains binary relations over the domain D . We say that the relations in \mathcal{R} are *jointly exhaustive* (JE) if $\bigcup \mathcal{R} = D^2$, and that they are *pairwise disjoint* (PD) if $R \cap R' = \emptyset$ for all distinct $R, R' \in \mathcal{R}$. Additionally, a constraint language which is both JE and PD is said to be JEPD.

Let $\varphi(x_1, \dots, x_n)$ be a first-order formula (with equality) over free variables x_1, \dots, x_n over a relational structure $\Gamma = (D; R_1, \dots, R_m)$. We write $\text{Sol}(\varphi(x_1, \dots, x_n))$ for the set of models of $\varphi(x_1, \dots, x_n)$ with respect to x_1, \dots, x_n , i.e., $(d_1, \dots, d_n) \in \text{Sol}(\varphi(x_1, \dots, x_n))$ if and only if $(D; R_1, \dots, R_m) \models \varphi(d_1, \dots, d_n)$, and we use the notation $R(x_1, \dots, x_n) \equiv \varphi(x_1, \dots, x_n)$ to define R as $\text{Sol}(\varphi(x_1, \dots, x_n))$. In this case, we say that R is *first-order definable* (fo-definable) in Γ . In addition to first-order logic, we sometimes use the quantifier-free (qffo), the primitive positive (pp), and the quantifier-free primitive positive (qfpp) fragments. The qffo fragment consists of all formulas without quantifiers, the pp fragment consists of formulas that are built using existential quantifiers, conjunction and equality, and the qfpp consists of quantifier-free pp formulas. We lift the notion of definability to these fragments in the obvious way. It is important to note that if R is pp-definable in Γ , then R is not necessarily qfpp-definable in Γ even if Γ admits quantifier elimination.

If the structure Γ admits quantifier elimination (i.e. every first-order formula has a logically equivalent formula without quantifiers), then fo-definability coincides with qffo-definability. This is sometimes relevant in the sequel since our results are mostly based on qffo-definability. There is a large number of structures admitting quantifier elimination and interesting examples are presented in every standard textbook on model theory, cf. Hodges [17]. Well-known examples include *Allen's interval algebra* (under the standard representation via intervals in \mathbb{Q}) and the spatial formalisms RCC-5 and RCC-8 (under the model-theoretically pleasant representation suggested by Bodirsky & Wöfl [4]). Some general quantifier elimination results that are highly relevant for computer science and AI are discussed in Bodirsky [1, Sec. 4.3.1].

2.2 The Constraint Satisfaction Problem

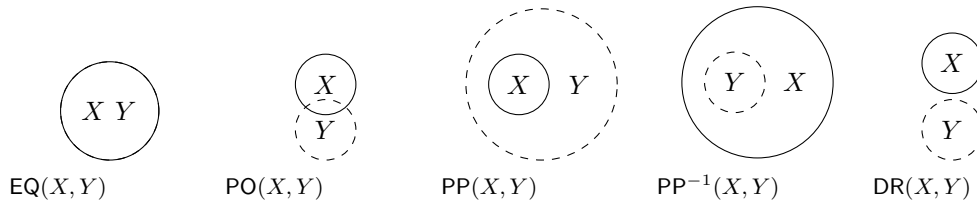
Turning to computability, a *constraint language*, or simply language, over a domain D is a set of relations Γ_D over D . The *constraint satisfaction problem* over a constraint language Γ_D ($\text{CSP}(\Gamma_D)$) is then the computational problem of determining whether a set of constraints over Γ_D admits at least one satisfying assignment.

$\text{CSP}(\Gamma_D)$

Input: A tuple (V, C) where V is a set of variables and C a set of constraints of the form $R(x_1, \dots, x_k)$, where $R \in \Gamma_D$ and $x_1, \dots, x_k \in V$.
 Question: Does there exists a satisfying assignment to (V, C) , i.e., a function $f: V \rightarrow D$ such that $(f(x_1), \dots, f(x_k)) \in R$ for each constraint $R(x_1, \dots, x_k) \in C$?

We write $\text{Sol}(I)$ for the set of all satisfying assignments to a $\text{CSP}(\Gamma)$ instance I . Finite-domain constraints admit a simple representation obtained by explicitly listing all tuples in the involved relation. For infinite domain CSPs, it is frequently assumed that Γ is a *first-order reduct* of an underlying relational structure \mathcal{R} , i.e., each $R \in \Gamma$ is fo-definable in \mathcal{R} . Whenever \mathcal{R} admits quantifier elimination, then we can always work with the *qffo reduct* where each $R \in \Gamma$ is qffo-definable in \mathcal{R} .

► **Example 1.** An *equality* language is a first-order reduct of a structure $(D; \emptyset)$ where D is a countably infinite domain. Each literal in a first-order formula over this structure is either of the form $x = y$, or $x \neq y \equiv \neg(x = y)$. The structure $(D; \emptyset)$ admits quantifier elimination so every first-order reduct can be viewed as a qffo reduct. For example, if we let S be defined via the formula $(x = y \wedge x \neq z) \vee (x \neq y \wedge y = z)$ then $\text{CSP}(\{S\})$ is known to be NP-complete [3]. On the other hand, $\text{CSP}(\{=, \neq\})$ is well-known to be tractable.



■ **Figure 1** Illustration of the basic relations of RCC-5 with two-dimensional disks.

A *temporal language* is a first-order reduct of $(\mathbb{Q}; <)$. The structure $(\mathbb{Q}; <)$ admits quantifier elimination so it is sufficient to consider qffo reducts. For example, the *betweenness relation* (Betw) can be defined via the formula $(x < y \wedge y < z) \vee (z < y \wedge y < x)$, and the resulting CSP is well-known to be NP-complete, due to Bodirsky & Kára [3].

It will occasionally be useful to assume that the underlying relational structure is JEPD. Clearly, neither $(\mathbb{N}; \emptyset)$, nor $(\mathbb{Q}; <)$ are JEPD, but they can easily be expanded to satisfy the JEPD condition by (1) adding the converse of each relation, and (2) adding the complement of each relation. Thus, an equality language can be defined as a first-order reduct of $(\mathbb{N}; =, \neq)$, and a temporal language as a first-order reduct of $(\mathbb{Q}; =, <, >)$. More ideas for transforming non-JEPD languages into JEPD languages can be found in [2, Sec. 4.2].

Constraint languages in this framework also capture many problems of particular interest in artificial intelligence. For example, consider the *region connection calculus* with the 5 basic relations $\Theta = \{\text{DR}, \text{PO}, \text{PP}, \text{PP}^{-1}, \text{EQ}\}$ (RCC-5). See Figure 1 for a visualisation of these relations. In the traditional formulation of this calculus one then allows unions of the basic relations, which (for two regions X and Y) e.g. allows us to express that X is a proper part of Y or X and Y are equal. This relation can easily be defined via the (quantifier-free) first-order formula $(x\text{PP}y) \vee (x\text{EQ}y)$. Hence, if we let $\Theta^{\vee=}$ be the constraint language consisting of all unions of basic relations in Θ , then $\Theta^{\vee=}$ is a qffo reduct of Θ .

2.3 Parameterized Complexity

To analyse complexity of CSPs we use the framework of *parameterized complexity* [7, 11] where the run-time of an algorithm is studied with respect to a parameter $p \in \mathbb{N}$ and the input size n . Given an instance I of some computational problem, we let $\|I\|$ denote the bit-size of I . Many important CSPs are NP-hard on general instances and are regarded as being theoretically intractable. However, realistic problem instances are not chosen arbitrarily and they often contain structure that can be exploited for solving the instance efficiently. The idea behind parameterized analysis is that the parameter describes the structure of the instance in a computationally meaningful way. The result is a fine-grained complexity analysis that is more relevant to real-world problems while still admitting a rigorous theoretical treatment including, for instance, algorithmic performance guarantees.

The most favourable complexity class is **FPT** (*fixed-parameter tractable*) which contains all problems that can be decided in $f(p) \cdot n^{O(1)}$ time, where f is a computable function. However, parameterised complexity offers strong theoretical evidence that inclusion in **FPT** is highly unlikely for some problem, e.g. those that are hard for the class **W[1]**. The latter contains all problems that admit many-to-one reductions from the **PARAMETERISED CLIQUE** problem, which asks whether a graph has a clique of size p , where p is the parameter, and the reduction runs in fixed-parameter time with respect to p .

We will prove that certain problems are not in **FPT** and this requires some machinery. A *parameterized problem* is, formally speaking, a subset of $\Sigma^* \times \mathbb{N}$ where Σ is the input alphabet. Reductions between parameterized problems need to take the parameter into

account. To this end, we will use *parameterized reductions* (or fpt-reductions). Let L_1 and L_2 denote parameterized problems with $L_1 \subseteq \Sigma_1^* \times \mathbb{N}$ and $L_2 \subseteq \Sigma_2^* \times \mathbb{N}$. A parameterized reduction from L_1 to L_2 is a mapping $P : \Sigma_1^* \times \mathbb{N} \rightarrow \Sigma_2^* \times \mathbb{N}$ such that (1) $(x, k) \in L_1$ if and only if $P((x, k)) \in L_2$, (2) the mapping can be computed by an fpt-algorithm with respect to the parameter k , and (3) there is a computable function $g : \mathbb{N} \rightarrow \mathbb{N}$ such that for all $(x, k) \in L_1$ if $(x', k') = P((x, k))$, then $k' \leq g(k)$. The class $W[1]$ contains all problems that are fpt-reducible to Independent Set when parameterized by the size of the solution, i.e. the number of vertices in the independent set. Showing $W[1]$ -hardness (by an fpt-reduction) for a problem rules out the existence of a fixed-parameter algorithm under the standard assumption $FPT \neq W[1]$.

3 Backdoors

This section is devoted to the motivation behind and the introduction of a general backdoor concept for CSPs.

3.1 Motivation

We begin by recapitulating the standard definition of backdoors for finite-domain CSPs. Let $\alpha : X \rightarrow D$ be an assignment. For a k -ary constraint $c = R(x_1, \dots, x_k)$ we denote by $c|_\alpha$ the constraint over the relation R_0 and with scope X_0 obtained from c as follows: R_0 is obtained from R by (1) removing (d_1, \dots, d_k) from R if there exists $1 \leq i \leq k$ such that $x_i \in X$ and $\alpha(x_i) \neq d_i$, and (2) removing from all remaining tuples all coordinates d_i with $x_i \in X$. The scope X_0 is obtained from x_1, \dots, x_k by removing every $x_i \in X$. For a set C of constraints we define $C|_\alpha$ as $\{c|_\alpha : c \in C\}$. We now have everything in place to define the standard notion of a (strong) backdoor, in the context of Boolean satisfiability problems and finite-domain CSPs.

► **Definition 2** ([13, 33]). *Let \mathcal{H} be a set of CSP instances. A \mathcal{H} -backdoor for a CSP(Γ_D) instance (V, C) is a set $B \subseteq V$ where $(V \setminus B, C|_\alpha) \in \mathcal{H}$ for each $\alpha : B \rightarrow D$.*

In practice, \mathcal{H} is typically defined as a polynomial-time solvable subclass of CSP and one is thus interested in finding a backdoor into the tractable class \mathcal{H} . If the CSP instance I has a backdoor of size k , then it can be solved in $|D|^k \cdot \text{poly}(|I|)$ time. This is an exponential running time with the advantageous feature that it is exponential not in the instance size $|I|$, but in the domain size and backdoor set size only.

► **Example 3.** Let us first see why Definition 2 is less impactful for infinite-domain CSPs. Naturally, the most obvious problem is that one, even for a fixed $B \subseteq V$, need to consider infinitely many functions $\alpha : V \rightarrow D$, and there is thus no general argument which resolves the backdoor evaluation problem. However, even for a fixed assignment $\alpha : V \rightarrow D$ we may run into severe problems. Consider a single equality constraint of the form $(x = y)$ and an assignment α where $\alpha(x) = 0$ but where α is not defined on y . Then $(x = y)|_\alpha = \{(0)\}$, i.e., the constant 0 relation, which is *not* an equality relation. Similarly, consider a constraint XrY where r is a basic relation in RCC-5. Regardless of r , assigning a fixed region to X but not to Y results in a CSP instance which is not included in *any* tractable subclass of RCC-5 (and is not even an RCC-5 instance).

Hence, the usual definition of a backdoor fails to compensate for a fundamental difference between finite and infinite-domain CSPs: that assignments to variables are typically much less important than the *relation* between variables.

3.2 Basic Definitions and Examples

Recall that we in the infinite setting are mainly interested in $\text{CSP}(\Gamma)$ problems where each relation in Γ is qffo-definable over a fixed relational structure \mathcal{R} . Hence, in the backdoor setting we obtain three components: a relational structure \mathcal{R} and two qffo reducts \mathcal{S} and \mathcal{T} over \mathcal{R} , where $\text{CSP}(\mathcal{S})$ is the (likely NP-hard) problem which we want to solve by finding a backdoor to the (likely tractable) problem $\text{CSP}(\mathcal{T})$. Additionally, we will assume that \mathcal{R} only consists of binary JEPD relations and that the equality relation is qffo-definable in \mathcal{R} .

► **Definition 4.** Let $\mathcal{R} = (D; R_1, R_2, \dots)$ be a relational structure where the relations are binary and JEPD, the equality relation on D is qffo-definable in \mathcal{R} , and let \mathcal{S} and \mathcal{T} be two qffo reducts of \mathcal{R} . We say that $[\mathcal{S}, \mathcal{T}, \mathcal{R}]$ is a language triple and we refer to

- \mathcal{S} as the source language,
- \mathcal{T} as the target language, and
- \mathcal{R} as the base language.

Note that \mathcal{S} and \mathcal{T} may contain non-binary relations even though \mathcal{R} only contains binary relations. One should note that all concepts work equivalently well for higher arity relations in \mathcal{R} but it complicates the presentation. Also note that the equality relation needs to be qffo-definable in \mathcal{R} since this relation is always available in first-order formulas. An alternative way is to require that the equality relation is a member of \mathcal{R} but this assumption is stronger than is needed for our purposes (see Example 8 for an example).

We begin by describing how constraints can be simplified in the presence of a partial assignment of relations from \mathcal{R} to pairs of variables.

► **Definition 5.** Let $[\mathcal{S}, \mathcal{T}, \mathcal{R}]$ be a language triple and let (V, C) be an instance of $\text{CSP}(\mathcal{S})$. Say that a partial mapping $\alpha: B \rightarrow \mathcal{R}$ for $B \subseteq V^2$ is consistent if the $\text{CSP}(\mathcal{R})$ instance

$$(V, \{R(x, y) \mid x, y \in V, \alpha(x, y) \text{ is defined}, R = \alpha(x, y)\})$$

is satisfiable. We define a reduced constraint with respect to a consistent α as:

$$R(x_1, \dots, x_k)_{|\alpha} = R(x_1, \dots, x_k) \wedge \bigwedge_{\alpha(x_i, x_j) = S, x_i, x_j \in \{x_1, \dots, x_k\}} S(x_i, x_j).$$

Next, we describe how reduced constraints can be translated to the target language. Let $[\mathcal{S}, \mathcal{T}, \mathcal{R}]$ be a language triple and let $a \in \mathbb{N} \cup \{\infty\}$ equal $\sup\{i \mid R \in \mathcal{S} \text{ has arity } i\}$. Let

$$\mathbf{S} = \{\text{Sol}(R(x_1, \dots, x_k)_{|\alpha}) \mid R \in \mathcal{S}, \alpha: \{x_1, \dots, x_k\}^2 \rightarrow \mathcal{R}\}$$

and

$$\mathbf{T} = \{\varphi(x_1, \dots, x_k) \mid k \leq a, \varphi(x_1, \dots, x_k) \text{ is a qfpp-definition over } \mathcal{T}\}.$$

We interpret these two sets as follows. Each mapping $\alpha: \{x_1, \dots, x_k\}^2 \rightarrow \mathcal{R}$ applied to a constraint $R(x_1, \dots, x_k)$ results in a (potentially) simplified constraint $R(x_1, \dots, x_k)_{|\alpha}$, which might or might not be expressible via a $\text{CSP}(\mathcal{T})$ instance. Then the condition that $\text{Sol}(R(x_1, \dots, x_k)_{|\alpha}) \in \mathbf{S}$ is qfpp-definable over \mathcal{S} simply means that the set of models of the constraint $R(x_1, \dots, x_k)_{|\alpha}$ can be defined as the set of models of a $\text{CSP}(\mathcal{S})$ instance. Thus, the set \mathbf{S} represents all possible simplifications of constraints (with respect to \mathcal{S}) and the set \mathbf{T} represents all possibilities of expressing constraints (up to a fixed arity) by the language \mathcal{T} . Crucially, note that \mathbf{S} and \mathbf{T} are finite whenever \mathcal{S} and \mathcal{T} are finite. With the help of the two sets \mathbf{S} and \mathbf{T} we then define the following method for translating (simplified) \mathcal{S} -constraints into \mathcal{T} -constraints.

► **Definition 6.** A simplification map is a partial mapping Σ from \mathbf{S} to \mathbf{T} such that for every $R \in \mathbf{S}$: $\Sigma(R) = \varphi(x_1, \dots, x_k)$ if $R(x_1, \dots, x_k) \equiv \varphi(x_1, \dots, x_k)$ for some $\varphi(x_1, \dots, x_k) \in \mathbf{T}$, and is undefined otherwise.

We typically say that a simplification map goes from the source language \mathcal{S} to the target language \mathcal{T} even though it technically speaking is a map from \mathbf{S} to \mathbf{T} . Note that if \mathcal{S} and \mathcal{T} are both finite, then there always exists a simplification map of finite size, and one may without loss of generality assume that it is possible to access the map in constant time. We will take a closer look at the computation of simplification maps for finite language in Section 4.1. If \mathcal{S} is infinite, then the situation changes significantly. First of all, a simplification map has an infinite number of inputs and we cannot assume that it is possible to access it in constant time. We need, however, always assume that it can be accessed in polynomial time. Another problem is that we have no general way of computing simplification maps so they need to be constructed on a case-by-case basis.

► **Definition 7.** Let $[S, \mathcal{T}, \mathcal{R}]$ be a language triple, and let Σ be a simplification map from \mathcal{S} to \mathcal{T} . For an instance (V, C) of $\text{CSP}(\mathcal{S})$ we say that $B \subseteq V^2$ is a backdoor if, for every consistent $\alpha: B \rightarrow \mathcal{R}$, $\Sigma(R(x_1, \dots, x_k)|_\alpha)$ is defined for every constraint $R(x_1, \dots, x_k) \in C$.

Before turning to computational aspects of finding and using backdoors, let us continue by providing additional examples, starting with finite-domain languages.

► **Example 8.** Let $D = \{1, \dots, d\}$ for some $d \in \mathbb{N}$ and define the relational structure $\mathbf{D} = (D; R_{ij} \mid 1 \leq i, j \leq d)$ where $R_{ij} = \{(i, j)\}$. This structure consists of binary JEPD relations and the equality relation on D is qffo-definable in \mathbf{D} via

$$x =_D y \equiv R_{11}(x, y) \vee R_{22}(x, y) \vee \dots \vee R_{dd}(x, y).$$

Note that *any* constraint language Γ with domain D can be viewed as a first-order reduct over \mathbf{D} . Hence, a backdoor in the style of Definition 2 is a special case of Definition 7, meaning that our backdoor notion is not merely an adaptation of the finite-domain concept, but a strict generalisation, since we allow arbitrary binary relations (and not only unary relations) in the underlying relational structure.

► **Example 9.** Consider equality languages, i.e. languages that are fo-definable over the base structure $(\mathbb{N}; =, \neq)$. Recall the NP-hard ternary relation S from Example 1 and consider a simplification map Σ with respect to the tractable target language $\{=, \neq\}$. Note that we *cannot* simplify an arbitrary constraint $S(x, y, z)$, but that we can simplify $S(x, y, z)|_\alpha$ if (e.g.) $\alpha(x, y)$ is '=', or if $\alpha(x, y)$ is \neq . Let (V, C) be an instance of $\text{CSP}(\{S\})$. Consider the set $B = \{(x, y) \mid S(x, y, z) \in C\} \subseteq V^2$. We claim that B is a backdoor with respect to $\{=, \neq\}$. Let $\alpha: B \rightarrow \{=, \neq\}$, and consider an arbitrary constraint $S(x, y, z) \in C$. Clearly, $(x, y) \in B$. Then, regardless of the relation between x and y , the constraint can be removed and replaced by $\{=, \neq\}$ -constraints.

► **Example 10.** Recall the definitions of Θ and $\Theta^{\vee=}$ for RCC-5 from Section 2. Consider a reduced constraint $R|_\alpha(x, y)$ with respect to an instance (V, C) of $\text{CSP}(\Theta^{\vee=})$, a set $B \subseteq V^2$, and a function $\alpha: B \rightarrow \Theta$. If $(x, y) \in B$ (or, symmetrically, $(y, x) \in B$) then $R(x, y) \wedge (\alpha(x, y))(x, y)$ is (1) unsatisfiable if $\alpha(x, y) \cap R = \emptyset$, or (2) equivalent to $\alpha(x, y)$. Hence, the simplification map in this case either outputs an unsatisfiable $\text{CSP}(\Theta)$ instance or replaces the constraints with the equivalent constraint over a basic relation. This results in an $O(5^{|B|}) \cdot \text{poly}(|I|)$ time algorithm for RCC-5, which can slightly be improved to $O(4^{|B|}) \cdot \text{poly}(|I|)$ with the observation that only the trivial relation $(\text{DRUPOUPPUPP}^{-1} \cup \text{EQ})$ contains all the five basic relations.

We now have a working backdoor definition for infinite-domain CSPs, but it remains to show that they actually simplify CSP solving, and that they can be found efficiently. We study such computational aspects in the following section.

4 Algorithms for Finite Languages

This section is divided into three parts where we analyse various computational problems associated with backdoors.

4.1 Computing Simplification Maps

We discussed (in Section 3.2) the fact that a simplification map $\mathcal{S} \rightarrow \mathcal{T}$ always exists when \mathcal{S} and \mathcal{T} are finite languages. How to compute such a map is an interesting question in its own right. In the finite-domain case, the computation is straightforward (albeit time-consuming) since one can enumerate all solutions to a CSP instance in finite time. This is clearly not possible when the domain is infinite. Thus, we introduce a method that circumvents this difficulty by enumerating other objects than concrete solutions.

Assume that $[\mathcal{S}, \mathcal{T}, \mathcal{R}]$ is a language triple. We first make the following observation concerning relations that are qffo-defined in some binary and JEPD relational structure \mathcal{R} (for additional details, see e.g. Sec. 2.2. in Lagerkvist & Jonsson [19]). If R is qffo-defined in \mathcal{R} , then it can be defined by a DNF \mathcal{R} -formula that involves only positive (i.e. negation-free) atomic formulas of type $R(\bar{x})$, where R is a relation in \mathcal{R} : every atomic formula $\neg R(x, y)$ can be replaced by

$$\bigvee_{S \in \mathcal{R} \setminus \{R\}} S(x, y)$$

and the resulting formula being transformed back to DNF.

Let $I = (V, C)$ denote an arbitrary instance of, for instance, $\text{CSP}(\mathcal{S})$. An \mathcal{R} -certificate for I is a satisfiable instance $\mathcal{C} = (V, C')$ of $\text{CSP}(\mathcal{R})$ that *implies* every constraint in C , i.e. for every $R(v_1, \dots, v_k)$ in C , there is a clause in the definition of this constraint (as a DNF \mathcal{R} -formula) such that all literals in this clause are in C' . It is not difficult to see that I has a solution if and only if I admits an \mathcal{R} -certificate (see, for instance, Theorem 6 by Jonsson and Lagerkvist [19] for a similar result). Hence, we will sometimes also say that an \mathcal{R} -certificate \mathcal{C} of a $\text{CSP}(\mathcal{S})$ instance I *satisfies* I . We will additionally use *complete certificates*: a $\text{CSP}(\cdot)$ instance is *complete* if it contains a constraint over every 2-tuple of (not necessarily distinct) variables, and a certificate is complete if it is a complete instance of $\text{CSP}(\cdot)$.

► **Example 11.** Consider the structure $(\mathbb{Q}; <, >, =)$, i.e. the rationals under the natural ordering. Let $B = \{(x, y, z) \in \mathbb{Q}^3 \mid x < y < z \vee z < y < x\}$. Let $I = (\{x, y, z, w\} \mid \{B(x, y, z), B(y, z, w)\})$ be an instance of $\text{CSP}(\{B\})$. The instance I is satisfiable and this is witnessed by the solution $f(x) = 0, f(y) = 1, f(z) = 2, f(w) = 3$. A certificate for this instance is $\{x < y, y < z, z < w\}$ and a complete certificate is

$$\begin{aligned} x < y, \quad x < z, \quad x < w, \quad y < z, \quad y < w, \quad z < w, \\ y > x, \quad z > x, \quad w > x, \quad z > y, \quad w > y, \quad w > z, \\ x = x, \quad y = y, \quad z = z, \quad w = w. \end{aligned}$$

We first show that satisfiable CSP instances always have complete certificates under fairly general conditions. Furthermore, every solution is covered by at least one such certificate.

► **Lemma 12.** (*) Assume \mathcal{R} is JEPD and that Γ is qffo-definable in \mathcal{R} . An instance (V, C) of $\text{CSP}(\Gamma)$ has a solution $f: V \rightarrow D$ if and only if there exists a complete \mathcal{R} -certificate for I that has solution f .

We use the previous lemma for proving that two instances I_s and I_t have the same solutions if and only if they admit the same complete certificates.

► **Lemma 13.** (*) Let $[\mathcal{S}, \mathcal{T}, \mathcal{R}]$ be a language triple such that \mathcal{S} , \mathcal{T} , and \mathcal{R} are finite. Given instances $I_s = (V, C)$ of $\text{CSP}(\mathcal{S})$ and $I_t = (V, C')$ of $\text{CSP}(\mathcal{T})$, the following are equivalent:

1. $\text{Sol}(I_s) = \text{Sol}(I_t)$ and
2. I_s and I_t have the same set of complete \mathcal{R} -certificates.

Finally, we present our method for computing simplification maps.

► **Lemma 14.** (*) Let X be the set of language triples $[\mathcal{S}, \mathcal{T}, \mathcal{R}]$ that enjoy the following properties:

1. \mathcal{S} , \mathcal{T} , and \mathcal{R} are finite and
2. $\text{CSP}(\mathcal{R})$ is decidable.

The problem of constructing simplification maps for members of X is computable.

4.2 Backdoor Evaluation

We begin by studying the complexity of the following problem, which intuitively, says to which degree the existence of a backdoor helps to solve the original problem.

$[\mathcal{S}, \mathcal{T}, \mathcal{R}]$ -BACKDOOR EVALUATION

Input: A CSP instance (V, C) of $\text{CSP}(\mathcal{S})$ and a backdoor $B \subseteq V^2$ into $\text{CSP}(\mathcal{T})$.

Question: Is (V, C) satisfiable?

Clearly, $[\mathcal{S}, \mathcal{T}, \mathcal{R}]$ -BACKDOOR EVALUATION is in many cases NP-hard: simply pick a language \mathcal{S} such that $\text{CSP}(\mathcal{S})$ is NP-hard. Note that one, strictly speaking, is not forced to use the backdoor when solving the $[\mathcal{S}, \mathcal{T}, \mathcal{R}]$ -BACKDOOR EVALUATION problem, but if the size of the backdoor is sufficiently small then we may be able to solve the instance faster via the backdoor. Indeed, as we will now prove, the problem is in FPT for finite languages when parameterised by the size of the backdoor.

► **Theorem 15.** $[\mathcal{S}, \mathcal{T}, \mathcal{R}]$ -BACKDOOR EVALUATION is in FPT when parameterised by the size of the backdoor, if \mathcal{S} , \mathcal{T} , and \mathcal{R} are finite and $\text{CSP}(\mathcal{T})$ and $\text{CSP}(\mathcal{R})$ are tractable.

Proof. Let $\Sigma: \mathcal{S} \rightarrow \mathcal{T}$ be a simplification map that has been computed off-line, let $I = (V, C)$ be an instance of $\text{CSP}(\mathcal{S})$, let $B \subseteq V^2$ be a backdoor of size k , and let $m = |\mathcal{R}|$. Then, we claim that I is satisfiable if and only if there is a consistent assignment $\alpha: B \rightarrow \mathcal{R}$ such that the $\text{CSP}(\mathcal{T})$ instance $I|_\alpha = (V, \{\Sigma(c|_\alpha) \mid c \in C\})$ is satisfiable.

Forward direction. Assume that I is satisfiable. Since \mathcal{R} is JEPD, and since \mathcal{S} is qffo-definable in \mathcal{R} , we know from Lemma 12 that I admits a complete certificate (V, \hat{C}) . For every pair $(x, y) \in B$ then define α to agree with the complete certificate (V, \hat{C}) , i.e., $\alpha(x, y) = S$ for $S(x, y) \in \hat{C}$. Naturally, α is consistent since (V, \hat{C}) is a complete certificate for I , and since B is a backdoor set it also follows that the $\text{CSP}(\mathcal{T})$ instance $(V, \{\Sigma(c|_\alpha) \mid c \in C\})$ is

well-defined. Pick an arbitrary constraint $\Sigma(R(x_1, \dots, x_{\text{ar}(R)}))|_\alpha$. It follows (1) that (V, \widehat{C}) satisfies $R(x_1, \dots, x_{\text{ar}(R)})$, and (2) that if $\alpha(x_i, x_j) = S$ for $x_i, x_j \in \{x_1, \dots, x_{\text{ar}(R)}\}$ then (V, \widehat{C}) satisfies $S(x_i, x_j)$, meaning that (V, \widehat{C}) satisfies

$$R(x_1, \dots, x_{\text{ar}(R)}) \wedge \bigwedge_{\alpha(x_i, x_j)=S, x_i, x_j \in \{x_1, \dots, x_{\text{ar}(R)}\}} S(x_i, x_j),$$

and hence also $\Sigma(R(x_1, \dots, x_{\text{ar}(R)}))|_\alpha$, since Σ is a simplification map.

Backward direction. Assume that there exists a consistent $\alpha: B \rightarrow \mathcal{R}$ such that $(V, \{\Sigma(c|_\alpha) \mid c \in C\})$ is satisfiable, and let (V, \widehat{C}) be a complete certificate witnessing this. Naturally, for any pair $(x, y) \in B$ it must then hold that $S(x, y) \in \widehat{C}$ for $\alpha(x, y) = S$, since (V, \widehat{C}) could not be a complete certificate otherwise. Pick a constraint $R(x_1, \dots, x_{\text{ar}(R)}) \in C$, and let $\Sigma(R(x_1, \dots, x_{\text{ar}(R)}))|_\alpha \equiv \varphi(x_1, \dots, x_{\text{ar}(R)})$ for some $\varphi(x_1, \dots, x_{\text{ar}(R)}) \in \mathbf{T}$. It follows that (V, \widehat{C}) satisfies

$$\varphi(x_1, \dots, x_{\text{ar}(R)})$$

and since

$$\text{Sol}(\varphi(x_1, \dots, x_{\text{ar}(R)})) = \text{Sol}(R(x_1, \dots, x_{\text{ar}(R)}))|_\alpha$$

it furthermore follows that $R(x_1, \dots, x_{\text{ar}(R)})|_\alpha$ must be satisfied, too. However, since

$$R(x_1, \dots, x_{\text{ar}(R)})|_\alpha \equiv R(x_1, \dots, x_{\text{ar}(R)}) \wedge \bigwedge_{\alpha(x_i, x_j)=S, x_i, x_j \in \{x_1, \dots, x_{\text{ar}(R)}\}} S(x_i, x_j),$$

and since every constraint $S(x_i, x_j)$ is clearly satisfied, it must also be the case that (V, \widehat{C}) is a complete certificate of I .

Put together, it thus suffices to enumerate all m^k choices for α and to check whether α is consistent and whether $I|_\alpha$ is satisfiable. $\text{CSP}(\mathcal{R})$ is tractable so checking whether α is consistent can be done in polynomial time. Moreover, using the simplification map Σ , we can reduce $I|_\alpha$ to an instance of $\text{CSP}(\mathcal{T})$, which can be solved in polynomial-time. The total running time is $O(m^k \cdot \text{poly}(\|I\|))$. ◀

4.3 Backdoor Detection

Theorem 15 implies that small backdoors are desirable since they can be used to solve CSP problems faster. Therefore, let us now turn to the problem of finding backdoors. The basic backdoor detection problem is defined as follows.

$[S, \mathcal{T}, \mathcal{R}]$ -BACKDOOR DETECTION

Input: A CSP instance (V, C) of $\text{CSP}(\mathcal{S})$ and an integer k .
 Question: Does (V, C) have a backdoor B into \mathcal{T} of size at most k ? (and if so output such a backdoor)

The problem is easily seen to be NP-hard even when \mathcal{S} and \mathcal{T} are finite; we will provide a proof of this in Corollary 20. We will now prove that the problem can be solved efficiently if the size of the backdoor is sufficiently small.

► **Theorem 16.** $[S, \mathcal{T}, \mathcal{R}]$ -BACKDOOR DETECTION is in FPT when parameterized by k , if \mathcal{S} , \mathcal{T} and \mathcal{R} are finite, and $\text{CSP}(\mathcal{T})$ and $\text{CSP}(\mathcal{R})$ are tractable.

Proof. Let $I = ((V, C), k)$ be an instance of $[\mathcal{S}, \mathcal{T}, \mathcal{R}]$ -BACKDOOR DETECTION, let a be the maximum arity of any constraint of I , and let Σ be a simplification map from \mathcal{S} to \mathcal{T} which we assume has been computed off-line. We solve $((V, C), k)$ using a bounded depth search tree algorithm as follows.

We construct a search tree T , for which every node is labeled by a set $B \subseteq V^2$ of size at most k . Additionally, every leaf node has a second label, which is either YES or NO. T is defined inductively as follows. The root of T is labeled by the empty set. Furthermore, if t is a node of T , whose first label is B , then the children of t in T are obtained as follows. If for every consistent assignment $\alpha: B \rightarrow \mathcal{R}$, where $\mathcal{R} = \{R_1, \dots, R_m\}$, and every $c \in C$, we have that $\Sigma(c|_\alpha)$ is defined, then B is a backdoor into \mathcal{T} of size at most k and therefore t becomes a leaf node, whose second label is YES. Otherwise, i.e., if there is a consistent assignment $\alpha: B \rightarrow \mathcal{R}$ and a constraint $c \in C$ such that $\Sigma(c|_\alpha)$ is not defined, we distinguish two cases: (1) $|B| = k$, then t becomes a leaf node, whose second label is NO, and (2) $|B| < k$, then for every pair p of variables in the scope of c with $p \notin B$, t has a child whose first label is $B \cup \{p\}$.

If T has a leaf node, whose second label is YES, then the algorithm returns the first label of that leaf node. Otherwise the algorithm return NO. This completes the description of the algorithm.

We now show the correctness of the algorithm. First, suppose the search tree T built by the algorithm has a leaf node t whose second label is YES. Here, the algorithm returns the first label, say B of t . By definition, we obtain that B is a backdoor into \mathcal{T} of size at most k .

Now consider the case where the algorithm returns NO. We need to show that there is no backdoor set B into \mathcal{T} with $|B| \leq k$. Assume, for the sake of contradiction that such a set B exists.

Observe that if T has a leaf node t whose first label is a set B' with $B' \subseteq B$, then the second label of t must be YES. This is because, either $|B'| < k$ in which case the second label of t must be YES, or $|B'| = k$ in which case $B' = B$ and by the definition of B it follows that the second label of t must be YES.

It hence remains to show that T has a leaf node whose first label is a set B' with $B' \subseteq B$. This will complete the proof about the correctness of the algorithm. We will show a slightly stronger statement, namely, that for every natural number ℓ , either T has a leaf whose first label is contained in B or T has an inner node of distance exactly ℓ from the root whose first label is contained in B . We show the latter by induction on ℓ .

The claim obviously holds for $\ell = 0$. So assume that T contains a node t at distance ℓ from the root of T whose first label, say B' , is a subset of B . If t is a leaf node of T , then the claim is shown. Otherwise, there is a consistent assignment $\alpha': B' \rightarrow \mathcal{R}$ and a constraint $c \in C$ such that $\Sigma(c|_{\alpha'})$ is not defined.

Let $\alpha: B \rightarrow \mathcal{R}$ be any consistent assignment of the pairs in B that agrees with α' on the pairs in B' . Then, $\Sigma(c|_\alpha)$ is defined because B is a backdoor set into \mathcal{T} . By definition of the search tree T , t has a child t' for every pair p of variables in the scope of some constraint $c \in C$ such that $\Sigma(c|_{\alpha'})$ is not defined. We claim that B contains at least one pair of variables within the scope of c . Indeed, suppose not. Then $\Sigma(c|_\alpha) = \Sigma(c|_{\alpha'})$ and this contradicts our assumption that $\Sigma(c|_\alpha)$ is defined. This concludes our proof concerning the correctness of the algorithm.

The running time of the algorithm is obtained as follows. Let T be a search tree obtained by the algorithm. Then the running time of the depth-bounded search tree algorithm is $O(|V(T)|)$ times the maximum time that is spend on any node of T . Since the number of children of any node of T is bounded by $\binom{a}{2}$ (recall that a is the maximum arity of any

constraint of (V, C)) and the longest path from the root of T to some leaf of T is bounded by $k + 1$, we obtain that $|V(T)| \leq \mathcal{O}(\binom{a}{2}^{k+1})$. Furthermore, the time required for any node t of T is at most $\mathcal{O}(m^k |C| \cdot \text{poly}(|I|))$ (where the polynomial factors stems from checking whether α is consistent). Therefore we obtain $\mathcal{O}(\binom{a}{2}^{k+1} m^k |C|)$ as the total run-time of the algorithm showing that $[\mathcal{S}, \mathcal{T}, \mathcal{R}]$ -BACKDOOR DETECTION is FPT when parameterized by k . ◀

5 Hardness Results for Infinite Languages

Our positive FPT results are mainly restricted to finite languages. In Section 5.1, we investigate how the situation differs for infinite languages, and will see that finiteness is not merely a simplifying assumption, but in many cases absolutely crucial for tractability. We remind the reader that if the source language is infinite, then there are an infinite number of possible inputs for the simplification map, and this implies that it is not necessarily accessible in polynomial time. However, we will see that simplification maps with good computational properties do exist in certain cases. Even under this assumption, we prove that the backdoor detection problem is in general $W[2]$ -hard. We do not study the backdoor evaluation problem since the hardness of backdoor detection makes the evaluation problem less interesting.

5.1 Hardness of Backdoor Detection

We begin by establishing the existence of a relation which turns out to be useful as a gadget in the forthcoming hardness reduction. For every $k \geq 2$, we let the k -ary equality relation R_k be defined as follows:

$$R_k(x_1, \dots, x_k) \equiv \bigwedge_{i, j, l, m \in [k] \text{ with } i \neq j \text{ and } l \neq m} (x_i \neq x_j \vee x_l = x_m)$$

► **Lemma 17.** (*) *The following holds for every $k \geq 2$.*

1. $R_k(x_1, \dots, x_k)$ cannot be written as a conjunction of binary equality relations $x_i = x_j$ and $x_i \neq x_j$, and
2. for every pair i, j with $1 \leq i < j \leq k$ and every assignment α of (x_i, x_j) to $\{(x_i = x_j, x_i \neq x_j)\}$, it holds that $R_k \wedge \alpha((x_i, x_j))$ can be written as a conjunction of binary equality relations.

Moreover, the definition of R_k can be computed in time k^4 .

Let $\mathcal{S}_e = \{R_i \mid i \geq 1\}$ where R_i is defined as in Lemma 17 and let $\mathcal{T}_e = \{=, \neq\}$. Note that both \mathcal{S}_e and \mathcal{T}_e are equality languages so they are qffo reducts of $\mathcal{R}_e = \{=, \neq\}$. We first verify that \mathcal{S}_e , despite being infinite, admits a straightforward simplification map to the target language $\mathcal{T}_e = \{=, \neq\}$.

► **Lemma 18.** (*) *There is a simplification map Σ_e from \mathcal{S}_e to \mathcal{T}_e that can be accessed in polynomial time.*

Our reduction is based on the following problem.

HITTING SET

Input: A finite set U , a family \mathcal{F} of subsets of U , and an integer $k \geq 0$.
 Question: Is there a set $S \subseteq U$ of size at most k such that $S \cap F \neq \emptyset$ for every $F \in \mathcal{F}$?

Hitting Set is NP-hard even if the sets in \mathcal{F} are restricted to sets of size 2: in this case, the problem is simply the Vertex Cover problem. Furthermore, Hitting Set is W[2]-hard when parameterized by k [7] but this does not hold if the sets in \mathcal{F} have size bounded by some constant.

► **Theorem 19.** $[\mathcal{S}_e, \mathcal{T}_e, \mathcal{R}_e]$ -BACKDOOR DETECTION is W[2]-hard when parameterised by the size of the backdoor.

Proof. We give a parameterized reduction from the Hitting Set problem. Given an instance (U, \mathcal{F}, k) of Hitting set, let (V, C) be the CSP(\mathcal{S}_e) instance with $V = U \cup \{n\}$ having one constraint C_F for every $F \in \mathcal{F}$, whose scope is $F \cup \{n\}$ and whose relation is $R_{|F|+1}$ (as defined in connection with Lemma 17). This can easily be accomplished in polynomial time. Next, we verify that (U, \mathcal{F}, k) has a hitting set of size at most k if and only if (V, C) has a backdoor set of size at most k into CSP(\mathcal{T}_e).

Forward direction. Let S be a hitting set for \mathcal{F} . We claim that $B = \{(n, s) | s \in S\}$ is a backdoor set into CSP(\mathcal{T}_e). Because S is a hitting set for \mathcal{F} , B contains at least two variables from the scope of every constraint in C . Let $\alpha : B \rightarrow \{=, \neq\}$ be an arbitrary consistent assignment. Arbitrarily choose a constraint $R_k(x_1, \dots, x_k)$ in C . By the construction of the simplification map (Lemma 18), it follows that $\Sigma_e(R_k(x_1, \dots, x_k)|_\alpha)$ is defined so B is indeed a backdoor.

Backward direction. Let B be a backdoor set for (V, C) into CSP(\mathcal{T}_e). Note first that we can assume that $b = (x, n)$ or $b = (n, x)$ for every $b \in B$. To see this, note that if this is not the case for some $b \in B$, then we can replace one of the variables in b with n , while still obtaining a backdoor set, since it is sufficient to fix a single relation between pairs of variables in R_k in order to simplify to CSP(\mathcal{T}_e). We claim that $(\bigcup_{b \in B} b) \setminus \{n\}$ is a hitting set for \mathcal{F} . This is clearly the case because for every constraint in C , there must be at least one pair $b \in B$ such that both variables in b are in the scope of the constraint. Otherwise, there would exist a constraint whose simplification is the constraint itself, and such a constraint cannot be expressed as a conjunction of $\{=, \neq\}$ constraints, due to the first condition of Lemma 17. ◀

One may note that CSP(\mathcal{S}_e) is polynomial-time solvable and that the $[\mathcal{S}_e, \mathcal{T}_e, \mathcal{R}_e]$ -BACKDOOR DETECTION problem is thus computationally harder than the CSP problem that we attempt to solve with the backdoor approach. This indicates that the backdoor approach must be used with care and it is, in particular, important to know the computational complexity of the CSPs under consideration. Certainly, there are also examples of infinite source languages with an NP-hard CSP such that backdoor detection is W[2]-hard. For instance, let $\mathcal{S}'_e = \mathcal{S} \cup \{S\}$ where S is the relation defined in Example 1 – it follows immediately that CSP(\mathcal{S}'_e) is NP-hard. Furthermore, it is not hard to verify that Lemma 18 can be extended to the source language \mathcal{S}'_e so the proof of Theorem 19 implies W[2]-hardness of $[\mathcal{S}'_e, \mathcal{T}_e, \mathcal{R}_e]$ -BACKDOOR DETECTION, too.

Finally, we can now answer the question (that was raised in Section 4.3) concerning the complexity of $[\mathcal{S}, \mathcal{T}, \mathcal{R}]$ -BACKDOOR DETECTION when \mathcal{S} and \mathcal{T} are finite. By observing that the reduction employed in Theorem 19 is a polynomial-time reduction from Hitting Set and using the fact that Hitting Set is NP-hard even if all sets have size at most 2, we obtain the following result.

► **Corollary 20.** The problem $[\{R_3\}, \mathcal{T}_e, \mathcal{S}_e]$ -BACKDOOR DETECTION is NP-hard.

6 Concluding Remarks

We have generalised the backdoor concept to CSPs over infinite domains and we have presented parameterized complexity results for infinite-domain backdoors. Interestingly, despite being a strict generalisation of finite-domain backdoors, both backdoor detection and evaluation turned out to be in FPT. Hence, the backdoor paradigm is applicable to infinite-domain CSPs, and, importantly, it is indeed possible to have a uniform backdoor definition (rather than having different definitions for equality languages, temporal languages, RCC-5, and so on). Let us now discuss a few different directions for future research.

Backdoor detection and evaluation for infinite languages

Our results show that there is a significant difference between problems based on finite constraint languages and those that are based on infinite languages. The backdoor detection and evaluation problems are fixed-parameter tractable when the languages are finite. In the case of infinite languages, we know that the backdoor detection problem is $W[2]$ -hard for certain choices of languages. This raises the following question: for which infinite source languages is backdoor detection fixed-parameter tractable? This question is probably very hard to answer in its full generality so it needs to be narrowed down in a suitable way. A possible approach is to begin by studying this problem for equality languages.

Broader tractable classes

Recent advances concerning backdoor sets for SAT and finite-domain CSP provide a rather large number of promising and important research directions for future work. For instance, Gaspers et al. [13] have introduced the idea of so-called *heterogeneous* backdoor sets, i.e. backdoor sets into the disjoint union of more than one base language, and Ganian et al. [12] have exploited the idea that if variables in the backdoor set separate the instance into several independent components, then the instance can still be solved efficiently as long as each component is in some tractable base class. Both of these approaches significantly enhance the power and/or generality of the backdoor approach for finite-domain CSP and there is a good chance that these concepts can also be lifted to infinite-domain CSPs. Another promising direction for future research is the use of decision trees (or the even more general concept of *backdoor DNFs*) for representing backdoors [27, 31]. Here the idea is to use decision trees or backdoor DNFs as a compact representation of all (partial) assignments of the variables in the backdoor set. This can lead to a much more efficient algorithm for backdoor evaluation since instead of considering all assignments of the backdoor variables, one only needs to consider a potentially much smaller set of partial assignments of those variables that (1) cover all possible assignments and (2) for each partial assignment the reduced instance is in the base class. It has been shown that this approach may lead to an exponential improvement of the backdoor evaluation problem in certain cases, and it has been verified experimentally that these kinds of backdoors may be substantially smaller than the standard ones [27, 31].

Another direction is to drop the requirement that backdoors move the instance to a polynomial-time solvable class – it may be sufficient that the class is solvable in, say, single-exponential $2^{O(n)}$ time. This can lead to substantial speedups when considering CSPs that are not solvable in $2^{O(n)}$ time. Natural classes of this kind are known to exist under the exponential-time hypothesis [20], and concrete examples are given by certain extensions of Allen’s algebra that are not solvable in $2^{o(n \log n)}$ time.

The complexity of simplifying constraints

We have presented an algorithm for constructing simplification maps that works under the condition that the source and target languages are finite. However, we have no general method for computing simplification maps for infinite languages. It seems conceivable that the computation of simplification maps is an undecidable problem and proving this is an interesting research direction. However, could it still make sense to allow suboptimal simplification maps which are oblivious to certain types of constraints, but which can be computed more efficiently? Or simplification maps where not all entries are polynomial time accessible? Thus, the general problem which we want to solve is, given a relation represented by a first-order formula over \mathcal{R} (i.e., corresponding to a simplified constraint) we wish to determine whether it is possible to find a $\text{CSP}(\mathcal{T})$ instance whose set of models coincides with this relation. This problem is in the literature known as an *inverse constraint satisfaction problem* over a constraint language \mathcal{T} ($\text{Inv-CSP}(\mathcal{T})$), and may be defined as follows.

$\text{Inv-CSP}(\mathcal{T}, \mathcal{R})$

Input: A relation R (represented by an fo-formula over \mathcal{R}).

Question: Can R be defined as the set of models of a $\text{CSP}(\mathcal{T})$ instance?

We are thus interested in finding polynomial-time solvable cases of this problem, since this would imply the existence of an efficiently computable simplification map to \mathcal{T} even if the source language is infinite. The Inv-CSP problem has been fully classified for the Boolean domain [22, 25], but little is known for arbitrary finite domains, and even less has been established for the infinite case. We suspect that obtaining such a complexity classification is a very hard problem even for restricted language classes such as equality languages. One of the reasons for this is the very liberal way that the input is represented. If one changes the representation, then a complexity classification may be easier to obtain. A plausible way of doing this is to restrict ourselves to ω -categorical base structures. The concept of ω -categoricity plays a key role in the study of complexity aspects of CSPs [1], but it is also important from an AI perspective [16, 18, 21]. Examples of such structures include all structures with a finite domain and many relevant infinite-domain structures such as $(\mathbb{N}; =)$, $(\mathbb{Q}; <)$, and the standard structures underlying formalisms such as Allen’s algebra and RCC. For ω -categorical base structures \mathcal{R} , each fo-definable relation R can be partitioned into a finite number of equivalence classes with respect to the automorphism group of \mathcal{R} , and this gives a much more restricted way of representing the input. We leave this as an interesting future research project.

References

- 1 M. Bodirsky. *Complexity of Infinite-Domain Constraint Satisfaction*. Cambridge University Press, 2021. Preprint available from <https://www.math.tu-dresden.de/~bodirsky/Book.pdf>.
- 2 M. Bodirsky and P. Jonsson. A model-theoretic view on qualitative constraint reasoning. *Journal of Artificial Intelligence Research*, 58:339–385, 2017. doi:10.1613/jair.5260.
- 3 M. Bodirsky and J. Kára. The complexity of temporal constraint satisfaction problems. *Journal of the ACM*, 57(2):9:1–9:41, 2010.
- 4 Manuel Bodirsky and Stefan Wöfl. RCC8 is polynomial on networks of bounded treewidth. In *Proc. of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-2011)*, pages 756–761, 2011.
- 5 A. Bulatov. A dichotomy theorem for nonuniform CSPs. In *Proc. 58th Annual Symposium on Foundations of Computer Science (FOCS-2017)*, pages 319–330, 2017.

- 6 C. Carbonnel, M. C. Cooper, and E. Hebrard. On backdoors to tractable constraint languages. In Barry O’Sullivan, editor, *Principles and Practice of Constraint Programming*, pages 224–239, Cham, 2014. Springer International Publishing.
- 7 R. Downey and M. Fellows. *Parameterized complexity*. Monographs in Computer Science. Springer, 1999. URL: <http://books.google.se/books?id=pt5QAAAAMAAJ>.
- 8 W. Dvorák, S. Ordyniak, and S. Szeider. Augmenting tractable fragments of abstract argumentation. *Artificial Intelligence*, 186:157–173, 2012.
- 9 F. Dylla, J. Lee, T. Mossakowski, T. Schneider, A. Van Delden, J. Van De Ven, and D. Wolter. A survey of qualitative spatial and temporal calculi: Algebraic and computational properties. *ACM Computing Surveys*, 50(1):7:1–7:39, 2017.
- 10 J. Fichte and S. Szeider. Backdoors to tractable answer set programming. *Artificial Intelligence*, 220:64–103, 2015.
- 11 J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, 2006.
- 12 Robert Ganian, M. S. Ramanujan, and Stefan Szeider. Discovering archipelagos of tractability for constraint satisfaction and counting. *ACM Trans. Algorithms*, 13(2):29:1–29:32, 2017.
- 13 S. Gaspers, N. Misra, S. Ordyniak, S. Szeider, and S. Živný. Backdoors into heterogeneous classes of SAT and CSP. *Journal of Computer and System Sciences*, 85:38–56, 2017. doi: 10.1016/j.jcss.2016.10.007.
- 14 S. Gaspers, S. Ordyniak, and S. Szeider. Backdoor sets for CSP. In *The Constraint Satisfaction Problem: Complexity and Approximability*, volume 7 of *Dagstuhl Follow-Ups*, pages 137–157. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.
- 15 S. Gaspers and S. Szeider. Backdoors to satisfaction. In *The Multivariate Algorithmic Revolution and Beyond - Essays Dedicated to Michael R. Fellows on the Occasion of His 60th Birthday*, pages 287–317. Springer, 2012.
- 16 Robin Hirsch. Relation algebras of intervals. *Artificial intelligence*, 83(2):267–295, 1996.
- 17 Wilfrid Hodges. *Model theory*. Cambridge University Press, 1993.
- 18 Jinbo Huang. Compactness and its implications for qualitative spatial and temporal reasoning. In *Proc. 13th International Conference on Principles of Knowledge Representation and Reasoning (KR-2012)*, 2012.
- 19 P. Jonsson and V. Lagerkvist. An initial study of time complexity in infinite-domain constraint satisfaction. *Artificial Intelligence*, 245:115–133, 2017. doi:10.1016/j.artint.2017.01.005.
- 20 P. Jonsson and V. Lagerkvist. Why are CSPs based on partition schemes computationally hard? In *43rd International Symposium on Mathematical Foundations of Computer Science (MFCS-2018)*, pages 43:1–43:15, 2018.
- 21 Peter Jonsson. Constants and finite unary relations in qualitative constraint reasoning. *Artificial Intelligence*, 257:1–23, 2018.
- 22 D. Kavvadias and M. Sideri. The inverse satisfiability problem. *SIAM Journal on Computing*, 28:152–163, 1998.
- 23 P. Kilby, J. Slaney, S. Thiébaux, and T. Walsh. Backbones and backdoors in satisfiability. In *Proc. 20th National Conference on Artificial Intelligence (AAAI-2005)*, page 1368–1373, 2005.
- 24 M. Kronegger, S. Ordyniak, and A. Pfandler. Backdoors to planning. *Artificial Intelligence*, 269:49–75, 2019.
- 25 V. Lagerkvist and B. Roy. Complexity of inverse constraint problems and a dichotomy for the inverse satisfiability problem. *Journal of Computer and System Sciences*, 117:23–39, 2021.
- 26 A. Meier, S. Ordyniak, M. Ramanujan, and I. Schindler. Backdoors for linear temporal logic. *Algorithmica*, 81(2):476–496, 2019.
- 27 Sebastian Ordyniak, André Schidler, and Stefan Szeider. Backdoor DNFs. In *Proc. 28th of the International Joint Conference on Artificial Intelligence (IJCAI-2021)*, 2021. To appear. Report version available from <https://www.ac.tuwien.ac.at/files/tr/ac-tr-21-001.pdf>.
- 28 A. Pfandler, S. Rümmele, and S. Szeider. Backdoors to abduction. In *Proc. 23rd International Joint Conference on Artificial Intelligence (IJCAI-2013)*, pages 1046–1052, 2013.

- 29 M. Samer and S. Szeider. Backdoor sets of quantified boolean formulas. *Journal of Automated Reasoning*, 42(1):77–97, 2009.
- 30 M. Samer and S. Szeider. Fixed-parameter tractability. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 425–454. IOS Press, 2009.
- 31 Marko Samer and Stefan Szeider. Backdoor trees. In *Proc. 23rd AAAI Conference on Artificial Intelligence (AAAI-2008)*, pages 363–368, 2008.
- 32 M. Sioutis and T. Janhunen. Towards leveraging backdoors in qualitative constraint networks. In *Proc. 42nd German Conference on AI (KI-2019)*, pages 308–315, 2019.
- 33 R. Williams, C. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI-2003)*, pages 1173–1178, 2003.
- 34 D. Zhuk. A proof of the CSP dichotomy conjecture. *Journal of the ACM*, 67(5):30:1–30:78, 2020. doi:10.1145/3402029.

A Additional Proofs for Section 4

A.1 Proof of Lemma 12

Proof. Let $I = (V, C)$ be an arbitrary instance of $\text{CSP}(\Gamma)$.

Assume $\mathcal{C} = (V, \widehat{C})$ is a complete \mathcal{R} -certificate for I with a solution $f: V \rightarrow D$. The certificate $\mathcal{C} = (V, \widehat{C})$ implies every constraint in C . Arbitrarily choose a constraint $R(v_1, \dots, v_k)$ in C . There is a clause in the definition of this constraint (viewed as a DNF \mathcal{R} -formula) such that all literals in this clause are in \widehat{C} . This implies that $(f(v_1), \dots, f(v_k)) \in R$ since f is a solution to \mathcal{C} . We conclude that f is a solution to I since $R(v_1, \dots, v_k)$ was chosen arbitrarily.

Assume $f: V \rightarrow D$ is a solution to I . We know that \mathcal{R} is JEPD. We construct a complete certificate $\mathcal{C} = (V, \widehat{C})$ such that f is a solution to \mathcal{C} . Consider a 2-tuple of (not necessarily distinct) variables (v, v') where $\{v, v'\} \subseteq V$. The tuple $(f(v), f(v'))$ appears in exactly one relation R in \mathcal{R} since \mathcal{R} is JEPD. Add the constraint $R(v, v')$ to \widehat{C} . Do the same thing for all 2-tuples of variables. The resulting instance \mathcal{C} is complete and it is satisfiable since f is a valid solution. ◀

A.2 Proof of Lemma 13

Proof. Arbitrarily choose an instance $I_s = (V, C_s)$ of $\text{CSP}(\mathcal{S})$ and an instance $I_t = (V, C_t)$ of $\text{CSP}(\mathcal{T})$.

Assume that I_s and I_t have the same set of complete \mathcal{R} -certificates. Arbitrarily choose a solution $f: V \rightarrow D$ to I_s that is not a solution to I_t (the other direction is analogous). There is a complete \mathcal{R} -certificate \mathcal{C} for I_s such that f is a solution to \mathcal{C} by Lemma 12. We know that \mathcal{C} is a certificate for I_t so Lemma 12 implies that f is a solution to I_t , too. This leads to a contradiction.

Assume that I_s and I_t have the same set of solutions. Assume \mathcal{C} is a complete \mathcal{R} -certificate for I_s but not for I_t (the other way round is analogous). By Lemma 12, every solution to \mathcal{C} is a solution to I_s . Since I_s and I_t have the same set of solutions, \mathcal{C} is a complete \mathcal{R} -certificate for I_t , too, which leads to a contradiction. ◀

A.3 Proof of Lemma 14

Proof. Arbitrarily choose $[\mathcal{S}, \mathcal{T}, \mathcal{R}]$ in X . Recall the definitions of \mathbf{S} and \mathbf{T} that were made in connection with Definition 6. Arbitrarily choose a relation $R \in \mathbf{S}$ with arity k and define $I_s = (V, C) = (\{v_1, \dots, v_k\}, \{R(v_1, \dots, v_k)\})$. Given a k -ary formula $\varphi \in \mathbf{T}$, let $I_t = (V, \varphi(v_1, \dots, v_k))$. Then, the following are equivalent

- (a) $\text{Sol}(I_s) = \text{Sol}(I_t)$,
- (b) I_s and I_t have the same set of complete \mathcal{R} -certificates

by Property 1 combined with Lemma 13. Property 2 implies that condition (a) is decidable: there is a straightforward algorithm based on enumerating all complete \mathcal{R} -certificates. Compute every possible complete \mathcal{R} -certificate on the variables in V and check whether I_s and I_t are equisatisfiable on these certificates. Recall that checking if a certificate implies the constraints in I_s and I_t is a decidable problem since $\text{CSP}(\mathcal{R})$ is decidable. This procedure can be performed in a finite number of steps since the number of complete \mathcal{R} -certificates on variable set V is finite.

With this in mind, there is an algorithm that computes a simplification map $\Sigma: \mathcal{S} \rightarrow \mathcal{T}$. Arbitrarily choose a k -ary relation R in \mathbf{S} and let $I_s = (V, C_s) = (\{v_1, \dots, v_k\}, \{R(v_1, \dots, v_k)\})$. Enumerate all $I_t = (V, \varphi(v_1, \dots, v_k))$ where $\varphi \in \mathbf{T}$ is k -ary. If there exists an I_t that satisfies condition (a), then let $\Sigma(R) = \varphi$, and, otherwise, let $\Sigma(R)$ be undefined. We know that testing condition (a) is decidable by Property 2 and we know that \mathbf{S} and \mathbf{T} are finite sets, so Σ can be computed in a finite number of steps. \blacktriangleleft

B Additional Proofs for Section 5.1

B.1 Proof of Lemma 17

Proof. For proving the first statement, we begin by showing that $(a_1, \dots, a_k) \in R_k$ if either (1) $a_1 = a_2 = \dots = a_k$ or (2) $a_i \neq a_j$ for every i and j with $i \neq j$. Assume this is not the case. Then there are $i, j, m, l \in [k]$ with $i \neq j$ and $m \neq l$ such that $a_i = a_j$ and $a_l \neq a_m$. But then the term $(x_i \neq x_j \vee x_l = x_m)$ in the definition of R_k is not satisfied by (a_1, \dots, a_k) . Now, consider a conjunction ϕ of atomic formulas from the set $\{R_i(x_i, x_j) \mid R_i \in \{=, \neq\}, i, j \in [k]\}$. If each atomic formula in ϕ is of the type $x_i = x_j$, then the models of ϕ cannot correctly define R_k : ϕ is not satisfied by any assignment where all variables are assigned distinct values. Similarly, if there exists an atomic formula of the type $x_i \neq x_j$ in ϕ , then ϕ cannot be satisfied by an assignment where all variables are assigned the same value. Hence, R_k cannot be defined as a conjunction of binary equality constraints.

For the second statement, let α be an assignment of a pair (x_i, x_j) to either $x_i \neq x_j$ or $x_i = x_j$. We observe the following.

- if $\alpha(x_i, x_j) = (x_i = x_j)$, then $R_k(x_1, \dots, x_k) \wedge \alpha(x_i, x_j)$ is logically equivalent to the formula $(x_1 = x_2) \wedge (x_1 = x_3) \wedge \dots \wedge (x_1 = x_k)$. The definition of R_k contains the clauses $(x_i \neq x_j \vee x_l = x_m)$ for all $1 \leq l \neq m \leq k$. Since $x_i \neq x_j$ does not hold due to $\alpha(x_i, x_j)$, it follows that all variables must be assigned the same value.
- if $\alpha(x_i, x_j) = (x_i \neq x_j)$, then $R_k(x_1, \dots, x_k) \wedge \alpha(x_i, x_j)$ is logically equivalent to the conjunction of $(x_i \neq x_j)$ for every $i, j \in [k]$ where $i \neq j$. The definition of R_k contains the clauses $(x_i \neq x_j \vee x_l = x_m)$ for all $1 \leq l \neq m \leq k$. Since $x_i = x_j$ does not hold due to $\alpha(x_i, x_j)$, it follows that all variables must be assigned distinct values.

We finally note that the definition of R_k can easily be computed in k^4 time so R_k satisfies the statement of the lemma. \blacktriangleleft

B.2 Proof of Lemma 18

Proof. Consider $\Sigma_e(R_k(x_1, \dots, x_k)|_\alpha)$. If $|\{x_1, \dots, x_k\}| < k$, then we may (without loss of generality) assume that we want to compute $\Sigma_e(R_k(x_1, x_1, x_2, \dots, x_{k-1})|_\alpha)$. This is equivalent to computing $\Sigma_e(R_k(x_1, y, x_2, \dots, x_{k-1})|_\alpha)$ where y is a fresh variable and α is extended to α' so that $\alpha'(x_1, y)$ implies $x_1 = y$. Then, we map $\Sigma(R_k(x_1, y, \dots, x_k)|_{\alpha'})$ to a suitable

CSP(\mathcal{T}_e) instance as prescribed by Lemma 17. Assume instead that $|\{x_1, \dots, x_k\}| = k$. We let $\Sigma_e(R_k(x_1, \dots, x_k)|_\alpha)$ be undefined if $\alpha(x_i, x_j)$ is not defined for any distinct $x_i, x_j \in \{x_1, \dots, x_k\}$ – this is justified by Lemma 17. Otherwise, we map $\Sigma(R_k(x_1, \dots, x_k)|_\alpha)$ to a suitable CSP(\mathcal{T}_e) instance as prescribed by Lemma 17. We conclude the proof by noting that these computations are easy to perform in polynomial time so Σ_e is trivially polynomial-time accessible. \blacktriangleleft

Learning TSP Requires Rethinking Generalization

Chaitanya K. Joshi ✉ 

Institute for Infocomm Research, A*STAR, Singapore

Quentin Cappart ✉

Ecole Polytechnique de Montréal, Canada

Louis-Martin Rousseau ✉

Ecole Polytechnique de Montréal, Canada

Thomas Laurent ✉

Loyola Marymount University, LA, USA

Abstract

End-to-end training of neural network solvers for combinatorial optimization problems such as the Travelling Salesman Problem is intractable and inefficient beyond a few hundreds of nodes. While state-of-the-art Machine Learning approaches perform closely to classical solvers when trained on trivially small sizes, they are unable to generalize the learnt policy to larger instances of practical scales. Towards leveraging transfer learning to solve large-scale TSPs, this paper identifies inductive biases, model architectures and learning algorithms that promote generalization to instances larger than those seen in training. Our controlled experiments provide the first principled investigation into such *zero-shot* generalization, revealing that extrapolating beyond training data requires rethinking the neural combinatorial optimization pipeline, from network layers and learning paradigms to evaluation protocols.

2012 ACM Subject Classification Computing methodologies → Neural networks

Keywords and phrases Combinatorial Optimization, Travelling Salesman Problem, Graph Neural Networks, Deep Learning

Digital Object Identifier 10.4230/LIPIcs.CP.2021.33

Related Version *arXiv Pre-Print*: <https://arxiv.org/abs/2006.07054>

Supplementary Material *Software (Source Code and Dataset)*: <https://github.com/chaitjo/learning-tsp>; archived at `swb:1:dir:49220f1be1ae634e106f41948968734ac1569dbd`

Acknowledgements We would like to thank X. Bresson, V. Dwivedi, A. Ferber, E. Khalil, W. Kool, R. Levie, A. Prouvost, P. Veličković and the anonymous reviewers for helpful discussions.

1 Introduction

NP-hard combinatorial optimization problems are the family of integer constrained optimization problems which are intractable to solve optimally at large scales. Robust approximation algorithms to popular problems have immense practical applications and are the backbone of modern industries. Among combinatorial problems, the 2D Euclidean Travelling Salesman Problem (TSP) has been the most intensely studied NP-hard graph problem in the Operations Research (OR) community, with applications in logistics, genetics and scheduling [31]. TSP is intractable to solve optimally above thousands of nodes for modern computers [2]. In practice, the Concorde TSP solver [1] uses linear programming with carefully handcrafted heuristics to find solutions up to tens of thousands of nodes, but with prohibitive execution



© Chaitanya K. Joshi, Quentin Cappart, Louis-Martin Rousseau, and Thomas Laurent;
licensed under Creative Commons License CC-BY 4.0

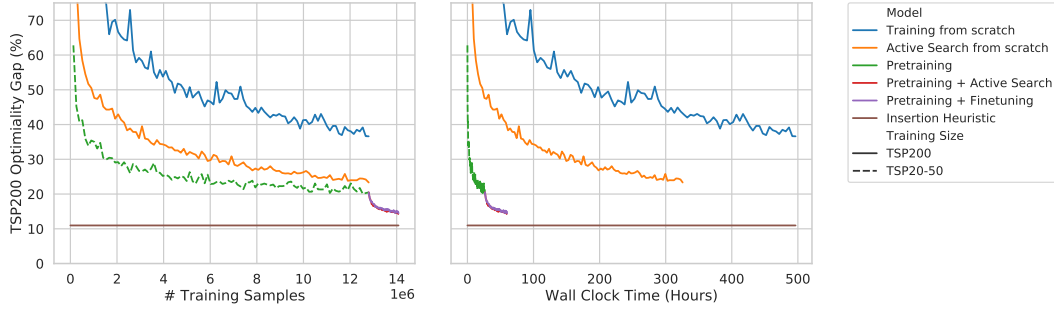
27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 33; pp. 33:1–33:21



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1 Computational challenges of learning large scale TSP.** We compare three identical autoregressive GNN-based models trained on 12.8 Million TSP instances via reinforcement learning. We plot average optimality gap to the Concorde solver on 1,280 held-out TSP200 instances vs. number of training samples (left) and wall clock time (right) during the learning process. Training on large TSP200 from scratch is intractable and sample inefficient. Active Search [4], which learns to directly overfit to the 1,280 held-out samples, further demonstrates the computational challenge of memorizing very few TSP200 instances. Comparatively, learning efficiently from trivial TSP20-TSP50 allows models to better generalize to TSP200 in a zero-shot manner, indicating positive knowledge transfer from small to large graphs. Performance can further improve via rapid finetuning on 1.28 Million TSP200 instances or by Active Search. Within our computational budget, a simple non-learned *furthest insertion* heuristic still outperforms all models. Precise experimental setup is described in **Appendix A**.

times.¹ Besides, the development of problem-specific OR solvers such as Concorde for novel or under-studied problems arising in scientific discovery [43] or computer architecture [37] requires significant time and specialized knowledge.

An alternate approach by the Machine Learning community is to develop generic learning algorithms which can be trained to solve *any* combinatorial problem directly from problem instances themselves [5]. Using classical problems such as TSP, Minimum Vertex Cover and Boolean Satisfiability as benchmarks, recent *end-to-end* approaches [28, 46, 33] leverage advances in graph representation learning [29, 19] and have shown competitive performance with OR solvers on trivially small problem instances up to few hundreds of nodes. Once trained, approximate solvers based on Graph Neural Networks (GNNs) have significantly favorable time complexity than their OR counterparts, making them highly desirable for real-time decision-making problems such as TSP and the associated class of Vehicle Routing Problems (VRPs).

1.1 Motivation

Scaling end-to-end approaches to practical and real-world instances is still an open question [5] as the training phase of state-of-the-art models on large graphs is extremely time-consuming. For graphs larger than few hundreds of nodes, the gap between GNN-based solvers and simple non-learned heuristics is especially evident for routing problems like TSP [28, 30].

As an illustration, Figure 1 presents the computational challenge of learning TSP on 200-node graphs (TSP200) in terms of both sample efficiency and wall clock time. Surprisingly, it is difficult to outperform a simple insertion heuristic when directly training on 12.8 Million TSP200 samples for 500 hours on university-scale hardware.

¹ The largest TSP solved by Concorde to date has 109,399 nodes with a total running time of 7.5 months.

We advocate for an alternative to expensive large-scale training: learning efficiently from trivially small TSP and transferring the learnt policy to larger graphs in a *zero-shot* fashion or via fast finetuning. Thus, identifying promising inductive biases, architectures and learning paradigms that enable such zero-shot generalization to large and more complex instances is a key concern for training practical solvers for real-world problems.

1.2 Contributions

Towards end-to-end learning of *scale-invariant* TSP solvers, we unify several state-of-the-art architectures and learning paradigms [40, 30, 12, 27] into one experimental pipeline and provide the first principled investigation on zero-shot generalization to large instances. Our findings suggest that learning scale-invariant TSP solvers requires rethinking the status quo of neural combinatorial optimization to explicitly account for generalization:

- The prevalent evaluation paradigm overshadows models’ poor generalization capabilities by measuring performance on fixed or trivially small TSP sizes.
- Generalization performance of GNN aggregation functions and normalization schemes benefits from explicit redesigns which account for shifting graph distributions, and can be further boosted by enforcing regularities such as constant graph diameters when defining problems using graphs.
- Autoregressive decoding enforces a sequential inductive bias which improves generalization over non-autoregressive models, but is costly in terms of inference time.
- Models trained with supervision are more amenable to post-hoc search, while reinforcement learning approaches scale better with more computation as they do not rely on labelled data.

We open-source our framework and datasets² to encourage the community to go beyond evaluating performance on fixed TSP sizes, develop more expressive and scale-invariant GNNs, as well as study transfer learning for combinatorial problems.

2 Related Work

Neural Combinatorial Optimization. In a recent survey, Bengio et al. [5] identified three broad approaches to leveraging machine learning for combinatorial optimization problems: learning alongside optimization algorithms [18, 8], learning to configure optimization algorithms [55, 14], and end-to-end learning to approximately solve optimization problems, *a.k.a.* neural combinatorial optimization [53, 4].

State-of-the-art end-to-end approaches for TSP use Graph Neural Networks (GNNs) [29, 19] and *sequence-to-sequence* learning [48] to construct approximate solutions directly from problem instances. Architectures for TSP can be classified as: (1) autoregressive approaches, which build solutions in a step-by-step fashion [28, 12, 30, 35]; and (2) non-autoregressive models, which produce the solution in one shot [40, 39, 27]. Models can be trained to imitate optimal solvers via supervised learning or by minimizing the length of TSP tours via reinforcement learning.

Other classical problems tackled by similar architectures include Vehicle Routing [38, 9], Maximum Cut [28], Minimum Vertex Cover [33], Boolean Satisfiability [46, 62], and Graph Coloring [23]. Using TSP as an illustration, we present a unified pipeline for characterizing neural combinatorial optimization architectures in Section 3.

² <https://github.com/chaitjo/learning-tsp>

Notably, TSP has emerged as a challenging testbed for neural combinatorial optimization. Whereas generalization to problem instances larger and more complex than those seen in training has at least partially been demonstrated on non-sequential problems such as SAT, MaxCut, and MVC [28, 33, 46], the same architectures do not show strong generalization for TSP [30, 27].

Combinatorial Optimization and GNNs. From the perspective of graph representation learning, algorithmic and combinatorial problems have recently been used to characterize the expressive power of GNNs [44]. An emerging line of work on learning to execute local graph algorithms [51] has led to the development of provably more expressive GNNs [10] and improved understanding of their generalization capability [60, 61]. Towards tackling realistic and large-scale combinatorial problems, this paper aims to quantify the limitations of prevalent GNN architectures and learning paradigms via zero-shot generalization to problems larger than those seen during training.

Novel Applications. Advances on classical combinatorial problems have shown promising results in downstream applications to novel or under-studied optimization problems in the physical sciences [20, 47] and computer architecture [36, 41], where the development of exact solvers is expensive and intractable. For example, autoregressive architectures provide a strong inductive bias for device placement optimization problems [37, 65], while non-autoregressive models [7] are competitive with autoregressive approaches [26, 63] for molecule generation tasks.

3 Neural Combinatorial Optimization Pipeline

Many NP-hard problems can be formulated as sequential decision making tasks on graphs due to their highly structured nature. Towards a controlled study of neural combinatorial optimization, we unify recent ideas [40, 30, 12, 27] via a five stage end-to-end pipeline illustrated in Figure 2. Our discussion focuses on the Travelling Salesman Problem (TSP), but the pipeline presented is generic and can be extended to characterize modern architectures for several NP-hard graph problems.

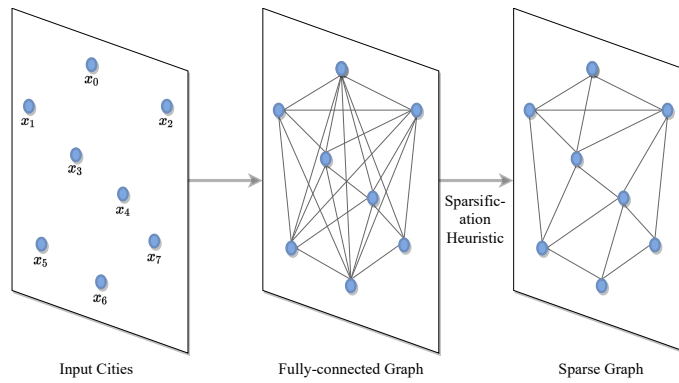
3.1 Problem Definition

The 2D Euclidean TSP is defined as follows: “Given a set of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?” Formally, given a fully-connected input graph of n cities (nodes) in the two dimensional unit square $S = \{x_i\}_{i=1}^n$ where each $x_i \in [0, 1]^2$, we aim to find a permutation of the nodes π , termed a tour, that visits each node once and has the minimum total length, defined as:

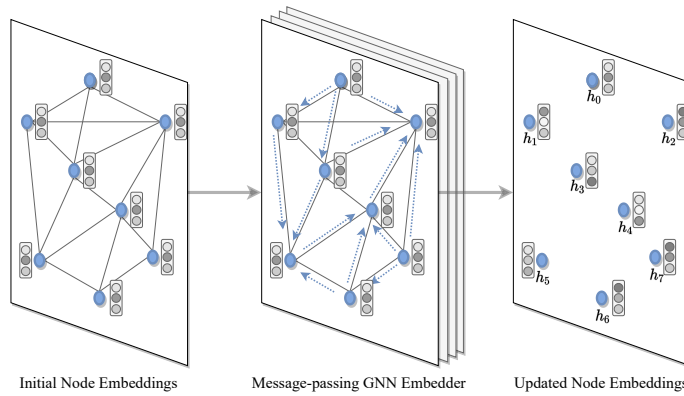
$$L(\pi|s) = \|x_{\pi_n} - x_{\pi_1}\|_2 + \sum_{i=1}^{n-1} \|x_{\pi_i} - x_{\pi_{i+1}}\|_2, \quad (1)$$

where $\|\cdot\|_2$ denotes the ℓ_2 norm.

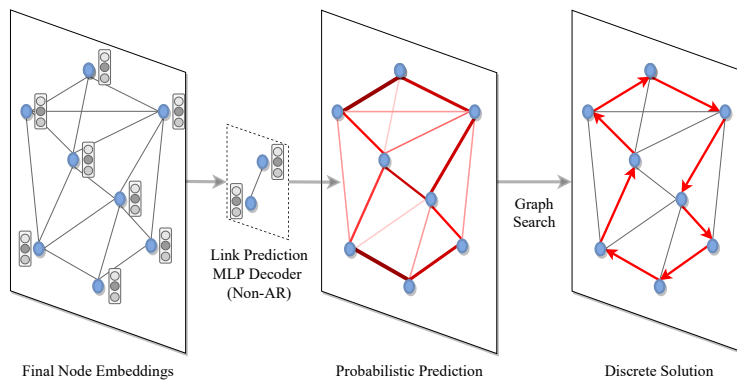
Graph Sparsification. Classically, TSP is defined on fully-connected graphs. Graph sparsification heuristics based on k -nearest neighbors aim to reduce TSP graphs, enabling models to scale up to large instances where pairwise computation for all nodes is intractable [28]



(a) Problem Definition: TSP is formulated via a fully-connected graph of cities/nodes. The graph can be sparsified via heuristics such as k -nearest neighbors.



(b) Graph Embedding: Embeddings for each graph node are obtained using a Graph Neural Network encoder. At each layer, nodes gather features from their neighbors to represent local graph structure via recursive message passing.



(c) Solution Decoding & Search: Probabilities are assigned to each node for belonging to the solution set, either independent of one-another (*i.e.* Non-autoregressive decoding) or conditionally through graph traversal (*i.e.* Autoregressive decoding). The predicted probabilities are converted into discrete decisions through classical graph search techniques such as greedy search or beam search.

■ **Figure 2 End-to-end neural combinatorial optimization pipeline:** The entire model is trained end-to-end via imitating an optimal solver (*i.e.* supervised learning) or through minimizing a cost function (*i.e.* reinforcement learning).

or learn faster by reducing the search space [27]. Notably, problem-specific graph reduction techniques have proven effective for out-of-distribution generalization to larger graphs for other NP-hard problems such as MVC and SAT [33].

Fixed size vs. variable size graphs. Most work on learning for TSP has focused on training with a fixed graph size [4, 30], likely due to ease of implementation. Learning from multiple graph sizes naturally enables better generalization within training size ranges, but its impact on generalization to larger TSP instances remains to be analyzed.

3.2 Graph Embedding

A Graph Neural Network (GNN) encoder computes d -dimensional representations for each node in the input TSP graph. At each layer, nodes gather features from their neighbors to represent local graph structure via recursive message passing [19]. Stacking L layers allows the network to build representations from the L -hop neighborhood of each node. Let h_i^ℓ and e_{ij}^ℓ denote respectively the node and edge feature at layer ℓ associated with node i and edge ij . We define the feature at the next layer via an *anisotropic* message passing scheme using an edge gating mechanism [6]:

$$h_i^{\ell+1} = h_i^\ell + \text{ReLU}\left(\text{NORM}\left(U^\ell h_i^\ell + \text{AGGR}_{j \in \mathcal{N}_i}\left(\sigma(e_{ij}^\ell) \odot V^\ell h_j^\ell\right)\right)\right), \quad (2)$$

$$e_{ij}^{\ell+1} = e_{ij}^\ell + \text{ReLU}\left(\text{NORM}\left(A^\ell e_{ij}^\ell + B^\ell h_i^\ell + C^\ell h_j^\ell\right)\right), \quad (3)$$

where $U^\ell, V^\ell, A^\ell, B^\ell, C^\ell \in \mathbb{R}^{d \times d}$ are learnable parameters, NORM denotes the normalization layer (BatchNorm [25], LayerNorm [3]), AGGR represents the neighborhood aggregation function (SUM, MEAN or MAX), σ is the sigmoid function, and \odot is the Hadamard product. As inputs $h_i^{\ell=0}$ and $e_{ij}^{\ell=0}$, we use d -dimensional linear projections of the node coordinate x_i and the euclidean distance $\|x_i - x_j\|_2$, respectively.

Anisotropic Neighborhood Aggregation. We make the aggregation function anisotropic or directional via a dense attention mechanism which scales the neighborhood features $h_j, \forall j \in \mathcal{N}_i$, using edge gates $\sigma(e_{ij})$. Anisotropic and attention-based GNNs such as Graph Attention Networks [50] and Gated Graph ConvNets [6] have been shown to outperform isotropic Graph ConvNets [29] across several challenging domains [13], including TSP [30, 27].

3.3 Solution Decoding

Non-autoregressive Decoding (NAR). Consider TSP as a link prediction task: each edge may belong/not belong to the optimal TSP solution independent of one another [40]. We define the edge predictor as a two layer MLP on the node embeddings produced by the final GNN encoder layer L , following Joshi et al. [27]. For adjacent nodes i and j , we compute the unnormalized edge logits:

$$\hat{p}_{ij} = W_2\left(\text{ReLU}\left(W_1\left([h_G, h_i^L, h_j^L]\right)\right)\right), \text{ where } h_G = \frac{1}{n} \sum_{i=0}^n h_i^L, \quad (4)$$

$W_1 \in \mathbb{R}^{3d \times d}, W_2 \in \mathbb{R}^{d \times 2}$, and $[\cdot, \cdot, \cdot]$ is the concatenation operator. The logits \hat{p}_{ij} are converted to probabilities over each edge p_{ij} via a softmax.

Autoregressive Decoding (AR). Although NAR decoders are fast as they produce predictions in one shot, they ignore the sequential ordering of TSP tours. Autoregressive decoders, based on attention [12, 30] or recurrent neural networks [53, 35], explicitly model this sequential inductive bias through step-by-step graph traversal.

We follow the attention decoder from Kool et al. [30], which starts from a random node and outputs a probability distribution over its neighbors at each step. Greedy search is used to perform the traversal over n time steps and masking enforces constraints such as not visiting previously visited nodes.

At time step t at node i , the decoder builds a context \hat{h}_i^C for the partial tour $\pi'_{t'}$, generated at time $t' < t$, by packing together the graph embedding h_G and the embeddings of the first and last node in the partial tour: $\hat{h}_i^C = W_C \begin{bmatrix} h_G, h_{\pi'_{t-1}}^L, h_{\pi'_1}^L \end{bmatrix}$, where $W_C \in \mathbb{R}^{3d \times d}$ and learned placeholders are used for $h_{\pi'_{t-1}}^L$ and $h_{\pi'_1}^L$ at $t = 1$. The context \hat{h}_i^C is then refined via a standard Multi-Head Attention (MHA) operation [49] over the node embeddings:

$$h_i^C = \text{MHA}\left(Q = \hat{h}_i^C, K = \{h_1^L, \dots, h_n^L\}, V = \{h_1^L, \dots, h_n^L\}\right), \quad (5)$$

where Q, K, V are inputs to the M -headed MHA ($M = 8$). The unnormalized logits for each edge e_{ij} are computed via a final attention mechanism between the context h_i^C and the embedding h_j :

$$\hat{p}_{ij} = \begin{cases} C \cdot \tanh\left(\frac{(W_Q h_i^C)^T \cdot (W_K h_j^L)}{\sqrt{d}}\right) & \text{if } j \neq \pi'_{t'} \quad \forall t' < t \\ -\infty & \text{otherwise.} \end{cases} \quad (6)$$

The tanh is used to maintain the value of the logits within $[-C, C]$ ($C = 10$) [4]. The logits \hat{p}_{ij} at the current node i are converted to probabilities p_{ij} via a softmax over all edges.

Inductive Biases. NAR approaches, which make predictions over edges independently of one-another, have shown strong out-of-distribution generalization for non-sequential problems such as SAT and MVC [33]. On the other hand, AR decoders come with the sequential/tour constraint built-in and are the default choice for routing problems [30]. Although both approaches have shown close to optimal performance on fixed and small TSP sizes under different experimental settings, it is important to fairly compare which inductive biases are most useful for generalization.

3.4 Solution Search

Greedy Search. For AR decoding, the predicted probabilities at node i are used to select the edge to travel along at the current step via sampling from the probability distribution p_i or greedily selecting the most probable edge p_{ij} , *i.e.* greedy search. Since NAR decoders directly output probabilities over all edges independent of one-another, we can obtain valid TSP tours using greedy search to traverse the graph starting from a random node and masking previously visited nodes. Thus, the probability of a partial tour π' can be formulated as $p(\pi') = \prod_{j' \sim i' \in \pi'} p_{i'j'}$, where each node j' follows node i' .

Beam Search and Sampling. During inference, we can increase the capacity of greedy search via limited width breadth-first beam search, which maintains the b most probable tours during decoding. Similarly, we can sample b solutions from the learnt policy and select the shortest tour among them. Naturally, searching longer, with more sophisticated

techniques [16, 58], or sampling more solutions allows trading off run time for solution quality. However, it has been noted that using large b for search/sampling or local search during inference may overshadow an architecture’s inability to generalize [15]. To better understand generalization, we focus on using greedy search and beam search/sampling with small $b = 128$.

3.5 Policy Learning

Supervised Learning. Models can be trained end-to-end via imitating an optimal solver at each step (*i.e.* supervised learning). For models with NAR decoders, the edge predictions are linked to the ground-truth TSP tour by minimizing the binary cross-entropy loss for each edge [40, 27]. For AR architectures, at each step, we minimize the cross-entropy loss between the predicted probability distribution over all edges leaving the current node and the next node from the groundtruth tour, following Vinyals et al. [53]. We use teacher-forcing to stabilize training [57].

Reinforcement Learning. Reinforcement learning is a elegant alternative in the absence of groundtruth solutions, as is often the case for understudied combinatorial problems. Models can be trained by minimizing problem-specific cost functions (the tour length in the case of TSP) via policy gradient algorithms [4, 30] or Q-Learning [28]. We focus on policy gradient methods due to their simplicity, and define the loss for an instance s parameterized by the model θ as $\mathcal{L}(\theta|s) = \mathbb{E}_{p_\theta(\pi|s)} [L(\pi)]$, the expectation of the tour length $L(\pi)$, where $p_\theta(\pi|s)$ is the probability distribution from which we sample to obtain the tour $\pi|s$. We use the REINFORCE gradient estimator [56] to minimize \mathcal{L} :

$$\nabla \mathcal{L}(\theta|s) = \mathbb{E}_{p_\theta(\pi|s)} [(L(\pi) - b(s)) \nabla \log p_\theta(\pi|s)], \quad (7)$$

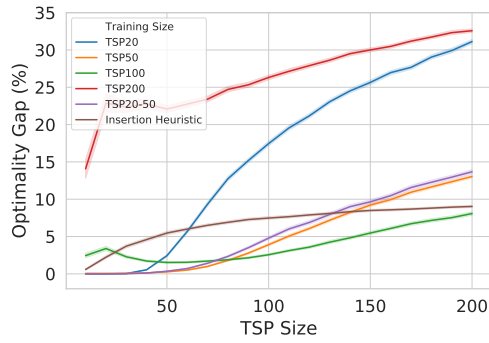
where the baseline $b(s)$ reduces gradient variance. Our experiments compare standard critic network baselines [4, 12] and the greedy rollout baseline proposed by Kool et al. [30].

4 Experiments

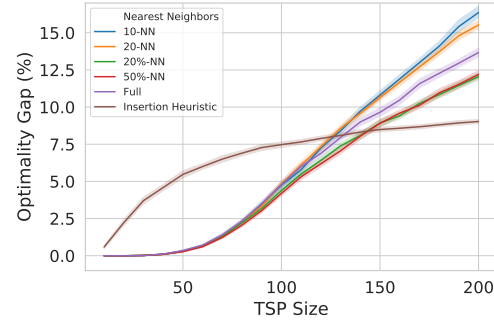
4.1 Controlled Experiment Setup

We design controlled experiments to probe the unified pipeline described in Section 3 in order to identify inductive biases, architectures and learning paradigms that promote zero-shot generalization. We focus on learning efficiently from small problem instances (TSP20-50) and measure generalization to a wider range of sizes, including large instances which are intractable to learn from (*e.g.* TSP200). We aim to fairly compare state-of-the-art ideas in terms of model capacity and training data, and expect models with good inductive biases for TSP to: (1) learn trivially small TSPs without hundreds of millions of training samples and model parameters; and (2) generalize reasonably well across smaller and larger instances than those seen in training. To quantify “good” generalization, we additionally evaluate our models against a simple, non-learned *furthest insertion* heuristic baseline [30].

Training Datasets. Our experiments focus on learning from variable TSP20-50 graphs. We also compare to training on fixed graph sizes TSP20, TSP50, TSP100, which have been the default choice in TSP literature. In the supervised learning paradigm, we generate a training set of 1,280,000 TSP samples and groundtruth tours using Concorde. Models are trained using the Adam optimizer for 10 epochs with a batch size of 128 and a fixed learning rate



■ **Figure 3 Learning from various TSP sizes.** The prevalent protocol of evaluation on training sizes overshadows brittle out-of-distribution performance to larger and smaller graphs.



■ **Figure 4 Impact of graph sparsification.** Maintaining a constant graph diameter across TSP sizes leads to better generalization on larger problems than using full graphs.

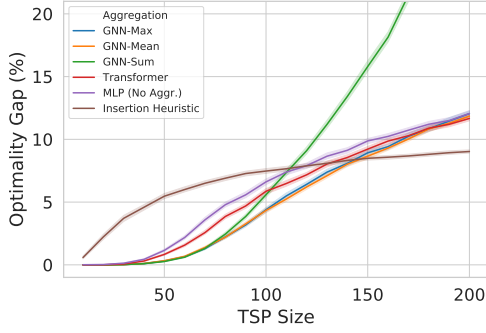
$1e-4$. For reinforcement learning, models are trained for 100 epochs on 128,000 TSP samples which are randomly generated for each epoch (without optimal solutions) with the same batch size and learning rate. Thus, both learning paradigms see 12,800,000 TSP samples in total. Considering that TSP20-50 are trivial in terms of complexity as they can be solved by simpler non-learned heuristics, training good solvers at this scale should ideally not require millions of instances.

Model Hyperparameters. For models with AR decoders, we use 3 GNN encoder layers followed by the attention decoder head, setting hidden dimension $d = 128$. For NAR models, we use the same hidden dimension and opt for 4 GNN encoder layers followed by the edge predictor. This results in approximately 350,000 trainable parameters for each model, irrespective of decoder type. Unless specified, most experiments use our best model configuration: AR decoding scheme and Graph ConvNet encoder with MAX aggregation and BatchNorm (with batch statistics). All models are trained via supervised learning except when comparing learning paradigms.

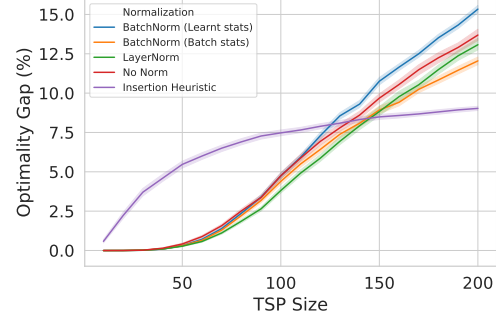
Evaluation. We compare models on a held-out test set of 25,600 TSP samples, consisting of 1,280 samples each of TSP10, TSP20, \dots , TSP200. Our evaluation metric is the optimality gap *w.r.t.* the Concorde solver, *i.e.* the average percentage ratio of predicted tour lengths relative to optimal tour lengths. To compare design choices among identical models, we plot line graphs of the optimality gap as TSP size increases (along with a 99%-ile confidence interval) using beam search with a width of 128. Compared to previous work which evaluated models on fixed problem sizes [4, 12, 30], our evaluation protocol identifies not only those models that perform well on training sizes, but also those that generalize better than non-learned heuristics for large instances which are intractable to train on.

4.2 Does learning from variable graphs help generalization?

We train five identical models on fully connected graphs of instances from TSP20, TSP50, TSP100, TSP200 and variable TSP20-50. The line plots of optimality gap across TSP sizes in Figure 3 indicates that learning from variable TSP sizes helps models retain performance across the range of graph sizes seen during training (TSP20-50). Variable graph training



■ **Figure 5 Impact of GNN aggregation functions.** For larger graphs, aggregation functions that are agnostic to node degree (MEAN and MAX) are able to outperform theoretically more expressive aggregators.



■ **Figure 6 Impact of normalization schemes.** Modifying BatchNorm to account for changing graph statistics leads to better generalization on larger graphs.

compared to training solely on the maximum sized instances (TSP50) leads to marginal gains on small instances but, somewhat counter-intuitively, does not enable better generalization to larger problems. Learning from small TSP20 is unable to generalize to large sizes while TSP100 models generalize poorly to trivially easy sizes, suggesting that the prevalent protocol of evaluation on training sizes [30, 27] overshadows brittle out-of-distribution performance.

Training on TSP200 graphs is intractable within our computational budget, see Figure 1. TSP100 is the only model which generalizes better to large TSP200 than the non-learned baseline. However, training on TSP100 can also be prohibitively expensive: one epoch takes approximately 8 hours (TSP100) vs. 2 hours (TSP20-50) (details in Appendix B). For rapid experimentation, we train efficiently on variable TSP20-50 for the rest of our study.

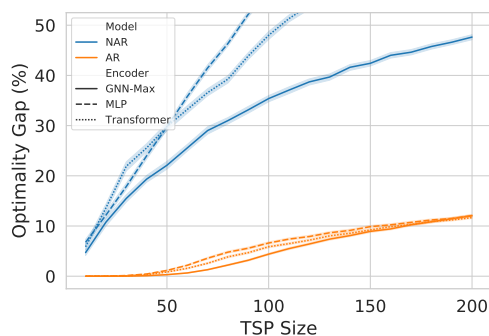
4.3 What is the best graph sparsification heuristic?

Figure 4 compares full graph training to the following heuristics: (1) **Fixed node degree** across graph sizes, via connecting each node in TSP_n to its k -nearest neighbors, enabling GNNs layers to specialize to constant degree k ; and (2) **Fixed graph diameter** across graph sizes, via connecting each node in TSP_n to its $n \times k\%$ -nearest neighbors, ensuring that the same number of message passing steps are required to diffuse information across both small and large graphs.

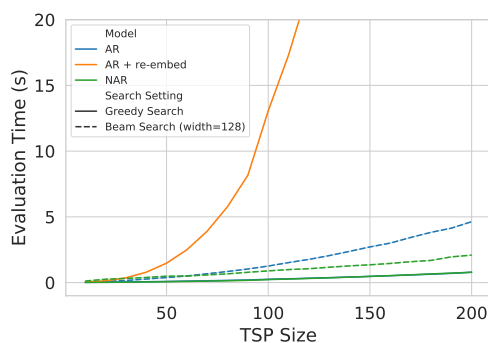
Although both sparsification techniques lead to faster convergence on training instance sizes, we find that only approach (2) leads to better generalization on larger problems than using full graphs. Consequently, all further experiments use approach (2) to operate on sparse 20%-nearest neighbors graphs. Our results also suggest that developing more principled graph reduction techniques beyond simple k -nearest neighbors for augmenting learning-based approaches may be a promising direction.

4.4 What is the relationship between GNN aggregation functions and normalization layers?

In Figure 5, we compare identical models with anisotropic SUM, MEAN and MAX aggregation functions. As baselines, we consider the Transformer encoder on full graphs [12, 30] as well as a structure-agnostic MLP on each node, which can be instantiated by not using any aggregation function in Eq.(2), *i.e.* $h_i^{\ell+1} = h_i^{\ell} + \text{ReLU}(\text{NORM}(U^{\ell}h_i^{\ell}))$.



■ **Figure 7 Comparing AR and NAR decoders.** Sequential decoding is a powerful inductive bias for TSP as it enables significantly better generalization, even in the absence of graph structure (MLP encoders).



■ **Figure 8 Inference time for various decoders.** One-shot NAR decoding is significantly faster than sequential AR, especially when re-embedding the graph at each decoding step [28].

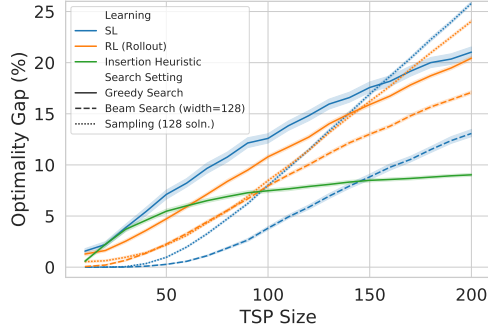
We find that the choice of GNN aggregation function does not have an impact when evaluating models within the training size range TSP20-50. As we tackle larger graphs, GNNs with aggregation functions that are agnostic to node degree (MEAN and MAX) are able to outperform Transformers and MLPs. Importantly, the theoretically more expressive SUM aggregator [59] generalizes worse than structure-agnostic MLPs, as it cannot handle the distribution shift in node degree and neighborhood statistics across graph sizes, leading to unstable or exploding node embeddings [51]. We use the MAX aggregator in further experiments, as it generalizes well for both AR and NAR decoders (not shown).

We also experiment with the following normalization schemes: (1) standard BatchNorm which learns mean and variance from training data, as well as (2) BatchNorm with batch statistics; and (3) LayerNorm, which normalizes at the embedding dimension instead of across the batch. Figure 6 indicates that BatchNorm with batch statistics and LayerNorm are able to better account for changing statistics across different graph sizes. Standard BatchNorm generalizes worse than not doing any normalization, thus our other experiments use BatchNorm with batch statistics.

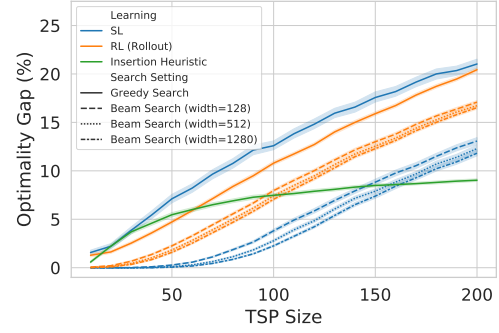
Poor performance on large graphs than those seen during training can be explained by unstable node and graph-level representations due to the choice of aggregation and normalization schemes. Using MAX aggregators and BatchNorm with batch statistics are temporary hacks to overcome the failure of the current architectural components. Overall, our results suggest that inference beyond training sizes will require the development of expressive GNN mechanisms that are able to leverage global graph topology [17, 52] while being invariant to distribution shift [32].

4.5 Which decoder has a better inductive bias for TSP?

Figure 7 compares NAR and AR decoders for identical models. To isolate the impact of the decoder’s inductive bias without the inductive bias imposed by GNNs, we also show Transformer encoders on full graphs as well as structure-agnostic MLPs. Within our experimental setup, AR decoders are able to fit the training data as well as generalize significantly better than NAR decoders, indicating that sequential decoding is powerful for TSP even without graph information.



■ **Figure 9 Comparing solution search settings.** Under greedy decoding, RL demonstrates better performance and generalization. Conversely, SL models improve over their RL counterparts when performing beam search or sampling.



■ **Figure 10 Impact of increasing beam width.** Teacher-forcing during SL leads to poor generalization under greedy decoding, but makes the probability distribution more amenable to beam search.

Conversely, NAR architectures are a poor inductive bias as they require significantly more computation to perform competitively to AR decoders. For instance, recent work [40, 27] used more than 30 layers with over 10 Million parameters. We believe that such overparameterized networks are able to memorize all patterns for small TSP training sizes [64], but the learnt policy is unable to generalize beyond training graph sizes. At the same time, when compared fairly within the same experimental settings, NAR decoders are significantly faster than AR decoders described in Section 3.3 as well as those which re-embed the graph at each decoding step [28], see Figure 8.

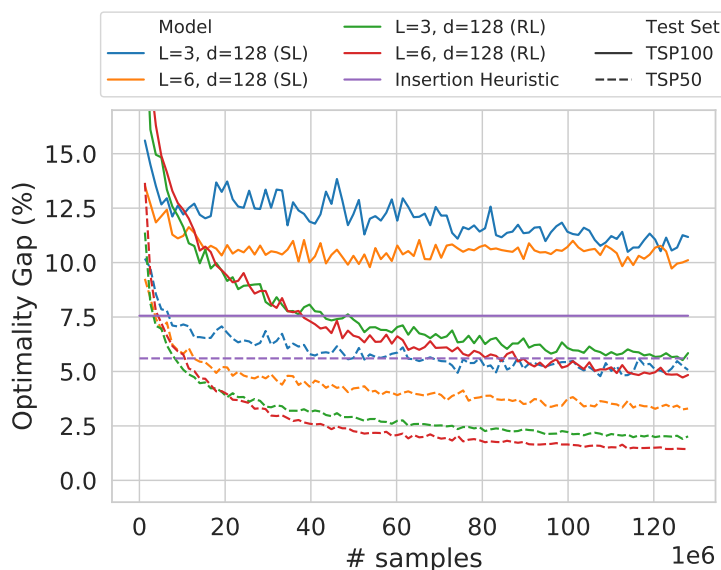
4.6 How does the learning paradigms impact the search phase?

Identical models are trained via supervised learning (SL) and reinforcement learning (RL) (We show only the greedy rollout baseline for clarity.). Figure 9 illustrates that, when using greedy decoding during inference, RL models perform better on the training size as well as on larger graphs. Conversely, SL models improve over their RL counterparts when performing beam search or sampling.

In Appendix D, we find that the rollout baseline, which encourages better greedy behaviour, leads to the model making very confident predictions about selecting the next node at each decoding step, even out of training size range. In contrast, SL models are trained with teacher forcing, *i.e.* imitating the optimal solver at each step instead of using their own prediction. This results in less confident predictions and poor greedy decoding, but makes the probability distribution more amenable to beam search and sampling, as shown in Figure 10. Our results advocate for tighter coupling between the training and inference phase of learning-driven TSP solvers, mirroring recent findings in generative models for text [21].

4.7 Which learning paradigm scales better?

Our experiments till this point have focused on isolating the impact of various pipeline components on zero-shot generalization under limited computation. At the same time, recent results on natural language have highlighted the power of large scale pre-training for effective transfer learning [42]. To better understand the impact of learning paradigms when scaling



■ **Figure 11 Scaling computation and parameters for SL and RL-trained models.** All models are trained on TSP20-50. We plot optimality gap on 1,280 held-out samples of both TSP50 (performance on training size) and TSP100 (out-of-distribution generalization) under greedy decoding. Note that SL models are less amenable than RL models to greedy search. RL models are able to keep improving their performance within as well as outside of training size range with more data. On the other hand, SL performance is bottlenecked by the need for optimal groundtruth solutions.

computation, we double the model parameters (up to 750,000) and train on tens times more data (12.8M samples) for AR architectures. We monitor optimality gap on the training size range (TSP20-50) as well as a larger size (TSP100) vs. the number of training samples.

In Figure 11, we see that increasing model capacity leads to better learning. Notably, RL models, which train on unique randomly generated samples throughout, are able to keep improving their performance within as well as outside of training size range as they see more samples. On the other hand, SL is bottlenecked by the need for optimal groundtruth solutions: SL models iterate over the same 1.28M unique labelled samples and stop improving at a point. Beyond favorable inductive biases, distributed and sample-efficient RL algorithms [45] may be a key ingredient for learning from larger TSPs beyond tens of nodes.

5 Conclusion

Learning-driven solvers for combinatorial problems such as the Travelling Salesman Problem have shown promising results for trivially small instances up to a few hundred nodes. However, scaling such *end-to-end* learning approaches to real-world instances is still an open question as training on large graphs is extremely time-consuming. As a motivating example, we have demonstrated that state-of-the-art techniques are unable to outperform simple insertion heuristics on TSP beyond 200 nodes when trained on university-scale hardware.

This paper advocates for an alternative to expensive large-scale training: the generalization gap between end-to-end approaches and insertion heuristics can be brought closer by training models efficiently from trivially small TSP and transferring the learnt policy to larger graphs in a *zero-shot* fashion or via fast fine-tuning. Thus, identifying promising inductive biases,

architectures and learning paradigms that enable such zero-shot generalization to large and more complex instances is a key concern for developing practical solvers for real-world combinatorial problems.

We perform the first principled investigation into zero-shot generalization for learning large scale TSP, unifying state-of-the-art architectures and learning paradigms into one experimental pipeline. Our findings suggest that key design choices such as Graph Neural Network layers, normalization schemes, graph sparsification, and learning paradigms need to be explicitly re-designed to consider out-of-distribution generalization.

Future work can tackle generalization to large-scale problem instances in the following ways: (1) GNN architectures which are sufficiently expressive beyond simple max/mean aggregation functions, while at the same time incorporating inductive biases which account for the shifting graph degree distribution and statistics that characterize larger scale combinatorial problems. (2) Novel learning paradigms which focus on generalization, *e.g.* this work explored zero-shot generalization to larger problems, but the logical next step is to fine-tune the model on a small number of larger problems. Thus, it will be interesting to explore fine-tuning/generalization as a meta-learning problem, wherein the goal is to train model parameters specifically for fast adaptation and fine-tuning to new data distributions and problem sizes.

References

- 1 David Applegate, Ribert Bixby, Vasek Chvatal, and William Cook. Concorde tsp solver, 2006.
- 2 David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2006.
- 3 Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint*, 2016. [arXiv:1607.06450](https://arxiv.org/abs/1607.06450).
- 4 Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *International Conference on Learning Representations*, 2017. URL: <https://openreview.net/pdf?id=Bk9mx1SFx>.
- 5 Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d’horizon. *arXiv preprint*, 2018. [arXiv:1811.06128](https://arxiv.org/abs/1811.06128).
- 6 Xavier Bresson and Thomas Laurent. An experimental study of neural networks for variable graphs. In *International Conference on Learning Representations*, 2018.
- 7 Xavier Bresson and Thomas Laurent. A two-step graph convolutional decoder for molecule generation. In *NeurIPS Workshop on Machine Learning and the Physical Sciences*, 2019.
- 8 Quentin Cappart, Emmanuel Goutier, 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.
- 9 Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. In *Advances in Neural Information Processing Systems*, pages 6278–6289, 2019.
- 10 Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Liò, and Petar Veličković. Principal neighbourhood aggregation for graph nets. *arXiv preprint*, 2020. [arXiv:2004.05718](https://arxiv.org/abs/2004.05718).
- 11 George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.
- 12 Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. Learning heuristics for the tsp by policy gradient. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 170–181. Springer, 2018.
- 13 Vijay Prakash Dwivedi, Chaitanya K Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *arXiv preprint*, 2020. [arXiv:2003.00982](https://arxiv.org/abs/2003.00982).

- 14 Aaron Ferber, Bryan Wilder, Bistra Dilkina, and Milind Tambe. Mipaal: Mixed integer program as a layer. In *AAAI Conference on Artificial Intelligence*, 2020.
- 15 Antoine François, Quentin Cappart, and Louis-Martin Rousseau. How to evaluate machine learning approaches for combinatorial optimization: Application to the travelling salesman problem. *arXiv preprint*, 2019. [arXiv:1909.13121](#).
- 16 Zhang-Hua Fu, Kai-Bin Qiu, Meng Qiu, and Hongyuan Zha. Targeted sampling of enlarged neighborhood via monte carlo tree search for {tsp}, 2020. URL: <https://openreview.net/forum?id=ByxtHCVKwB>.
- 17 Vikas K Garg, Stefanie Jegelka, and Tommi Jaakkola. Generalization and representational limits of graph neural networks. *arXiv preprint*, 2020. [arXiv:2002.06157](#).
- 18 Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *arXiv preprint*, 2019. [arXiv:1906.01629](#).
- 19 Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1263–1272. JMLR. org, 2017.
- 20 Rafael Gómez-Bombarelli, Jennifer N Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D Hirzel, Ryan P Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 4(2):268–276, 2018.
- 21 Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020. URL: <https://openreview.net/forum?id=rygGQyrFvH>.
- 22 John J Hopfield and David W Tank. “neural” computation of decisions in optimization problems. *Biological cybernetics*, 52(3):141–152, 1985.
- 23 Jiayi Huang, Mostofa Patwary, and Gregory Diamos. Coloring big graphs with alphagozero. *arXiv preprint*, 2019. [arXiv:1902.10162](#).
- 24 Gurobi Optimization Inc. Gurobi optimizer reference manual. URL <http://www.gurobi.com>, 2015.
- 25 Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint*, 2015. [arXiv:1502.03167](#).
- 26 Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Junction tree variational autoencoder for molecular graph generation. In *International Conference on Machine Learning*, pages 2323–2332, 2018.
- 27 Chaitanya K Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint*, 2019. [arXiv:1906.01227](#).
- 28 Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.
- 29 Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.
- 30 Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019. URL: <https://openreview.net/forum?id=ByxBFsRqYm>.
- 31 Jan Karel Lenstra and AHG Rinnooy Kan. Some simple applications of the travelling salesman problem. *Journal of the Operational Research Society*, 26(4):717–733, 1975.
- 32 Ron Levie, Michael M Bronstein, and Gitta Kutyniok. Transferability of spectral graph convolutional neural networks. *arXiv preprint*, 2019. [arXiv:1907.12972](#).
- 33 Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*, pages 539–548, 2018.

- 34 John DC Little, Katta G Murty, Dura W Sweeney, and Caroline Karel. An algorithm for the traveling salesman problem. *Operations research*, 11(6):972–989, 1963.
- 35 Qiang Ma, Suwen Ge, Danyang He, Darshan Thaker, and Iddo Drori. Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. In *AAAI Workshop on Deep Learning on Graphs: Methodologies and Applications*, 2020.
- 36 Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrisnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. ACM, 2019.
- 37 Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML’17, page 2430–2439. JMLR.org, 2017.
- 38 Mohammadreza Nazari, Afshin Oroojlooy, Lawrence Snyder, and Martin Takác. Reinforcement learning for solving the vehicle routing problem. In *Advances in Neural Information Processing Systems*, pages 9861–9871, 2018.
- 39 Alex Nowak, David Folqué, and Joan Bruna Estrach. Divide and conquer networks. In *6th International Conference on Learning Representations, ICLR 2018*, 2018.
- 40 Alex Nowak, Soledad Villar, Afonso S Bandeira, and Joan Bruna. A note on learning algorithms for quadratic assignment with graph neural networks. *arXiv preprint*, 2017. [arXiv:1706.07450](https://arxiv.org/abs/1706.07450).
- 41 Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Regal: Transfer learning for fast optimization of computation graphs. *arXiv preprint*, 2019. [arXiv:1905.02494](https://arxiv.org/abs/1905.02494).
- 42 Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint*, 2019. [arXiv:1910.10683](https://arxiv.org/abs/1910.10683).
- 43 Maithra Raghu and Eric Schmidt. A survey of deep learning for scientific discovery. *arXiv preprint*, 2020. [arXiv:2003.11755](https://arxiv.org/abs/2003.11755).
- 44 Ryoma Sato, Makoto Yamada, and Hisashi Kashima. Approximation ratios of graph neural networks for combinatorial problems. In *Advances in Neural Information Processing Systems*, pages 4081–4090, 2019.
- 45 John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint*, 2017. [arXiv:1707.06347](https://arxiv.org/abs/1707.06347).
- 46 Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a sat solver from single-bit supervision. *arXiv preprint*, 2018. [arXiv:1802.03685](https://arxiv.org/abs/1802.03685).
- 47 Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Židek, Alexander WR Nelson, Alex Bridgland, et al. Improved protein structure prediction using potentials from deep learning. *Nature*, pages 1–5, 2020.
- 48 Ilya Sutskever, Oriol Vinyals, and Quoc VV Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.
- 49 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.
- 50 Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018. URL: <https://openreview.net/forum?id=rJXMpikCZ>.
- 51 Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. In *International Conference on Learning Representations*, 2020. URL: <https://openreview.net/forum?id=SkgK00EtvS>.
- 52 Clement Vignac, Andreas Loukas, and Pascal Frossard. Building powerful and equivariant graph neural networks with message-passing. *arXiv preprint*, 2020. [arXiv:2006.15107](https://arxiv.org/abs/2006.15107).
- 53 Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

- 54 Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint*, 2019. [arXiv:1909.01315](#).
- 55 Bryan Wilder, Bistra Dilkina, and Milind Tambe. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1658–1665, 2019.
- 56 Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.
- 57 Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.
- 58 Zhihao Xing and Shikui Tu. A graph neural network assisted monte carlo tree search approach to traveling salesman problem, 2020. URL: <https://openreview.net/forum?id=Syg6fxxrKDB>.
- 59 Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint*, 2018. [arXiv:1810.00826](#).
- 60 Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? In *International Conference on Learning Representations*, 2019.
- 61 Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. *arXiv preprint*, 2020. [arXiv:2009.11848](#).
- 62 Emre Yolcu and Barnabas Poczos. Learning local search heuristics for boolean satisfiability. In *Advances in Neural Information Processing Systems*, pages 7990–8001, 2019.
- 63 Jiaxuan You, Bowen Liu, Zhitao Ying, Vijay Pande, and Jure Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. In *Advances in neural information processing systems*, pages 6410–6421, 2018.
- 64 Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint*, 2016. [arXiv:1611.03530](#).
- 65 Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, et al. Gdp: Generalized device placement for dataflow graphs. *arXiv preprint*, 2019. [arXiv:1910.01578](#).

A Additional Context for Figure 1: Computational challenges of learning large scale TSP

Experimental Setup. In Figure 1, we illustrate the computational challenges of learning large scale TSP by comparing three identical models trained on 12.8 Million TSP instances via reinforcement learning. Our experimental setup largely follows Section 4.1. All models use identical configurations: autoregressive decoding and Graph ConvNet encoder with MAX aggregation and LayerNorm. The TSP20-50 model is trained using the greedy rollout baseline [30] and the Adam optimizer with batch size 128 and learning rate $1e - 4$. Direct training, active search and finetuning on TSP200 samples is done using learning rate $1e - 5$, as we found larger learning rates to be unstable. During active search and finetuning, we use an exponential moving average baseline, as recommended by Bello et al. [4].

Furthest Insertion Baseline. We characterize “good” generalization across our experiments by the well-known *furthest insertion* heuristic, which constructively builds a solution/partial tour π' by inserting node i between tour nodes $j_1, j_2 \in \pi'$ such that the distance from node i to its nearest tour node j_1 is maximized. The Appendix of Kool et al. [30] provides a detailed description of insertion heuristic approaches.

We motivate our work by showing that learning from large TSP200 is intractable on university-scale hardware, and that efficient pre-training on trivial TSP20-50 enables models to better generalize to TSP200 in a zero-shot manner. Within our computational budget, furthest insertion still outperforms our best models. At the same time, we are not claiming that it is *impossible* to outperform insertion heuristics with current approaches: reinforcement learning-driven approaches will only continue to improve performance with more computation, training data and sample efficient learning algorithms. We want to use simple non-learned baselines to motivate the development of better architectures, learning paradigms and evaluation protocols for neural combinatorial optimization.

Routing Problems and Generalization. It is worth mentioning why we chose to study TSP in particular. Firstly, TSP has stood the test of time in terms of relevance and continues to serve as an engine of discovery for general purpose techniques in applied mathematics [11, 34, 22].

TSP and associated routing problems have also emerged as a challenging testbed for learning-driven approaches to combinatorial optimization. Whereas generalization to problem instances larger and more complex than those seen in training has at least partially been demonstrated on non-sequential problems such as SAT, MaxCut, Minimum Vertex Cover [28, 33, 46]³, the same architectures do not show strong generalization for TSP. For example, furthest insertion heuristics outperforms or are competitive with state-of-the-art approaches for TSP above tens of nodes, see Figure D.1.(e, f) from Khalil et al. [28] or Figure 5 from Kool et al. [30], despite using more computation and data than our controlled study.

B Hardware and Timings

Fairly timing research code can be difficult due to differences in libraries used, hardware configurations and programmer skill. In Table 1, we report approximate total training time and inference time across TSP sizes for the model setup described in Section 4.1. All experiments were implemented in PyTorch and run on an Intel Xeon CPU E5-2690 v4 server and four Nvidia 1080Ti GPUs. Four experiments were run on the server at any given time (each using a single GPU). Training time may vary based on server load, thus we report the lowest training time across several runs in Table 1.

We experimented with improving the latency of GNN-based models by using graph machine learning libraries such as DGL [54]. DGL requires graphs to be prepared as sparse library-specific data objects, which significantly boosts the inference speed of GNNs. However, using DGL had a negative impact on the speed of the rest of our pipeline (batched data preparation, decoders, beam search). This issue is especially amplified for reinforcement learning, where we constantly generate new random datasets at each epoch. For now, we present timings and results with pure PyTorch code. We confirm that results are consistent with using DGL, but decided against it in order to run a large volume of experiments for more comprehensive analysis.

C Datasets

We generate 2D Euclidean TSP instances of varying sizes and complexities as graphs of n node locations sampled uniformly in the unit square $S = \{x_i\}_{i=1}^n$ and $x_i \in [0, 1]^2$. For supervised learning, we generate a training set of 1,280,000 samples each for TSP20, TSP50,

³ It is worth noting that classical algorithmic and symbolic components such as graph reduction, sophisticated tree search as well as post-hoc local search have been pivotal and complementary to GNNs in enabling such generalization [33].

■ **Table 1** Approximate training time (12.8M samples) and inference time (1,280 samples) across TSP sizes and search settings for SL and RL-trained models. *GS*: Greedy search, *BS128*: beam search with width 128, *S128*: sampling 128 solutions. RL training uses the rollout baseline and timing includes the time taken to update the baseline after each 128,000 samples.

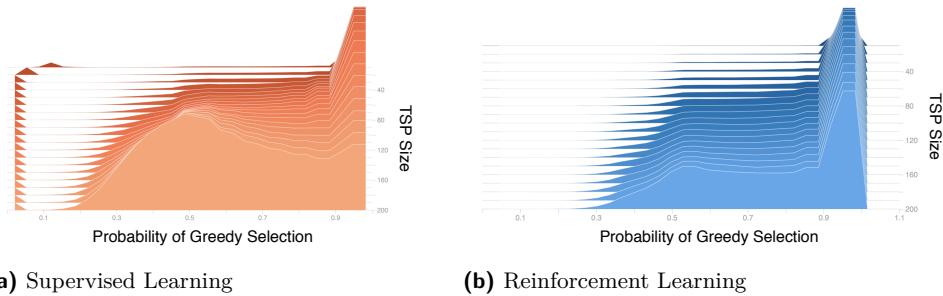
Graph Size	Training Time		Inference Time		
	SL	RL	GS	BS128	S128
TSP20	4h 24m	8h 02m	2.62s	7.06s	63.37s
TSP20-50	9h 49m	15h 47m	-	-	-
TSP50	16h 11m	40h 29m	7.45s	29.09s	86.48s
TSP100	68h 34m	108h 30m	19.04s	98.26s	180.30s
TSP200	-	495h 55m	54.88s	372.09s	479.37s

TSP100, and TSP20-50. The groundtruth tours are obtained using the Concorde solver [1]. For reinforcement learning, 128,000 samples are randomly generated for each epoch (without optimal solutions). We compare models on a held-out test set of 25,600 TSP samples and their corresponding optimal tours, consisting of 1,280 instances each of TSP10, TSP20, ..., TSP200. We release all dataset files as well as the associated scripts to produce TSP datasets of arbitrarily large sizes along with our open-source codebase.

D Learning Paradigms and Amenity to Search

Figure 10 demonstrate that SL models are more amenable to beam search and sampling, but are outperformed by RL-rollout models under greedy search. In Figure 12, we investigate the impact of learning paradigms on probability distributions by plotting histograms of the probabilities of greedy selections during inference across TSP sizes for identical models trained with SL and RL. We find that the rollout baseline, which encourages better greedy behaviour, leads to the model making very confident predictions about selecting the next node at each decoding step, even beyond training size range. In contrast, SL models are trained with teacher forcing, *i.e.* imitating the optimal solver at each step instead of using their own prediction. This results in less confident predictions and poor greedy decoding, but makes the probability distribution more amenable to beam search and sampling techniques.

We understand this phenomenon as follows: More confident predictions (Figure 12b) do not automatically imply better solutions. However, sampling repeatedly or maintaining the top- b most probable solutions from such distributions is likely to contain very similar tours. On the other hand, less sharp distributions (Figure 12a) are likely to yield more diverse tours with increasing b . This may result in comparatively better optimality gap, especially for TSP sizes larger than those seen in training.



■ **Figure 12** Histograms of greedy selection probabilities (x-axis) across TSP sizes (y-axis).

E Visualizing Model Predictions

As a final note, we present a visualization tool for generating model predictions and heatmaps of TSP instances, see Figures 13 and 14. We advocate for the development of more principled approaches to neural combinatorial optimization, *e.g.* along with model predictions, visualizing the reduce costs for each edge (obtained using the Gurobi solver [24]) may help debug and improve learning-driven approaches in the future.

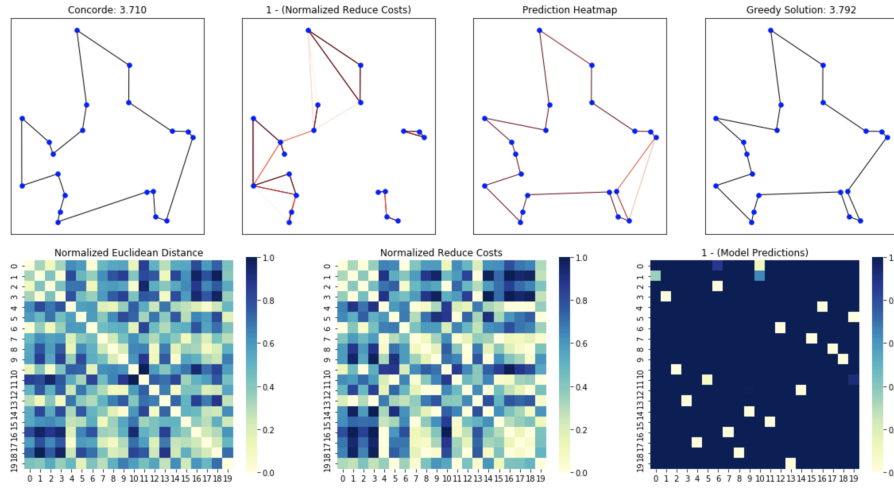


Figure 13 Prediction visualization for TSP20.

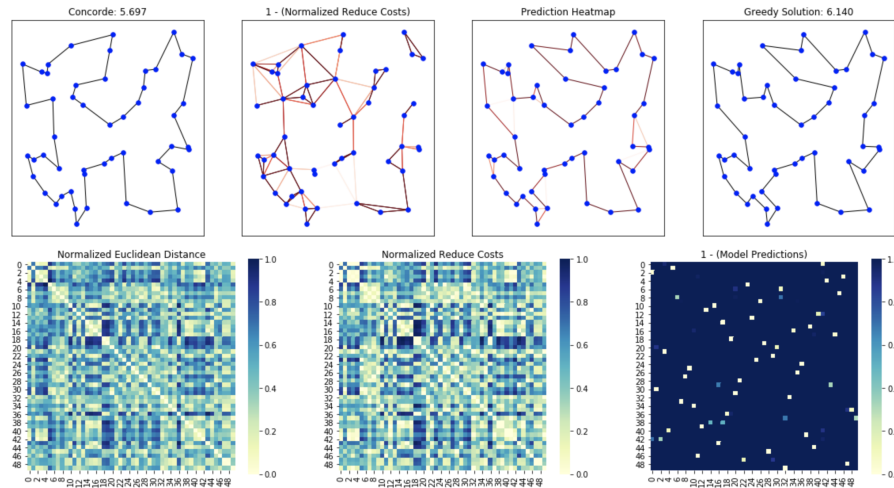
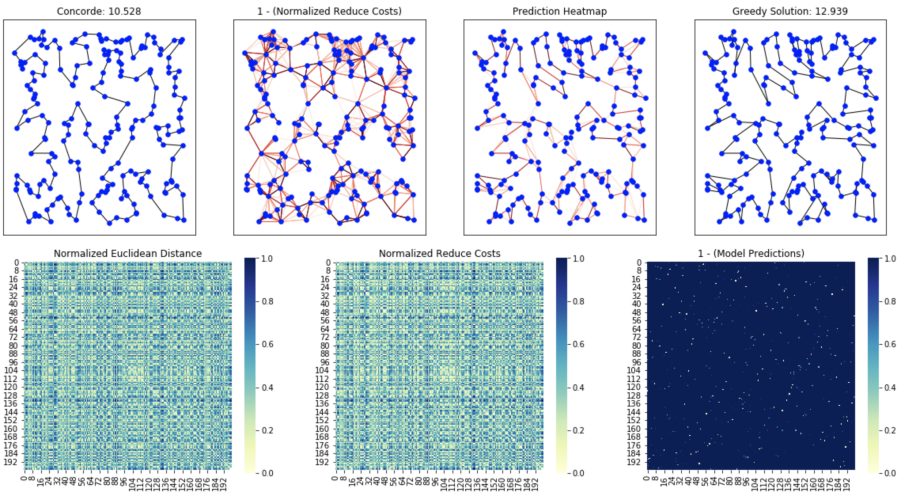


Figure 14 Prediction visualization for TSP50.



■ **Figure 15** Prediction visualization for TSP200.

SAT Modulo Symmetries for Graph Generation

Markus Kirchweger ✉

Algorithms and Complexity Group, TU Wien, Austria

Stefan Szeider ✉

Algorithms and Complexity Group, TU Wien, Austria

Abstract

We propose a novel constraint-based approach to graph generation. Our approach utilizes the interaction between a CDCL SAT solver and a special symmetry propagator where the SAT solver runs on an encoding of the desired graph property. The symmetry propagator checks partially generated graphs for minimality w.r.t. a lexicographic ordering during the solving process. This approach has several advantages over a static symmetry breaking: (i) symmetries are detected early in the generation process, (ii) symmetry breaking is seamlessly integrated into the CDCL procedure, and (iii) the propagator can perform a complete symmetry breaking without causing a prohibitively large initial encoding. We instantiate our approach by generating extremal graphs with certain restrictions in terms of girth and diameter. With our approach, we could confirm the Simon-Murty Conjecture (1979) on diameter-2-critical graphs for graphs up to 18 vertices.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Mathematics of computing → Extremal graph theory

Keywords and phrases symmetry breaking, SAT encodings, graph generation, combinatorial search, extremal graphs, CDCL

Digital Object Identifier 10.4230/LIPIcs.CP.2021.34

Supplementary Material *Software (Source Code)*: <https://doi.org/10.5281/zenodo.5170575>

Funding The authors acknowledge the support from the Austrian Science Fund (FWF), project P32441, and from the Vienna Science and Technology Fund (WWTF), project ICT19-065.

1 Introduction

Many challenging problems in Combinatorics can be stated as the question of whether a graph with a particular property exists. A common approach to such problems is to use a tool like Nauty [29] to generate all connected graphs up to isomorphism with a given number n of vertices and to check each of them for the desired property. However, up to isomorphism, already for $n = 11$ there are over a billion connected graphs, so this method quickly approaches its limit.

As demonstrated by Codish et al. [10], constraint-based graph generation offers a compelling alternative approach. The graph property is expressed in terms of constraints, and further constraints expressing a *static symmetry break* are added. The latter constraints are based on a lexicographic ordering of solution graphs and exclude some graphs that are not minimal for this ordering. This approach has the advantage that the graph property is taken into account already during the generation process. However, a complete symmetry breaking requires a prohibitively large encoding size. Therefore, one needs to confine only to a partial check, e.g., that swapping two vertices doesn't yield a lexicographically smaller graph.

We propose the novel approach *SAT modulo Symmetries* (SMS) to constraint-based graph generation. SMS utilizes the interaction between a CDCL¹ SAT solver and a special propagator excluding lexicographically non-minimal graphs during the search. Thus, in contrast to static symmetry breaking, the minimality check is not added to the encoding but is carried out dynamically by the symmetry propagator.

¹ Conflict-Driven Clause Learning is the predominantly leading algorithmic paradigm for state-of-the-art SAT solvers [28].



The symmetry propagator is called from within a CDCL-based SAT solver already when only a few of the graph's edges are determined. For this purpose, we introduce *partially defined graphs*, the corresponding lexicographic ordering, and the algorithm MINCHECK that checks their minimality. If the symmetry propagator detects the current partially defined graph is not minimal, clauses are learned and added to the solver's collection of clauses. On partially defined graphs, the minimality check is not guaranteed to be complete, but eventually, when all edges are determined, it performs a complete minimality check. Consequently, we can guarantee that all generated graphs are minimal and unique up to isomorphism. MINCHECK also detects, in some cases, whether the current partially defined graph implies under minimality the existence or non-existence of further edges. This is done without increasing the encoding size. When the minimality check reveals that the current partially defined graph isn't minimal, this conflict is analyzed, and a suitable clause is added to the solver; also when an implied edge is detected, a unit clause is added.

We implemented a prototype version of SMS and tested it on two prominent problems from Extremal Graph Theory [6]. The first asks for graphs with a prescribed minimum *girth* (length of a shortest cycle) and the largest number of edges. The second asks for graphs whose *diameter* (largest distance between any two vertices) is 2 but decreases when any edge is deleted. Both are fundamental problems that have been studied for many decades [6, 14, 27]. We could verify some extremal numbers for the girth problem, and confirm the Simon-Murty Conjecture [7] on diameter-2-critical graphs with up to 18 vertices, improving upon the known bound of 11.

Our experimental results show that SMS exhibits an encouraging performance, particularly on unsatisfiable instances. We developed the testing algorithm MINCHECK from scratch, since existing methods based on various other forms of graph canonization cannot handle partially defined graphs [29]. The check often takes a significant amount of the solving time; here, we see ample room for improvement. However, the time which SMS spends on the SAT solving itself is significantly reduced in comparison to a static symmetry breaking.

Related Work

Dynamic symmetry breaking in the broader sense, where symmetry breaking constraints or clauses are added during the search, has a long history, see, e.g., [3, 13, 15, 21, 32]. More recently, this has also been combined with nogood (or clause) learning [9, 12, 34], using the fact that if a new clause/nogood is learned, and the symmetries of the problem are known, then one can propagate and learn further clauses/nogoods. Metin et al. [30] explored another way of dynamically utilizing symmetries in the context of SAT by considering the lexicographic order on the assignments themselves; the symmetries are computed by external tools or provided by the user before the SAT-solving and are then taken into account by the solver. Equipping a SAT solver with a special-purpose propagator has been explored before, e.g., by Liffiton and Maglalang [25] for cardinality constraints and by Gebser et al. [18] for digraph acyclicity. The SAT Modulo Theory (SMT) framework uses a similar approach, where the SAT solver interacts with a theory solver, which provides propagators for a first-order logic theory [5].

2 Preliminaries

Graphs. All considered graphs are undirected and simple (i.e., without parallel edges or self-loops). A *graph* G consists of set $V(G)$ of vertices and a set $E(G)$ of edges; we denote the edge between vertices $u, v \in V(G)$ by uv or equivalently vu . We write $G - e$ for the graph

obtained from G by deleting the edge e and $G - v$ for the graph obtained from G by deleting the vertex v . \mathcal{G}_n denotes the class of all graphs G with $V(G) = \{1, \dots, n\}$. A_G denotes the *adjacency matrix* of a graph $G \in \mathcal{G}_n$ where the element at row v and column u , denoted by $A_G[v][u]$, is 1 if $vu \in E$ and 0 otherwise. $A_G[v]$ denotes the v -th row of A_G . \mathcal{S}_n denotes the set of all permutations over $\{1, \dots, n\}$.

Graphs $G_1, G_2 \in \mathcal{G}_n$ are *isomorphic* if there is a permutation $\pi \in \mathcal{S}_n$ such that for all $1 \leq u < v \leq n$ we have $uv \in E(G_1)$ if and only if $\pi(u)\pi(v) \in E(G_2)$. $\pi(G)$ denotes the graph obtained from $G \in \mathcal{G}_n$ by the permutation $\pi \in \mathcal{S}_n$, where $E(\pi(G)) = \{\pi(u)\pi(v) : uv \in E(G)\}$. The total order \preceq is defined for $G, H \in \mathcal{G}_n$ by setting $G \preceq H$ if and only if $A_G[1]A_G[2] \dots A_G[n]$ is lexicographically smaller or equal to $A_H[1]A_H[2] \dots A_H[n]$. G is *lexicographically smaller* than H (in symbols $G \prec H$) if $G \preceq H$ and $G \neq H$. $G \in \mathcal{G}_n$ is *lexicographically minimal* or \preceq -*minimal* if $G \preceq \pi(G)$ for every $\pi \in \mathcal{S}_n$.

We will also consider the lexicographic ordering of pairs of vertices from $\{1, \dots, n\}$ where $(v_1, v_2) < (u_1, u_2)$ if and only if either (i) $v_1 < u_1$ or (ii) $v_1 = u_1$ and $v_2 < u_2$. We observe that $A[v_1][v_2]$ occurs before $A[u_1][u_2]$ in an adjacency matrix A if and only if $(v_1, v_2) < (u_1, u_2)$. We will say, that a vertex pair is *more important* than another if the vertex pair is smaller by this order. Similarly, we will say that an entry $A[v_1][v_2]$ of the adjacency matrix A is more important than $A[u_1][u_2]$ if $(v_1, v_2) < (u_1, u_2)$.

► **Observation 1.** Let $G, H \in \mathcal{G}_n$. Then $G \prec H$ if and only if there are $1 \leq i, j \leq n$ such that $A_G[i][j] = 0$, $A_H[i][j] = 1$, and for all more important pairs of vertices (i', j') the values of the adjacency matrices are equal, i.e., $A_G[i'][j'] = A_H[i'][j']$.

Formulas and Satisfiability. A *literal* is a propositional variable or negated propositional variable. A *clause* is a disjunction of literals. A formula in Conjunctive Normal Form (CNF) is a conjunction of clauses. A (*partial*) *assignment* is a function $f : X \rightarrow \{\text{true}, \text{false}\}$ defined on a set X of propositional variables. For a variable $x \notin X$ we say that f is *undefined*. Assignments extend to literals in an obvious way. A *model* of a CNF formula F is an assignment f defined on the variables of F such that each clause of F contains a literal that is set to true by f . A clause containing a single literal is *unity clause*.

3 Dynamic Symmetry Breaking in SMS

This section presents our method for dramatically reducing the search space for finding all graphs in \mathcal{G}_n modulo isomorphism, that satisfy a given property. As we deal with graphs from \mathcal{G}_n for some fixed n , we will use CNF formulas that contain all the propositional variables $e_{v,u}$, for $v < u$, which are true if the edge vu is present in the implicitly represented graph. Hence, we can extract a graph from a model of the formula.

Our aim is to decide during the CDCL SAT solver's run whether the current partial assignment can be extended to a model, such that the represented graph is \preceq -minimal. For this purpose we consider *partially defined graphs*, since during solving we don't know the final graph yet. A partially defined graph is a graph G where $E(G)$ is split into two disjoint sets $D(G)$ and $U(G)$. $D(G)$ contains the *defined* edges, $U(G)$ contains the *undefined* edges. A (fully defined) graph is a partially defined graph G with $U(G) = \emptyset$. Similarly to \mathcal{G}_n , let \mathcal{P}_n denote the class of all partially defined graphs G with $V(G) = \{1, \dots, n\}$. Analogously to the adjacency matrix of a fully defined graph, we define the adjacency matrix A_G of a partially defined graph $G \in \mathcal{P}_n$ as follows: $A_G[v_1][v_2] = 1$ if $v_1v_2 \in D(G)$, $A_G[v_1][v_2] = \star$ if $v_1v_2 \in U(G)$, and $A_G[v_1][v_2] = 0$ otherwise. From a partial assignment $f : X \rightarrow \{\text{true}, \text{false}\}$ we can extract the partially defined graph G with $V(G) = \{1, \dots, n\}$, $D(G) = \{ij : e_{i,j} \in X, f(e_{i,j}) = \text{true}\}$ and $U(G) = \{ij : e_{i,j} \notin X\}$.

A partially defined graph $G \in \mathcal{P}_n$ can be *extended* to a graph $H \in \mathcal{G}_n$ if $D(G) \subseteq E(H) \subseteq D(G) \cup U(G)$. We write $\mathcal{X}(G)$ for the set of all graphs to which G can be extended. A partially defined graph $G \in \mathcal{P}_n$ is \preceq -*minimal* if $\mathcal{X}(G)$ contains a \preceq -*minimal* graph.

A permutation $\pi \in \mathcal{S}_n$ is a *witness* of the non- \preceq -minimality of $G \in \mathcal{P}_n$ if $\pi(H) \prec H$ for all $H \in \mathcal{X}(G)$; observe that in that case G cannot be \preceq -minimal.

Let $G \in \mathcal{P}_n$, $\pi \in \mathcal{S}_n$, and (i, j) a vertex pair. We say (i, j) is (G, π) -*equal* if $A_H[i][j] = A_{\pi(H)}[i][j]$ for all $H \in \mathcal{X}(G)$ (i.e., $(i, j) \in \{(\pi(i), \pi(j)), (\pi(j), \pi(i))\}$, or $A_G[i][j] = A_{\pi(G)}[i][j] \neq \star$), and (i, j) is (G, π) -*critical* if $(A_G[i][j], A_{\pi(G)}[i][j]) \in \{(1, 0), (\star, 0), (1, \star)\}$.

Next we will introduce indicator pairs, which will be of crucial importance for checking whether a partially defined graph is \preceq -minimal. For $G \in \mathcal{P}_n$ and $\pi \in \mathcal{S}_n$ the vertex pair (i, j) is a (G, π) -*indicator pair* if (i, j) is (G, π) -critical and for every $(i', j') < (i, j)$ with $i' < j'$ at least one of the three cases holds: (i) $(i', j') \in \{(\pi(i'), \pi(j')), (\pi(j'), \pi(i'))\}$, or (ii) $A_G[i'][j'] = 1$, or (iii) $A_{\pi(G)}[i'][j'] = 0$. A (G, π) -indicator pair (i, j) is a *strict* (G, π) -indicator pair if $A_G[i][j] = 1$ and $A_{\pi(G)}[i][j] = 0$. A partially defined graph $G \in \mathcal{P}_n$ is *constraining* if there is a (G, π) -indicator pair for some $\pi \in \mathcal{S}_n$.

► **Proposition 2.** *Let $G \in \mathcal{P}_n$. If there is a strict (G, π) -indicator pair for some $\pi \in \mathcal{S}_n$ then G is not \preceq -minimal. Furthermore, if G is fully defined and not \preceq -minimal, then there is a strict (G, π) -indicator pair for some $\pi \in \mathcal{S}_n$.*

Proof. Let $G \in \mathcal{P}_n$ and (i, j) be a strict (G, π) -indicator pair. For the sake of a contradiction, assume that G is \preceq -minimal. By definition, there is a fully defined graph $H \in \mathcal{X}(G)$ which is \preceq -minimal. First, we show by induction over all more important vertex pairs (i', j') than (i, j) (including $i' > j'$) that $A_H[i'][j'] = A_{\pi(H)}[i'][j']$. We distinguish five cases.

1. If $i' > j'$ then $(j', i') < (i', j')$ hence $A_H[i'][j'] = A_H[j'][i'] = A_{\pi(H)}[j'][i'] = A_{\pi(H)}[i'][j']$ holds by induction hypothesis.
2. If $(i', j') = (\pi(i'), \pi(j'))$ then $A_{\pi(H)}[i'][j'] = A_H[\pi^{-1}(i')][\pi^{-1}(j')] = A_H[i'][j']$ holds.
3. If $(i', j') = (\pi(j'), \pi(i'))$ then $A_{\pi(H)}[i'][j'] = A_H[\pi^{-1}(i')][\pi^{-1}(j')] = A_H[j'][i'] = A_H[i'][j']$ holds.
4. If $A_G[i'][j'] = 1$ then $A_H[i'][j'] = 1$. By induction hypothesis, $A_{\pi(H)}[i''][j''] = A_H[i''][j'']$ holds for all more important vertex pairs (i'', j'') than (i', j') . Hence also $A_{\pi(H)}[i'][j'] = 1$ must hold, otherwise Observation 1 would be violated, so $A_H[i'][j'] = A_{\pi(H)}[i'][j']$.
5. If $A_{\pi(G)}[i'][j'] = 0$ then $A_{\pi(H)}[i'][j'] = 0$. Again, by induction hypothesis, $A_{\pi(H)}[i''][j''] = A_H[i''][j'']$ holds for all more important vertex pairs (i'', j'') than (i', j') . Hence also $A_H[i'][j'] = 0$ must hold, otherwise Observation 1 would be violated, so $A_H[i'][j'] = A_{\pi(H)}[i'][j']$.

So, we know that $A_{\pi(H)}[i'][j'] = A_H[i'][j']$ for all more important vertex pairs (i', j') than (i, j) and therefore, by Observation 1, H cannot be \preceq -minimal, consequently G cannot be \preceq -minimal in contradiction to our assumption.

For the second part of the proposition, let G be an arbitrary, non- \preceq -minimal, fully defined graph. Then, by definition of \preceq -minimality, there is a $\pi \in \mathcal{S}_n$ such that $\pi(G) < G$. Due to Observation 1, we know that there is a vertex pair (i, j) such that $A_G[i][j] = 1$, $A_{\pi(G)}[i][j] = 0$, and $A_{\pi(G)}[i'][j'] = A_G[i'][j']$ for all more important pairs (i', j') than (i, j) , hence either $A_G[i][j] = 1$ or $A_{\pi(G)}[i][j] = 0$, so (i, j) is a strict (G, π) -indicator pair. ◀

► **Observation 3.** *Let $G \in \mathcal{P}_n$, $H \in \mathcal{X}(G)$ a \preceq -minimal graph, and (i, j) a (G, π) -indicator pair for some $\pi \in \mathcal{S}_n$. Then $A_H[i][j] = A_{\pi(H)}[i][j]$.*

Observation 3 states that if (i, j) is a (G, π) -indicator pair and $A_G[i][j] = 1$ then we can imply that $A_{\pi(H)}[i][j] = 1$ for every \preceq -minimal graph $H \in \mathcal{X}(G)$, and if $A_{\pi(G)}[i][j] = 0$ then we can imply that $A_H[i][j] = 0$.

A key component for the symmetry propagator that we use within SMS is an algorithm that tests whether the partially defined graph $G \in \mathcal{P}_n$ as represented by the current partial assignment is \preceq -minimal, i.e., whether it can be extended to a \preceq -minimal fully defined graph. Performing this test is computationally hard in the worst case. For example, finding the lexicographically smallest isomorphic graph is known to be NP-hard [2].

We restrict our search for permutations π with a (G, π) -indicator pair. In the worst case, this test still requires considering all the $n!$ permutations of the vertices. However, we use a more sophisticated approach that exploits the partially defined graph's structure which often allows us to avoid the consideration of most of the permutations. Whenever during search, SMS has a new partial assignment which updates the partially defined graph G , it calls `MINCHECK`(G) which searches for an indicator pair. If a (G, π) -indicator pair is found, it extracts a clause representing the reason why G isn't \preceq -minimal or why a certain edge must be present or cannot be present in any \preceq -minimal extension of G .

We integrate this procedure into the conflict-driven clause learning algorithm (CDCL). Whenever the CDCL algorithm assigns an edge variable a truth value, we check the \preceq -minimality of the current partially defined graph G and add a clause if necessary. For efficiency, some minimality checks may be skipped. If the added clause is invalidated by the current partial assignment, then the current partial assignment must be discarded and the solver backtracks. Otherwise, the clause is a unit clause, so a literal can be propagated by Boolean constraint propagation. All the created clauses are added as learned clauses to the solver, so the solver's clause-deletion policy can discard them if they aren't needed anymore.

Below we present the `MINCHECK` algorithm in detail.

3.1 Minimality Check

Our minimality test is based on the concept of a *generalized ordered partition* (GOP) of $V = \{1, \dots, n\}$, which is a list of triples $P = [(V_1, l_1, u_1), \dots, (V_k, l_k, u_k)]$ such that $V_1 \cup \dots \cup V_k = V$, V_i, V_j are disjoint for $1 \leq i < j \leq k$, $u_i, l_i \in \{1, \dots, n\}$, $u_i + 1 = l_{i+1}$ for $1 \leq i \leq k-1$, and $|V_i| = u_i - l_i + 1$. Let f be the mapping that indicates to which set each vertex belongs to, i.e., $f(v) = i$ if and only if $v \in V_i$. Then we associate with a GOP P the set of permutations $\text{Perm}(P) = \{\pi \in \mathcal{S}_n : l_{f(v)} \leq \pi(v) \leq u_{f(v)} \text{ for all } v \in V\}$; i.e., the GOP gives a range for each vertex to which it can potentially be mapped.

Next we describe the algorithm `MINCHECK`. The input for the initial call of `MINCHECK` is a partially defined graph $G \in \mathcal{P}_n$. The idea is to start with the GOP $P = [(V, 1, n)]$ and refine it until we have found a (G, π) -indicator pair with some π represented by the GOP or can conclude that no such indicator pair exists. Therefore, we recursively assign a vertex v to r , i.e., $\pi(v) = r$ for all $\pi \in \text{Perm}(P)$, starting with $r = 1$ and adapt the GOP correspondingly by splitting up the triples (V_i, l_i, u_i) into multiple triples if necessary.

We use a recursive procedure `MINCHECK` with the following input: a partially defined graph $G \in \mathcal{P}_n$, a GOP $P = [(V_1, l_1, u_1), \dots, (V_k, l_k, u_k)]$, and a row $r \in \{1, \dots, n\}$. For all $j < r$ we have $|V_j| = 1$, in other words, all the vertices which are mapped to the first $r-1$ vertices are already fixed. Furthermore, we will see that all pairs up to $(r-1, n)$ are (G, π) -equal for all $\pi \in \text{Perm}(P)$. If $r = n$ we return nil.

Now we choose a $v \in V_r$: we adapt the GOP P to a GOP P_v such that $\pi(r) = v$ for all permutations represented by P_v and split up all triples (V_i, l_i, u_i) with $i \geq r$ (in the order they occur in the list) such that also the current row, (i.e., all pairs (r, j) with $j > r$) are (G, π) -equal for all permutations π represented by the GOP P_v or until we have found an indicator pair. In the former case, we call `MINCHECK`($G, P_v, r+1$) with the adapted GOP P_v and return an indicator pair if found; otherwise we backtrack. If every choice of $v \in V_r$ returned no indicator pair, then we return nil.

Before we stipulate how to split the triples in more detail, we mention some preconditions for MINCHECK which will allow us to argue that the preconditions are also satisfied at recursive calls. The preconditions are as follows:

- P1** The vertices which are mapped to the first $r - 1$ vertices are already determined, i.e., $|V_i| = 1$ for every $i < r$.
- P2** For every permutation $\pi \in \text{Perm}(P)$ every pair (i', j') with $i' < j', i' < r$ is equal under π .
- P3** For every $\pi \in \mathcal{S}_n \setminus \text{Perm}(P)$ with $\pi(k) \in V_k$ for $k < r$, there is some pair (i', j') with $i' < j', i' < r$ which is not (G, π) -equal and is more important than any (G, π) -critical vertex pair, hence there is no (G, π) -indicator pair.

First, we split (V_r, l_r, u_r) into $(\{v\}, l_r, l_r)$ and $(V_r \setminus \{v\}, l_r + 1, u_r)$, so $\pi(v) = r$ for all $\pi \in \text{Perm}(P_v)$. For each $(V_i, l_i, u_i), i \in \{r, \dots, n\}$, we apply the following steps, where $V_i^0 := \{u \in V_i : A_G[v][u] = 0\}$, $V_i^* := \{u \in V_i : A_G[v][u] = \star\}$, and $V_i^1 := \{u \in V_i : A_G[v][u] = 1\}$. (Some special care is needed for $i = r$ since it was already split, so we use $(V_r \setminus \{v\}, l_r + 1, u_r)$ instead of (V_r, l_r, u_r)):

1. We split (V_i, l_i, u_i) into the triples $(V_i^0, l_i, l_i + |V_i^0| - 1)$ and $(V_i^* \cup V_i^1, l_i + |V_i^0|, u_i)$; this ensures that the vertices not adjacent to v are mapped to the smallest vertices possible without violating the previous GOP.
2. If the set $J = \{u : A_G[r][u] \neq 0, l_i \leq u < l_i + |V_i^0|\} \neq \emptyset$, then we put $j = \min J$ and (r, j) is a (G, π) -indicator pair for every $\pi \in \text{Perm}(P_v)$. Otherwise $A_G[r][u] = A_{\pi(G)}[r][u] = 0$ for every $\pi \in \text{Perm}(P_v)$, hence Preconditions 2 and 3 hold up to all more important pairs than $(r, l_i + |V_i^0|)$.
3. Now we iterate over the remaining indexes $p \in [l_i + |V_i^0|, \dots, u_i]$. We distinguish between three cases:
 - a. $A_G[r][p] = \star$: Because all vertices from V_i^0 are mapped to smaller vertices than p , the only way to guarantee that the pair is equal (or critical) is to ensure that $(r, p) \in \{(\pi(r), \pi(p)), (\pi(r), \pi(p))\}$. In all other cases condition 2 does not hold up to all pairs including (r, p) . This can only be the case if
 - $v = r$ and $p \in V_i^*$, then we must split $(V_i^* \cup V_i^1, p, u_i)$ into the triples $(\{p\}, p, p)$ and $(V_i^* \setminus \{p\} \cup V_i^1, p + 1, u_i)$ and remove p from V_i^* , or
 - $v = p$ and $r \in V_i^*$, then we must split $(V_i^* \cup V_i^1, p, u_i)$ into the triples $(\{r\}, p, p)$ and $(V_i^* \setminus \{r\} \cup V_i^1, p + 1, u_i)$ and remove r from V_i^* .
 In all other cases, we backtrack.
 - b. If $A_G[r][p] = 0$ we backtrack, since $A_{\pi(G)}[r][p] \neq 0$ for every $\pi \in \text{Perm}(P_v)$.
 - c. If $A_G[r][p] = 1$, we again distinguish three cases:
 - If $V_i^* \neq \emptyset$ then we split $(V_i^* \cup V_i^1, p, u_i)$ into the triples $(V_i^*, p, p + |V_i^*| - 1)$ and $(V_i^1, p + |V_i^*|, u_i)$ so (r, p) is a (G, π) -indicator pair for all $\pi \in \text{Perm}(P_v)$.
 - If $V_i^* = \emptyset$ and $A_G[r][p'] \neq 1$ for some $p' \in \{p, \dots, u_i\}$, then we backtrack.
 - If $V_i^* = \emptyset$ and $A_G[r][p'] = 1$ for all $p' \in \{p, \dots, u_i\}$ then $A_G[r][p'] = A_{\pi(G)}[r][p']$ for all $\pi \in \text{Perm}(P_v)$. So, all more important pairs up to (r, u_i) are (G, π) -equal for all $\pi \in \text{Perm}(P_v)$.

One of the steps 1–3 will either produce an indicator pair or cause a backtrack to another $v \in V_r$. If all choices $v \in V_r$ have been exhausted unsuccessfully, we return nil.

► **Lemma 4.** *Let $G \in \mathcal{P}_n$ and $r \in \{1, \dots, n\}$. If the Preconditions P1–P3 are satisfied, MINCHECK($G, [(V_1, l_1, u_1), \dots, (V_k, l_k, u_k)], r$) will return a permutation π with $\pi(u) \in V_u$ for $u < r$ and a (G, π) -indicator pair if such a permutation and indicator pair exists; otherwise the procedure returns nil.*

Proof. In MINCHECK, we only backtrack if we can ensure, that (i) there is no indicator pair with $\pi(u) \in V_u$ for $u < r$ and $\pi(v) = r$ and (ii) whenever we find an indicator pair, we return it immediately. Precondition 2 ensures in all cases that it is indeed an indicator pair and Precondition 3 that we do not loose any potential candidates. Since the preconditions are satisfied for recursive calls, the lemma follows by induction. \blacktriangleleft

We can now put the above auxiliary results together and establish the MINCHECK algorithm's correctness and the fact that it performs a complete minimality check on fully defined graphs.

► **Theorem 5.** *Let $G \in \mathcal{P}_n$. If G is constraining, then MINCHECK(G) returns a permutation π and a (G, π) -indicator pair; otherwise MINCHECK(G) returns nil.*

Proof. MINCHECK($G, [(V, 1, n)], 1$) is called at the beginning, hence the proposition is true due to Lemma 4, since $\mathcal{S}_n = \text{Perm}([(V, 1, n)])$ and all preconditions are satisfied. \blacktriangleleft

Finally, we present how to extract a suitable clause from a partially defined graph $G \in \mathcal{P}_n$ and a (G, π) -indicator pair (i, j) .

Let $S = \{(i', j') : (i', j') \in \{(\pi(i'), \pi(j')), (\pi(j'), \pi(i'))\}\}$, $S_1 = \{(i', j') : (i', j') < (i, j), i' < j', A_G[i][j] = 1, (i', j') \notin S\}$ and $S_2 = \{(i', j') : (i', j') < (i, j), i' < j', A_{\pi(G)}[i][j] = 0, (i', j') \notin S\}$. Then

$$\bigvee_{(i', j') \in S_1} \neg e_{i', j'} \bigvee_{(i', j') \in S_2} e_{\pi^{-1}(i'), \pi^{-1}(j')} \vee \neg e_{i, j} \vee e_{\pi^{-1}(i), \pi^{-1}(j)}$$

is the resulting clause, which we add as learned clause. By Proposition 2, this clause must be satisfied by every assignment which represents a \preceq -minimal graph. We would like to stress that the created clause is satisfied by every partial assignment representing a \preceq -minimal graph; hence, the correctness of the symmetry breaking is independent of the found permutations and indicator pairs.

4 Static Symmetry Breaking

For comparison, we describe the static symmetry breaking approach, which is essentially the “improved lexicographic break” proposed by Codish et al. [10]. The SAT solver runs on a CNF formula $F \wedge M$, where F encodes the properties of the graph sought for, and M encodes a minimality property. A complete symmetry breaking would require a prohibitively large encoding size. Therefore, M just ensures that swapping any two vertices does not lead to a lexicographically smaller graph (if it would, the current graph can't be \preceq -minimal).

Swapping two vertices i, j leads to swapping in the adjacency matrix the elements $A[i][k]$ with $A[j][k]$ and $A[k][i]$ with $A[k][j]$, respectively, for $k \in \{1, \dots, n\} \setminus \{i, j\}$. All the other entries of the matrix will not change, because the diagonal only contains zeros and we have $A[i][j] = A[j][i]$. So we have to check whether swapping the entries does not lead to a \preceq -smaller graph. Following [10], we define an order on the rows $A[i]$ and $A[j]$, by setting $A[i] <_{i,j} A[j]$ if and only if $A[i]$ is lexicographically smaller than $A[j]$ while ignoring the i -th and j -th elements in both rows.

► **Proposition 6** ([10]). *If $G \in \mathcal{G}_n$ is \preceq -minimal, then $A_G[i] \leq_{i,j} A_G[j]$ for all $1 \leq i < j \leq n$.*

We can now define the formula M for the static symmetry break as

$$\bigwedge_{1 \leq i < j \leq n} \bigwedge_{k \in \{1, \dots, n\} \setminus \{i, j\}} \left(\bigwedge_{l \in \{1, \dots, k\} \setminus \{i, j\}} (e_{i,l} \leftrightarrow e_{j,l}) \rightarrow (e_{i,k} \rightarrow e_{j,k}) \right).$$

Instead of this direct SAT encoding, Codish et al. [10] used the solver BEE to encode the property stated in Proposition 6. BEE [31] compiles finite domain constraints to SAT while additionally applying transformations to simplify the encoding and optimize it.

5 Prototype Implementation and Experimental Setup

In this section, we will describe our experimental setup and some implementation details. As the SAT solver, we use Clingo [19, 20], an ASP solver containing a complete state-of-the-art CDCL SAT solver. Clingo comes with a C interface that supports rapid prototyping for developing custom propagators. We used Clingo's C-interface to integrate our implementation of MINCHECK into the solver. Our implementation is available at Zenodo [24].

The parameter *frequency* allows us to balance the time spent on the minimality check and the time spent by the SAT solver itself. If *frequency* has the value $1/q$, then MINCHECK is called only every q -th time an edge variable has been assigned.

Our experiments are run on a computer with Intel Xeon E5540 at 2.53 GHz, 24 GB RAM, under Ubuntu 18.04. We use Clingo 5.5.0 and all tests are executed with a single thread.

6 Extremal Graphs with Required Girth

A prominent research topic in Extremal Graph Theory [6] is the study of extremal graphs (i.e., graphs with the largest possible number of edges) on n vertices that exclude a given family \mathcal{F} of graphs as subgraphs. $\text{EX}(n, \mathcal{F})$ denotes the class of extremal graphs with that property, and $\text{ex}(n, \mathcal{F})$ denotes the number of edges of the graphs in $\text{EX}(n, \mathcal{F})$. The special case, where $\mathcal{F} = \mathcal{C}_k$, the family of cycles up to length k , has received much attention; for convenience, we write $f_k(n) = \text{ex}(n, \mathcal{C}_k)$. The *girth* of a graph G is the length of a shortest cycle in G (or ∞ if G is acyclic). Hence $\text{EX}(n, \mathcal{C}_k)$ contains precisely the edge-maximal graphs of girth $> k$. The base case of $k = 3$ has been settled over a century ago by Mantel's Theorem [27]: $f_3(n) = \text{ex}(n, \mathcal{C}_3) = \lfloor n^2/4 \rfloor$, where $\text{EX}(n, \mathcal{F})$ contains precisely the complete bipartite graph $K_{\lfloor n/2 \rfloor, \lfloor n/2 \rfloor}$. For the general case $k > 3$, however, no closed formula is known, and researchers have tried to compute $f_k(n)$ for small values of k [1, 10, 17, 35, 36], or at least provide lower and upper bounds.

Next, we describe a SAT encoding that produces for given integers n, m, k a propositional CNF formula $F(n, m, k)$. The formula is satisfiable if and only if there is a graph $G \in \text{EX}(n, \mathcal{C}_k)$ with m edges, and where we can construct G from the satisfying assignment. We will then evaluate the formula with our SMS-solver and report the experimental results.

6.1 Encoding

We state a useful result before we present the encoding for $F(n, m, k)$ where δ_G and Δ_G denote the minimum and maximum degree of a graph G , respectively.

► **Lemma 7** ([17]). *If G is a graph of girth ≥ 5 with n vertices and m edges, then $n \geq 1 + \Delta_G \cdot \delta_G \geq 1 + \delta_G^2$, $\delta_G \geq m - f_4(n - 1)$, and $\Delta_G \cdot n \geq 2m$.*

In particular, this applies to all graphs in $\text{EX}(n, \mathcal{C}_k)$ for $k \geq 4$. We also use the obvious inequality $\delta_G n \leq 2m$, which follows from the Handshaking Lemma, to discard some cases.

According to Lemma 7, we can compute for each pair n, m the set $I_{n,m}$ of possible intervals $[a, b]$ such that for each graph G with n vertices and m edges, we have $a \leq \delta_G \leq \Delta_G \leq b$ for some $[a, b] \in I$. We can add to $F(n, m, k)$ suitable cardinality constraints that ensure that vertex degrees belong to one of the intervals.

To guarantee that the resulting graph has girth $> k$, we use two methods: a basic one and an improved one. The *basic method* explicitly forbids that any subset of up to k vertices forms a cycle. The set of all possible cycles of length k can be described with some basic symmetry breaking as follows:

$$C_k = \{ (v_1, \dots, v_k) \in \{1, \dots, n\}^k : i \neq j \rightarrow v_i \neq v_j, v_1 = \min\{v_1, \dots, v_k\}, v_2 < v_k \}.$$

Taking v_1 as the minimum fixes a particular rotation of the cycle, requiring $v_2 < v_k$ fixes an orientation of the cycle. Now we add for each element of C_k the constraint that one edge of the corresponding cycle must not be present:

$$\bigwedge_{(v_1, \dots, v_k) \in C_k} (\neg e_{v_1, v_2} \vee \neg e_{v_2, v_3} \vee \dots \vee \neg e_{v_{k-1}, v_k} \vee \neg e_{v_k, v_1}).$$

For $k \leq 4$ this is a workable solution, but the *improved method* scales better for larger k . It is based on the following observation where $\text{dist}_G(u, v)$ denotes the length of a shortest path between vertices u and v in graph G .

► **Observation 8.** *A shortest cycle in a graph G containing the edge $uv \in E(G)$ has length $\text{dist}_{G-uv}(u, v) + 1$.*

Hence, we can enforce that for every edge uv , $\text{dist}_{G-uv}(u, v) + 1 \geq g$ for a required girth g . Therefore, we start at vertex i and mark all vertices adjacent to i in $G - ij$. In the next step, we additionally mark all vertices which are adjacent to already marked vertices. This will be repeated $g - 2$ times. If at the end the vertex j is marked, the girth is smaller than desired.

For the concrete encoding, we introduce propositional variables $\text{reached}_{i,j,k,s}$ for representing that vertex k can be reached in s steps from vertex i in the graph $G - ij$. Consequently

$$\begin{aligned} \text{reached}_{i,j,k,1} &= e_{i,k} \text{ for } k \in \{1, \dots, n\} \setminus \{i, j\} \text{ and} \\ \text{reached}_{i,j,k,s} &= \bigvee_{l \in V(G) \setminus \{k\}} (e_{l,k} \wedge \text{reached}_{i,j,l,s-1}) \text{ for } s \in \{2, \dots, g-2\}, k \in \{1, \dots, n\} \setminus \{i\}. \end{aligned}$$

If at any point vertex j is reached, the girth restriction is invalidated. Hence we can use the following encoding:

$$\text{girth} = \bigwedge_{i < j} \bigwedge_{s=2}^{g-2} (\neg \text{reached}_{i,j,j,s} \vee \neg e_{i,j}).$$

We further improve this encoding. If we start checking whether a vertex v is part of a cycle smaller than the given girth, i.e. we check whether $\text{dist}_{G-uv} + 1 \geq g$ for all $vu \in E$, then v cannot be on a cycle which is shorter than the girth g . So for all subsequent vertices $v' > v$, we only consider the graph $G - v$. This yields the following final encoding:

$$\begin{aligned} \text{reached}_{i,j,k,1} &= e_{i,k} \text{ for } k \in \{i+1, \dots, n\} \setminus \{j\} \text{ and} \\ \text{reached}_{i,j,k,s} &= \bigvee_{l \in V(G) \setminus \{k\}} (e_{l,k} \wedge \text{reached}_{i,j,l,s-1}) \text{ for } s \in \{2, \dots, g-2\}, k \in \{i+1, \dots, n\}. \end{aligned}$$

6.2 Results

We computed $f_k(n)$ for $k \in \{4, 5, 6\}$, and thereby verified known results [1, 10]. For fixed k and n we run SMS on the formulas $F(n, f_k(n), k)$ and $F(n, f_k(n) + 1, k)$; we performed separate runs for all the intervals in $I_{n,m}$ where m donates the number of edges. The first formula must be satisfiable for at least one interval in $I_{n, f_k(n)}$ while the second must be

unsatisfiable for every interval in $I_{n,f_k(n)+1}$. In some cases, we didn't need to compute $F(n, f_k(n) + 1, k)$, since already the bounds from Lemma 7 show the non-existence of a graph with $f_k(n) + 1$ edges.

In general, for fixed k and n we run SMS on $F(n, m, k)$ for different values of m , starting from a lower bound obtained by Lemma 7. As long as $F(n, m, k)$ is satisfiable, we increment m by one and repeat until we arrive at a value for which $F(n, m, k)$ is unsatisfiable or we can apply Lemma 7. Then we know that $f_k(n) = m - 1$.

Table 1 shows our results for $k = 4$ and $n \in \{15, \dots, 28\}$. We use the basic method to encode the girth requirement and choose a frequency of $1/5$. For all tables in the current section, the runtimes are given in seconds; for SMS we provide in parenthesis the fraction of the total time spent for the minimality check. The columns labeled sat give the minimal time over all intervals in $I_{n,m}$; the columns labeled unsat give the maximum. An entry n/a indicates that the unsatisfiability check is covered by Lemma 7; t.o. indicates that the timeout of 4 hours has been reached without producing a result.

■ **Table 1** Results for $f_4(n)$.

n	$f_4(n)$	sat		unsat	
		SMS	Static	SMS	Static
15	26	0.11(67%)	1.26(0.30)	n/a	n/a
16	28	0.07(59%)	0.56(0.61)	0.91(71%)	529.42(32.20)
17	31	0.13(66%)	0.58(0.48)	n/a	n/a
18	34	0.08(53%)	2.80(0.60)	n/a	n/a
19	38	0.11(53%)	1.06(0.57)	n/a	n/a
20	41	2.41(73%)	2457.85(161.41)	n/a	n/a
21	44	0.20(61%)	1.90(151.84)	0.98(72%)	7319.96(1019.41)
22	47	1.28(72%)	3.44(16.69)	11.74(74%)	t.o.(t.o.)
23	50	2.95(79%)	1.58(367.83)	177.11(75%)	t.o.(t.o.)
24	54	30.91(74%)	638.37(80.74)	n/a	n/a
25	57	193.68(72%)	204.00(655.66)	t.o.	t.o.(t.o.)
26	61	74.63(74%)	t.o.(168.90)	n/a	n/a
27	65	1270.38(68%)	t.o.(193.44)	n/a	n/a
28	68	37.84(75%)	t.o.(t.o.)	t.o.	t.o.(t.o.)

We would like to emphasize that the purpose of the experiments is not to identify which algorithm is the fastest but rather to gain insights into the potential of a dynamic symmetry breaking for graph generation. We provide for reference the running times of our encoding of the static symmetry breaking (columns labeled Static) and the times reported by Codish et al. [10] with their “improved lexicographic break” (given in parentheses) for the same problems. This is not meant as a direct comparison, as the results by Codish et al. [10] have been run on different hardware, but just to give a rough idea on the order of magnitude the two approaches take. It is not completely clear how Codish et al. combined runtimes over all intervals $I_{n,m}$ into one single result. This has no impact on the unsat-times, because for those there is only a single interval in $I_{n,m}$.

We can see that SMS is significantly faster for the unsatisfiable instances. For example, SMS determines the unsatisfiable case for $n = 22$ quickly, although the static approach reached the timeout. SMS could also establish the unsatisfiability case for $n = 23$. We see that SMS uses a large fraction of the time for the minimality check. Therefore, a speedup for the check would have a significant impact on the runtime.

■ **Table 2** Results for $f_5(n)$ and $f_6(n)$.

n	$f_5(n)$	sat	unsat	$f_6(n)$	sat	unsat
15	22	0.25(15%)	5.40(14%)	18	0.24(6%)	3.28(9%)
16	24	0.25(9%)	0.66(15%)	20	0.60(8%)	n/a
17	26	0.59(15%)	1.56(13%)	22	1.14(9%)	n/a
18	29	0.54(15%)	n/a	23	1.09(8%)	61.35(5%)
19	31	9.30(11%)	8.12(9%)	25	2.34(6%)	n/a
20	34	13.08(9%)	n/a	27	2.36(7%)	n/a
21	36	3.87(9%)	97.16(6%)	29	11.33(5%)	n/a
22	39	22.49(7%)	n/a	31	16.37(4%)	n/a
23	42	9.49(7%)	n/a	33	10.46(5%)	n/a
24	45	56.01(6%)	n/a	36	3.59(5%)	n/a
25	48	50.55(6%)	n/a	37	17.84(4%)	t.o.
26	52	21.08(6%)	n/a	39	12.05(5%)	174.83(3%)
27	53	40.13(5%)	t.o.	41	217.73(3%)	449.78(3%)
28	56	25.35(5%)	376.81(5%)	43	4961.74(2%)	1245.43(3%)

■ **Table 3** Results for different frequencies.

frequency	$n = 27$ (sat)	$n = 23$ (unsat)
1/1	t.o.	240.00(94.25%)
1/2	2514.35(85.55%)	140.06(88.67%)
1/5	1293.92(67.36%)	103.72(72.46%)
1/10	208.26(51.44%)	96.99(50.79%)
1/20	782.71(30.03%)	125.25(29.45%)
1/50	636.99(12.72%)	243.30(13.63%)
1/100	4454.68(5.23%)	214.43(6.51%)
1/200	3860.19(2.80%)	608.77(3.06%)
1/500	t.o.	774.36(0.97%)
1/1000	t.o.	1103.35(0.46%)

Codish et al. [10] used some further methods to improve their results, i.e., they included embedded stars in their graphs, leading to a significant speedup. We do not use these improvements in our experiments.

Next, we report on results for computing $f_k(n)$ for $k \in \{5, 6\}$. For these cases, we used the girth-constraints based on edge-removal from Section 6.1. In these experiments, we see that far less time is spent on the minimality check than in the previous experiments, although there we had a frequency of 1/5. Most likely, the reason is the additionally created variables for the girth-constraints, because the minimality check is only called when a variable $e_{i,j}$ is assigned. The results are shown in Table 2.

Table 3 shows the influence of the parameter frequency on SMS's performance. For this analysis, we took the unsatisfiable case for $f_4(23)$ with the degree interval $[4, 5]$ and the satisfiable case for $f_4(27)$ with the degree interval $[4, 6]$.

Interestingly, the frequency shows a very clear pattern. Up to a frequency of 1/10, the runtime decreases and then increases again. The reason seems to be the high fraction of the time spent in MINCHECK for a high frequency and possibly the increased number of added clauses.

7 Application: Diameter-2-Critical Graphs

The *diameter* of a graph G is the largest distance among all pairs of vertices in G , where the *distance* of two vertices is the length of a shortest path between them. A disconnected graph has diameter ∞ . A graph is *diameter- d -critical* if its diameter is d , but the deletion of any edge decreases the diameter. The study of extremal properties of graphs with prescribed diameter has been initiated by Erdős and Rényi in the early 1960s [14] and has been the subject of intensive research. An important topic in the field is the characterization of diameter- d -critical graphs [7, 8, 23, 26]. An intriguing open problem is whether the Simon-Murty Conjecture [7] holds, which states that if G is a diameter-2-critical graph with n vertices and m edges, then $m \leq \lfloor n^2/4 \rfloor$, with equality precisely for the complete bipartite graph $K_{\lfloor n/2 \rfloor, \lfloor n/2 \rfloor}$ (i.e., similar to Mantel’s Theorem mentioned above).

Using the list of non-isomorphic graphs generated with Nauty [29], Radosavljević and Živković [33] computed all diameter-2-critical graphs with up to 10 vertices. Also, Dailly et al. [11] report on a “computer search” for graphs with up to 11 vertices, focusing on graphs with a certain number of edges. With SMS we were able to extend these results to graphs with 12 vertices. The basis for this computation is a SAT encoding that produces for given integers n and m a propositional CNF formula $D_2(n, m)$ which is satisfiable if and only if there is a diameter-2-critical graph G with n vertices and m edges. As above, one can construct G from the satisfying assignment.

7.1 Encoding

Equivalently to the above definition, a graph is diameter-2-critical if and only if (i) its diameter is at most 2 and (ii) when any edge is deleted, the diameter is larger than 2. We observe that property (i) allows graphs with diameter one, i.e., complete graphs. However, after deleting any edge, the diameter would still be at most 2 for $n > 3$, which violates property (ii).

Our encoding of $D_2(n, m)$ handles both properties separately. To encode property (i), we use

$$\bigwedge_{1 \leq i < j \leq n} (e_{i,j} \vee \bigvee_{1 \leq k \leq n} (e_{i,k} \wedge e_{j,k})).$$

For property (ii), we first define a subformula $N(i, j, c)$ which encodes that vertices i and j have exactly c common neighbors, so $N(i, j, c) \leftrightarrow |\{k \in V : e_{i,k} \wedge e_{j,k}\}| = c$. We can use cardinality constraints to encode this (see, e.g., [4]) or directly use features of the ASP solver Clingo to express the cardinality constraints. With the help of this subformula, we can encode property (ii) as follows:

$$\bigwedge_{1 \leq i < j \leq n} e_{i,j} \rightarrow \left(N(i, j, 0) \vee \bigvee_{1 \leq k \leq n} (e_{i,k} \wedge N(j, k, 1)) \vee \bigvee_{1 \leq k \leq n} (e_{j,k} \wedge N(i, k, 1)) \right).$$

If $N(i, j, 0)$ is satisfiable, then $\text{dist}_{G-ij}(i, j) > 2$; in the other cases, either $\text{dist}_{G-ij}(j, k) > 2$ or $\text{dist}_{G-ij}(i, k) > 2$.

7.2 Results

We use the encoding $D_2(n, m)$ to enumerate all \preceq -minimal, diameter-2-critical graphs in \mathcal{G}_n . Therefore, we run SMS repeatedly; each time we find a new graph, we explicitly exclude it from the search space until no further graph is found. Additionally, we abort the minimality check after a certain number of steps, because there are some rare cases, where the check takes far too long. We use a frequency of 1/5.

Table 4 shows the results of this computation. Column #-sol gives the number of solutions found; column time gives the runtime in seconds; as above, the percentage of the runtime that has been spent for the minimality check is given in parenthesis. Column Static gives the number of solutions found with the static symmetry breaking method, without filtering isomorphic solutions. The static version could not find all solutions for $n = 12$ within 4 hours.

■ **Table 4** Results for generating all diameter-2-critical graphs with $n \leq 10$.

n	SMS		Static	
	#-sol	time	#-sol	time
3	1	0.11(0%)	1	0.01
4	2	0.11(1%)	2	0.01
5	3	0.12(4%)	4	0.02
6	5	0.15(12%)	11	0.05
7	10	0.26(40%)	32	0.14
8	30	0.83(67%)	163	0.82
9	103	2.53(73%)	1018	6.62
10	519	10.00(63%)	9727	149.20
11	3746	80.47(47%)	133316	9214.83
12	40866	1338.09(33%)	t.o.	t.o.

When using a cutoff within SMS, we cannot guarantee that all the resulting graphs are unique up to isomorphism. Nevertheless, a check with Nauty showed that indeed all the computed graphs are unique. The number of solutions for $n \in \{11, 12\}$, stated in boldface, were unknown, as this goes beyond a generate-and-test approach.

Checking the computed graphs, we could confirm the Simon-Murty Conjecture for graphs with up to 12 vertices. By a minor adaption of the encoding, i.e., enforcing that the number of edges is $\geq \lfloor n^2/4 \rfloor$, we could extend this to $n = 13$. If we know the degree of the vertices in advance, we can create an initial GOP for the minimality check, such that only vertices with the same degree can be permuted. So, we can use SMS for every possible combination of vertex degrees. Trivially, all cases where a vertex has degree 1 can be excluded. Additionally, we can use the following theorem by Fan [16] to discard further combinations in advance ($d_G(v)$ denotes the degree of vertex v in G):

► **Theorem 9** ([16]). *If G is a diameter-2-critical with n vertices and m edges, then $\sum_{v \in V(G)} d_G(v)^2 \leq \frac{4}{15}n^3$. If $n \leq 24$ or $n = 26$, then $m \leq \lfloor n^2/4 \rfloor$.*

Consequently, since $\sum_{v \in V(G)} d_G(v) = 2m$, Theorem 9 limits the combinations of vertex degrees. Adding these degree constraints, we could confirm the conjecture for $n \in \{14, \dots, 17\}$. For the case $n = 18$, we used an additional theorem to further restrict the combinations:

► **Theorem 10** ([22]). *If G is a diameter-2-critical with n vertices and maximum degree $\geq 0.7 \cdot n$, then G has fewer than $\lfloor n^2/4 \rfloor$ edges.*

We give some details on the computation in Table 5.

► **Corollary 11.** *The Simon-Murty Conjecture holds for graphs with up to 18 vertices.*

■ **Table 5** Confirming the Simon-Murty Conjecture for $n \in \{14, \dots, 18\}$. Column n denotes the number of vertices, column #-comb the number of degree combinations, max-time the maximal runtime of a single combination, and total-time the accumulated runtime over all combinations. All times are given in seconds.

n	total time	max-time	#-comb
14	14512	131	1021
15	156116	604	4319
16	923660	2847	6494
17	11700237	19582	24067
18	46612962	216384	12974*

8 Conclusion

We presented SMS, a novel approach for SAT-based graph generation that utilizes dynamic symmetry breaking. A key ingredient of SMS is the concept of partially defined graphs and an algorithm that checks the lexicographical minimality of such graphs. We evaluated a prototype implementation of SMS on two showcase problems from Extremal Graph Theory, related to the graph invariants girth and diameter, respectively. We compared SMS with static symmetry breaking. We used the same encoding for the graph property and the same underlying SAT solver for both approaches. We think that this double strategy might be of independent interest, as it supports comparing the very same SAT-encoding on both methods. The experiments show encouraging results for SMS, in particular on unsatisfiable instances. As a side effect of our experiments on diameter-2-critical graphs, we could compute some values that haven't been known before and confirm the Simon-Murty Conjecture for graphs with up to 18 vertices.

We see several avenues for improving SMS in the future. An obvious area for improvement is the minimality check, where we currently use a relatively simple algorithm written from scratch. This leaves much room for improvement for algorithm design and engineering. The parameter frequency which controls the calls to the minimality check is currently a static parameter that stays constant for an entire SMS run. Here, dynamic changes of this parameter that depend on the current state of the solving progress could significantly increase the efficiency of SMS.

References

- 1 E. Abajo and A. Diánez. Exact value of $ex(n; \{C_3, \dots, C_s\})$ for $n \leq \lfloor \frac{25(s-1)}{8} \rfloor$. *Discrete Math.*, 185:1–7, 2015. doi:10.1016/j.dam.2014.11.021.
- 2 Vikraman Arvind, Bireswar Das, and Johannes Köbler. A logspace algorithm for partial 2-tree canonization. In Edward A. Hirsch, Alexander A. Razborov, Alexei L. Semenov, and Anatol Slissenko, editors, *Computer Science - Theory and Applications, Third International Computer Science Symposium in Russia, CSR 2008, Moscow, Russia, June 7-12, 2008, Proceedings*, volume 5010 of *Lecture Notes in Computer Science*, pages 40–51. Springer, 2008. doi:10.1007/978-3-540-79709-8_8.
- 3 Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. *Constraints*, 7(3-4):333–349, 2002. doi:10.1023/A:1020533821509.
- 4 Olivier Bailleux and Yacine Bouffkhad. Efficient CNF encoding of boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003. doi:10.1007/978-3-540-45193-8_8.

- 5 Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1330. IOS Press, second edition, 2021.
- 6 Béla Bollobás. *Extremal graph theory*. Academic Press, 1978.
- 7 Louis Caccetta and Roland Häggkvist. On diameter critical graphs. *Discrete Math.*, 28(3):223–229, 1979. doi:10.1016/0012-365X(79)90129-8.
- 8 Ya-Chen Chen and Zoltán Füredi. Minimum vertex-diameter-2-critical graphs. *J. Graph Theory*, 50(4):293–315, 2005. doi:10.1002/jgt.20111.
- 9 Geoffrey Chu, Maria Garcia de la Banda, Christopher Mears, and Peter J. Stuckey. Symmetries, almost symmetries, and lazy clause generation. *Constraints*, 19(4):434–462, 2014. doi:10.1007/s10601-014-9163-9.
- 10 Michael Codish, Alice Miller, Patrick Prosser, and Peter J. Stuckey. Constraints for symmetry breaking in graph representation. *Constraints*, 24(1):1–24, 2019. doi:10.1007/s10601-018-9294-5.
- 11 Antoine Dailly, Florent Foucaud, and Adriana Hansberg. Strengthening the Murty-Simon conjecture on diameter 2 critical graphs. *Discrete Math.*, 342(11):3142–3159, 2019. doi:10.1016/j.disc.2019.06.023.
- 12 Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. Symmetric explanation learning: Effective dynamic symmetry handling for SAT. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 83–100. Springer Verlag, 2017. doi:10.1007/978-3-319-66263-3_6.
- 13 Jo Devriendt, Bart Bogaerts, Broes De Cat, Marc Denecker, and Christopher Mears. Symmetry propagation: Improved dynamic symmetry breaking in SAT. In *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*, pages 49–56. IEEE Computer Society, 2012. doi:10.1109/ICTAI.2012.16.
- 14 P. Erdős and A. Rényi. On a problem in the theory of graphs. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 7:623–641 (1963), 1962.
- 15 Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In Toby Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*, pages 93–107. Springer Verlag, 2001. doi:10.1007/3-540-45578-7_7.
- 16 Genghua Fan. On diameter 2-critical graphs. *Discret. Math.*, 67(3):235–240, 1987. doi:10.1016/0012-365X(87)90174-9.
- 17 David K. Garnick, Y. H. Harris Kwong, and Felix Lazebnik. Extremal graphs without three-cycles or four-cycles. *J. Graph Theory*, 17(5):633–645, 1993. doi:10.1002/jgt.3190170511.
- 18 Martin Gebser, Tomi Janhunen, and Jussi Rintanen. SAT modulo graphs: Acyclicity. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, volume 8761 of *Lecture Notes in Computer Science*, pages 137–151. Springer Verlag, 2014. doi:10.1007/978-3-319-11558-0_10.
- 19 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with Clingo 5. In Manuel Carro, Andy King, Neda Saeedloei, and Marina De Vos, editors, *Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016 TCs, October 16-21, 2016, New York City, USA*, volume 52 of *OASICS*, pages 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/OASICS.ICLP.2016.2.
- 20 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014. arXiv:1405.3694.

- 21 Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 329–376. Elsevier, 2006.
- 22 Teresa Haynes, Michael Henning, Lucas Merwe, and Anders Yeo. A maximum degree theorem for diameter-2-critical graphs. *Open Mathematics*, 12(12):1882–1889, 2014. doi:10.2478/s11533-014-0449-3.
- 23 Teresa W. Haynes, Michael A. Henning, Lucas C. van der Merwe, and Anders Yeo. Progress on the Murty-Simon Conjecture on diameter-2 critical graphs: a survey. *J. Comb. Optim.*, 30(3):579–595, 2015. doi:10.1007/s10878-013-9651-7.
- 24 Markus Kirchweber and Stefan Szeider. SAT modulo symmetries for graph generation, 2021. doi:10.5281/zenodo.5170575.
- 25 Mark H. Liffiton and Jordyn C. Maglalang. A cardinality solver: More expressive constraints for free - (poster presentation). In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 485–486. Springer Verlag, 2012. doi:10.1007/978-3-642-31612-8_47.
- 26 Po-Shen Loh and Jie Ma. Diameter critical graphs. *J. Combin. Theory Ser. B*, 117:34–58, 2016. doi:10.1016/j.jctb.2015.11.004.
- 27 W. Mantel. Problem 28. *Wiskundige Opgaven*, 10:60–61, 1907.
- 28 João Marques-Silva and Sharad Malik. Propositional SAT solving. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 247–275. Springer, 2018. doi:10.1007/978-3-319-10575-8_9.
- 29 Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symbolic Comput.*, 60:94–112, 2014. doi:10.1016/j.jsc.2013.09.003.
- 30 Hakan Metin, Souheib Baarir, Maximilien Colange, and Fabrice Kordon. CDCLSym: introducing effective symmetry breaking in SAT solving. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2018. doi:10.1007/978-3-319-89960-2_6.
- 31 Amit Metodi and Michael Codish. Compiling finite domain constraints to SAT with BEE. *Theory Pract. Log. Program.*, 12(4-5):465–483, 2012. doi:10.1017/S1471068412000130.
- 32 Jean-François Puget. Symmetry breaking using stabilizers. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 585–599. Springer Verlag, 2003. doi:10.1007/978-3-540-45193-8_40.
- 33 Jovan Radosavljević and Miodrag Živković. The list of diameter-2-critical graphs with at most 10 nodes. *IPSI Trans. Adv. Res.*, 16(1):1–5, 2020. URL: <http://ipsitransactions.org/journals/papers/tar/2020jan/p9.pdf>.
- 34 Bas Schaafsma, Marijn Heule, and Hans van Maaren. Dynamic symmetry breaking by simulating Zykov contraction. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 223–236. Springer Verlag, 2009. doi:10.1007/978-3-642-02777-2_22.
- 35 Jianmin Tang, Yuqing Lin, Camino Balbuena, and Mirka Miller. Calculating the extremal number $\text{ex}(v; \{C_3, C_4, \dots, C_n\})$. *Discr. Appl. Math.*, 157(9):2198–2206, 2009. doi:10.1016/j.dam.2007.10.029.
- 36 P. Wang, G. W. Dueck, and S. MacMillan. Using simulated annealing to construct extremal graphs. *Discrete Math.*, 235(1-3):125–135, 2001. Combinatorics (Prague, 1998). doi:10.1016/S0012-365X(00)00265-X.

Counterfactual Explanations via Inverse Constraint Programming

Anton Korikov ✉

Department of Mechanical & Industrial Engineering, University of Toronto, Canada

J. Christopher Beck ✉

Department of Mechanical & Industrial Engineering, University of Toronto, Canada

Abstract

It is increasingly recognized that automated decision making systems cannot be black boxes: users require insight into the reasons that decisions are made. Explainable AI (XAI) has developed a number of approaches to this challenge, including the framework of counterfactual explanations where an explanation takes the form of the minimal change to the world required for a user's desired decisions to be made. Building on recent work, we show that for a user query specifying an assignment to a subset of variables, a counterfactual explanation can be found using inverse optimization. Thus, we develop inverse constraint programming (CP): to our knowledge, the first definition and treatment of inverse optimization in constraint programming. We modify a cutting plane algorithm for inverse mixed-integer programming (MIP), resulting in both pure and hybrid inverse CP algorithms. We evaluate the performance of these algorithms in generating counterfactual explanations for two combinatorial optimization problems: the 0-1 knapsack problem and single machine scheduling with release dates. Our numerical experiments show that a MIP-CP hybrid approach extended with a novel early stopping criteria can substantially out-perform a MIP approach particularly when CP is the state of the art for the underlying optimization problem.

2012 ACM Subject Classification Computing methodologies → Planning and scheduling

Keywords and phrases Explanation, Inverse Optimization, Scheduling

Digital Object Identifier 10.4230/LIPIcs.CP.2021.35

Funding This research was supported by the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

As automated decision making systems continue to make high-impact decisions [9], the need to provide insight into why decisions were made has become crucial. Black box solvers are becoming less and less acceptable. In fact, recent EU legislation [19] argues that users substantially affected by an automated decision have a right to an explanation. As a consequence, there has been a surge of research aimed at explaining algorithmic decisions to users, particularly in machine learning (ML) [3]. In contrast, in constraint programming (CP) and mathematical programming, work on explaining decisions has been more limited, with the majority of explainability research focused on explaining infeasibility through the identification of minimal sets of infeasible constraints [11, 5]. We introduce new techniques based on counterfactual explanations to explain optimal decisions made by discrete constraint-based optimization systems.

For example, consider a manufacturer placing several orders for part deliveries, specifying priority values on each order. After seeing the initial schedule, the manufacturer wants an explanation. They ask “Why was the schedule not different? Why is order A not delivered in two days, and B in five days?”



© Anton Korikov and J. Christopher Beck;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 35; pp. 35:1–35:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We address such questions using *counterfactual explanations*, which present the questioner with the minimal change to the world required for the hypothetical situation described in their query to have occurred. In previous work [16], we formalized a counterfactual explanation as an optimization problem, which we showed to be a generalization of inverse optimization [13]. We then developed solution methods for single variable questions and explanations.

In this paper, we present the first multi-variate counterfactual explanation approach for discrete optimization systems with linear objectives, showing that we can harness inverse optimization algorithms when the questioner is interested in a partial assignment of decision variables, such as in the delivery example above. In Section 3, we formulate this kind of explanation as a *Partial Assignment Nearest Counterfactual Explanation* (PA-NCE) problem. In Section 4, we prove that the PA-NCE can be solved in two steps by first finding an optimal solution to the original decision problem with the addition of the user’s assignments and then solving an inverse optimization problem.

We then turn our attention to discrete inverse optimization algorithms, used for the second stage of generating explanations but also applicable to other inverse problems. In Section 5 we develop three new inverse algorithms by modifying an existing mixed integer programming (MIP) cutting plane algorithm [26] to use CP. This results in one pure inverse CP algorithm and two MIP-CP hybrids. To our knowledge, this marks the first use of CP for inverse optimization. We also introduce a novel early stopping criteria, showing it is beneficial for inverse CP. We then show through numerical experiments in Sections 6 – 8 that a hybrid MIP-CP approach extended with our early stopping criteria can outperform alternatives when CP is state-of-the-art for the initial problem. The final sections discuss limitations and related work.

2 Background

2.1 Counterfactual Explanations

In a counterfactual explanation [25], a user would like to know why a set of facts c led to a decision x . They first ask a *contrastive question* “Why was the decision x and not \bar{x} ?” A *counterfactual explanation* presents the user with an alternative set of facts d , typically minimally different from the initial facts c , which would have resulted in decision \bar{x} . The term *counterfactual* (meaning contrary to the facts) refers to the observation that neither \bar{x} nor d were present in the initial (or, *factual*) context, and originates in studies of counterfactual reasoning [8]. An advantage of using counterfactual explanations for automated decision making systems is that the user is not required to understand the inner workings of the algorithm; a significant benefit for interacting with complex solvers. Further, explicitly presenting a user with a set of facts that would have led to a different outcome not only helps them understand the decision, but also empowers them to contest or act to change it [25].

As in the Explainable AI (XAI) literature [18], we refer to a counterfactual decision \bar{x} specified by a user as a *foil*. In the formulation developed in our previous work [16], the user is interested in multiple counterfactual decisions, implicitly described by a *foil set*, a feasible set, X_ψ . We studied the question “Why was the decision x and not one of the decisions in X_ψ ?”, and defined the problem of generating explanations to such questions as the *Nearest Counterfactual Explanation* (NCE) problem.

2.2 Nearest Counterfactual Explanation (NCE)

In a standard (or *forward*) optimization problem $\langle c, f, X \rangle$, the purpose is to find values for decision vector $x \in X \subseteq \mathbb{R}^n$ given a parameter vector $c \in \mathcal{C} \subseteq \mathbb{R}^n$ which optimize an objective function $f : \mathcal{C} \times X \rightarrow \mathbb{R}$. For a minimization objective, the goal is to find an optimal x^* so that $f(c, x^*) = \min_x \{f(c, x) : x \in X\}$. In this paper, we focus on problems with linear objectives and either binary decision variables, $x \in \{0, 1\}^n$, or integer decision variables, $x \in \mathbb{N}_0^n$. If f is omitted from a forward problem $\langle c, X \rangle$, it is assumed that $f(c, x) = c^T x$.

The Nearest Counterfactual Explanation (NCE) problem [16] starts with x^* , an optimal solution to the forward problem $\langle c, f, X \rangle$ for which the user requires an explanation. Specifically, the user wants an explanation of why the solution did not also satisfy an additional set of constraints, not initially captured in X . Let these additional constraints describe a feasible set $\psi \subseteq \mathbb{R}^n$ and be called *foil constraints*, and assume that $x^* \notin \psi$. For example, if a user asks “Why was I not scheduled to receive my COVID-19 vaccine in April rather than June?”, the foil constraints restrict the vaccination appointment to be in April.

The user is interested in those solutions in the intersection of X and ψ and described by the foil set: $X_\psi = X \cap \psi$. The counterfactual question asked by the user is then “Why is the solution x^* and not one of the foils in X_ψ ?”

The explanation problem is to find the minimal change to the initial parameter vector c which would make a foil in X_ψ optimal for the forward problem. If $d \in \mathcal{C}$ is the modified parameter vector, where \mathcal{C} can be used to express any restrictions on the feasible values of d , and $\|\cdot\|$ is some norm, the NCE problem $\langle c, \mathcal{C}, f, \psi, X, \|\cdot\| \rangle$ [16] is

$$\min_{d \in \mathcal{C}} \|d - c\| \tag{1}$$

$$\text{s.t. } \min_{x \in X_\psi} f(d, x) = \min_{x \in X} f(d, x). \tag{2}$$

Throughout this paper, we use an L_1 norm, and assume that none of the parameters in c are present in the constraints that define the feasible set X , meaning that our explanations only involve changes to objective parameters, and not constraint parameters.

We previously showed that the NCE is a generalization of *inverse optimization*, proposing that, for many problems, inverse optimization algorithms may applied [16]. However, we did not develop inverse optimization approaches, instead considering a restricted set of problems where only one parameter is allowed to change and inverse optimization is unnecessary. Our work here lifts this single variable restriction and develops a method to generate multi-variate explanations using full inverse optimization.

2.3 Inverse Optimization

While forward optimization seeks a variable assignment that satisfies a set of constraints and optimizes an objective function, inverse optimization tries to find the minimal change in the objective function such that a given feasible variable assignment is optimal. Given a forward problem $\langle c, f, X \rangle$, and a feasible solution $x^d \in X$, the inverse optimization problem is to find the minimal modification to the parameter vector c so that x^d becomes optimal [6].

As in the NCE, if $d \in \mathcal{C}$ is the modified parameter vector, then the inverse optimization problem $\langle c, \mathcal{C}, f, x^d, X, \|\cdot\|_1 \rangle$ is

$$\min_{d \in \mathcal{C}} \|d - c\|_1 \tag{3}$$

$$\text{s.t. } f(d, x^d) = \min_{x \in X} f(d, x). \tag{4}$$

The NCE is a generalization of the inverse optimization problem. In the latter, we find the values of d which make a *single* given solution optimal, while in the former, we find a d such that a *set* of solutions contains an optimal one.

Inverse Mixed Integer Programming. Much of the work in inverse optimization has focused on inverse linear optimization, where the forward problem is a linear program [13]. The work on inverse mixed integer optimization is relatively sparse with a cutting plane algorithm, *InvLP-MIP*, being the standard solution technique [26]. In Section 5, we modify this algorithm to use CP and an early stopping criteria. *InvLP-MIP* is an iterative, two-level approach where a master problem searches for a d that minimizes $\|d - c\|_1$ such that x^d is at least as good as all currently known forward solutions. Then, the subproblem attempts to generate a new forward solution that is better than x^d for the current d vector. If no such forward solution exists, the current d vector is optimal. Otherwise, the improving solution is added to the known forward solution set and the master is re-solved in the next iteration.

Formally, based on Wang [26], we are given a forward mixed-integer program $\langle c, X \rangle$ where $X = \{x \in \mathbb{R}_+ : Ax \leq b, x_I \in \mathbb{N}_0\}$ with $A \in \mathbb{R}^{k \times n}$, $b \in \mathbb{R}^k$, and $I \subseteq \{1, \dots, n\}$, a known solution x^d , and feasible set \mathcal{C} for parameter values. The L_1 norm objective of the inverse problem is linearized using $g, h \in \mathbb{R}_+^n$, such that $c - d = g - h$. Intuitively, the magnitude of the change to the parameter c_i is represented by g_i if it is negative and h_i if it is positive. Any d for which the forward problem $\langle d, X \rangle$ is unbounded can be avoided by adding the constraint $A^T y \geq d$, $y \in \mathbb{R}^k$, ensuring that d results in a feasible dual. Let \mathcal{S}^0 be the, initially empty, set of known feasible solutions to the forward problem. The master problem \mathcal{MP} is:

$$\min_{y, g, h} g + h \quad (5)$$

$$\text{s.t. } A^T y \geq c - g + h \quad (6)$$

$$(c - g + h)^T x^d \leq (c - g + h)^T x^0 \quad \forall x^0 \in \mathcal{S}^0 \quad (7)$$

$$g, h \in \mathbb{R}_+^n, \quad y \in \mathbb{R}^k, \quad (c - g + h) \in \mathcal{C} \quad (8)$$

Constraint (7) forces the objective value of x^d to be at least as good as any known solution. The optimal solution from the master problem $d^* = (c - g^* + h^*)$ is used to construct the subproblem $\mathcal{SP} \langle d^*, X \rangle$: $\min_x \{d^{*T} x : x \in X\}$. The complete algorithm is:

► **Definition 1** (Algorithm: *InvLP-MIP* [26]).

1. Initialize $\mathcal{S}^0 = \emptyset$.
2. Solve \mathcal{MP} to obtain $d^* = (c - g^* + h^*)$.
3. Solve $\mathcal{SP} \langle d^*, X \rangle$ to get optimal solution x^0 . If $d^{*T} x^d \leq d^{*T} x^0$, stop: d^* is optimal for the inverse problem. Otherwise, update $\mathcal{S}^0 = \mathcal{S}^0 \cup \{x^0\}$ and return to step 2.

3 Problem Formulation

We now formulate a class of counterfactual explanation problems that can be solved with the help of inverse optimization. In particular, we explore the case when the user's contrastive question specifies a set of assignments to a subset of decision variables.

3.1 Example: Explaining a Delivery Schedule

Recall our delivery example where clients place a set of orders, specifying a priority level for each order. Expressed as an optimization problem, let the decision variables $x \in X \subseteq \mathbb{N}_0^n$ represent the delivery dates for each order, and the objective coefficients $c \in \mathcal{C} \subseteq \mathbb{N}_0^n$ represent order priorities. Let the scheduling problem $\langle c, X \rangle$ be to minimize the priority weighted delivery dates, that is, to find $\min_{x \in X} c^T x$.

Upon seeing the initial optimal schedule x^* , one of the clients asks “Why was order A not scheduled to arrive in two days, and B in five days?” As we did in the NCE, we use this question to formulate a set of foil constraints, in this case giving $\psi = \{x : x_A = 2, x_B = 5\}$. Observe that these foil constraints take the form of assignments to a subset of variables.

In an NCE, the explanation takes the form of changes to objective parameters c ; in this example, the order priorities. However, an explanation that could include changes to *any* order priority c_i , $i \in \{1, \dots, n\}$, might not be very useful to our client. While our client is able to increase the priorities of their own orders by paying more, they have no control or knowledge of orders from other clients. Furthermore, it may be important to protect the privacy of other clients. Finally, our client may only want an explanation for a *subset* of their own orders. For these reasons, we assume that our client is primarily interested in an explanation involving changes *only* to the order priorities associated with the deliveries in their question, c_A and c_B . Such an explanation might look like: “If you increase the delivery priorities of part A from a Level 1 to a Level 3 (for \$50 extra), and part B from a Level 1 to a Level 4 (\$30 extra), the parts will be delivered on the dates you specified.”

3.2 The Partial Assignment NCE

We now formulate a specific NCE problem to model cases such as our delivery example. We begin with a contrastive question from a user who is interested in a subset of m variables x_i , $i \in \mathcal{M} \subseteq \{1, \dots, n\}$, $m = |\mathcal{M}|$, and desires to know why they were not assigned to specific values, x_i^p , $i \in \mathcal{M}$. We use this question to formulate the partial assignment foil constraints, giving $\psi = \{x : x_i = x_i^p \forall i \in \mathcal{M}\}$.

The user desires an explanation in terms of *only* changes to the parameters c_i , $i \in \mathcal{M}$ associated with the variables in the subset \mathcal{M} . The motivation for this is similar to the delivery example: explanations containing parameters associated with other users may not be useful or secure, and, additionally, this restriction allows the questioner to isolate a specific subset of variables. In an NCE, the feasible values for the modified parameters d are defined by the feasible set \mathcal{C} , so we can require any parameters not in \mathcal{M} to retain their initial values by setting $\mathcal{C} = \{d : d_j = c_j \forall j \in \mathcal{M}^C\}$, where $\mathcal{M}^C = \{1, \dots, n\} \setminus \mathcal{M}$.

► **Definition 2** (Partial Assignment Nearest Counterfactual Explanation (PA-NCE)). *The Partial Assignment NCE is an NCE $\langle f, c, \mathcal{C}, X, \|\cdot\|, \psi \rangle$ in which $\psi = \{x : x_i = x_i^p \forall i \in \mathcal{M}\}$ where $\mathcal{M} \subseteq \{1, \dots, n\}$, $x^p \in \mathbb{R}^m$, $m = |\mathcal{M}|$, $\mathcal{C} = \{d : d_i = c_i \forall i \in \mathcal{M}^C\}$, and $\mathcal{M}^C = \{1, \dots, n\} \setminus \mathcal{M}$.*

4 Theoretical Results

Both the NCE and inverse optimization aim to find a minimally perturbed $d \in \mathcal{C}$; the former such that a set of solutions X_ψ contains an optimal solution and the latter such that a particular solution x^d is optimal. For a PA-NCE with a linear objective, we show that there exists an $x^\psi \in X_\psi$ which is optimal for *any* feasible $d \in \mathcal{C}$. In particular, we prove that such an x^ψ is given by an optimal solution to the initial forward problem plus the foil constraints, $\langle c, X_\psi \rangle$. This result implies that we can solve a PA-NCE in two steps: first, solving $\langle c, X_\psi \rangle$ to find x^ψ , and then solving the analogous inverse optimization problem with $x^d = x^\psi$.

► **Theorem 3.** *The Partial Assignment NCE $\langle c, \mathcal{C}, c^T x, X, \|\cdot\|_1, \psi \rangle$ is equivalent to the Inverse Optimization Problem $\langle c, \mathcal{C}, c^T x, x^d, X, \|\cdot\|_1 \rangle$ with $x^d \in \arg \min\{c^T x : x \in X_\psi\}$.*

Proof. Observe that the NCE and Inverse Optimization problems differ only by the left-hand sides of Constraints (2) and (4). To show the two problems are equivalent, it is sufficient to show that

$$f(d, x^d) = d^T x^d = \min_x \{d^T x : x \in X_\psi\}, \quad \forall d \in \mathcal{C}. \quad (9)$$

From the optimality of x^d to the problem $\langle c, X_\psi \rangle$ and by separating the objective into the contributions from the variables $i \in \mathcal{M}$ and the variables $j \in \mathcal{M}^C$,

$$c^T x^d = \sum_{i \in \mathcal{M}} c_i x_i^d + \sum_{j \in \mathcal{M}^C} c_j x_j^d \leq \sum_{i \in \mathcal{M}} c_i x_i + \sum_{j \in \mathcal{M}^C} c_j x_j, \quad \forall x \in X_\psi. \quad (10)$$

The form of the foil constraint set ψ requires that $x_i^p = x_i^d = x_i$ for all $i \in \mathcal{M}$, so

$$\sum_{i \in \mathcal{M}^C} c_j x_j^d \leq \sum_{i \in \mathcal{M}^C} c_j x_j, \quad \forall x \in X_\psi. \quad (11)$$

Due to the constraints in \mathcal{C} from Definition 2, $d_j = c_j$ for $j \in \mathcal{M}^C$, so the above inequality is valid for all values of $d \in \mathcal{C}$. Further, because the values of x_i , $i \in \mathcal{M}$, are identical in all foils, the contributions from the components in \mathcal{M} are also equivalent given a value of d . Adding this contribution to both sides, we get

$$\sum_{i \in \mathcal{M}} d_i x_i^d + \sum_{i \in \mathcal{M}^C} c_j x_j^d \leq \sum_{i \in \mathcal{M}} d_i x_i + \sum_{i \in \mathcal{M}^C} c_j x_j, \quad \forall x \in X_\psi, \quad \forall d \in \mathcal{C}. \quad (12)$$

Thus $d^T x^d \leq d^T x$ for all $x \in X_\psi$ and all $d \in \mathcal{C}$, satisfying (9) and completing the proof. ◀

All our results continue to hold if the user specifies a full assignment of variables $x^p \in \mathbb{R}^n$ instead of a partial one, so that $\mathcal{M}^C = \emptyset$. In this case, the foil set is a singleton, $X_\psi = \{x^p\}$, so we can skip the first step of finding the optimal foil and proceed directly to the inverse problem with $x^d = x^p$.

5 Inverse Constraint Programming

In order to generate counterfactual explanations for constraint programs, we are interested in solving the PA-NCE for problems in which the forward problem is a constraint program. Given the connection shown above between the PA-NCE and inverse optimization, we can solve the PA-NCE by solving the forward problem using CP to find an optimal foil and then by formulating the inverse optimization problem as a constraint program. For the latter, we adopt the *InvLP-MIP* [26] approach, generalizing it to CP.

Due to the discrete nature of CP, we are primarily interested in problems where the objective coefficients are integral, $c, d \in \mathbb{N}_0^n$, which are also more difficult for MIP based inverse optimization than problems with continuous cost coefficients. Let $\mathcal{MP}_{d \in \mathbb{N}_0^n}$ be an \mathcal{MP} with the constraint $g, h \in \mathbb{R}_+^n$ in (8) replaced with $g, h \in \mathbb{N}_0^n$. Such a master problem can no longer use LP, but can be formulated and solved using MIP. We call the variation of *InvLP-MIP* which uses MIP for the master problem *InvMIP-MIP*.

We can also formulate both the master and subproblems with CP. Let an $\mathcal{MP}_{d \in \mathbb{N}_0^n}$ model defined using CP be called \mathcal{MP}_{CP} and an \mathcal{SP} model $\langle d^*, X \rangle$ defined using CP be called \mathcal{SP}_{CP} . Examples for specific problems are provided in Section 6. We call *InvCP-CP* the algorithm which solves these models using CP and follows the cutting plane approach of *InvLP-MIP*. To our knowledge, this is the first use of CP for inverse optimization.

► **Definition 4** (*InvCP-CP*). Follow steps 1-3 in *InvLP-MIP*, using CP to solve \mathcal{MP}_{CP} and \mathcal{SP}_{CP} , instead of using LP and MIP to solve \mathcal{MP} and \mathcal{SP} , respectively.

■ **Algorithm 1** *InvMIP-MIP*(ESC).

```

1  Inputs:  $c, \mathcal{C}, \gamma, x^d, X$ 
2  Step 1: Initialize  $S^0 \leftarrow \emptyset$ 
3  Step 2: Solve  $\mathcal{MP}_{d \in \mathbb{N}_0^n}$  to get optimal solution  $d^*$ 
4  Step 3: while TRUE:
5      get next feasible solution  $x^i$  to  $\mathcal{SP} \langle d^*, X \rangle$ 
6      if  $d^{*T} x^i \leq d^{*T} x^d$ :
7          if  $x^i$  optimal:
8              if  $d^{*T} x^i == d^{*T} x^d$ :
9                  Stop.  $d^*$  is optimal to the inverse problem.
10             else:
11                 Update  $S^0 \leftarrow S^0 \cup \{x^i\}$ 
12                 go to step 2
13         else:
14             if  $d^{*T} x^i < d^{*T} x^d$ :
15                 if cumulative time spent in  $\mathcal{SP} \geq \gamma$ :
16                     Update  $S^0 \leftarrow S^0 \cup \{x^i\}$ 
17                     go to step 2

```

Duality. A general consideration of duality and unbounded objectives is beyond our scope, however most constraint programs involve finite domains and therefore have bounded objectives. In such cases, the dual constraints (6) in the master problem are unnecessary. If unbounded objectives could exist, it may not always be possible to formulate the dual constraints using CP; we have not yet developed a way to deal with this case. We show in Section 6 that the problems in our experiments are guaranteed to have bounded objectives.

5.1 Hybrid Approaches

In addition to a pure inverse CP algorithm, we also define hybrid inverse algorithms which use both MIP and CP. Specifically, we define *InvMIP-CP* as the algorithm that solves the master problem $\mathcal{MP}_{d \in \mathbb{N}_0^n}$ with MIP, and the subproblem $\mathcal{SP}_{\mathcal{CP}}$ with CP. Similarly, we define *InvCP-MIP* as the algorithm that solves the master problem $\mathcal{MP}_{\mathcal{CP}}$ with CP, and the subproblem \mathcal{SP} with MIP.

5.2 Early Stopping Criteria

In each of the above inverse algorithms, the subproblem is solved to optimality at every iteration and its optimal solution x^0 is added to the master. However, if a feasible, but not necessarily optimal solution x^f has been found which gives a better objective value than the foil, $d^{*T} x^f < d^{*T} x^d$, then a valid cut can be generated by adding x^f to S^0 . While a better forward solution may yield a stronger cut in the master, we may wish to balance the strength and computational expense of a new cut by implementing an early stopping criteria: if a such a solution x^f is found in a given iteration, we can stop solving the \mathcal{SP} after γ seconds.

We define our early stopping criteria (ESC) algorithm in Algorithm 1 using *InvMIP-MIP* as a base. It can be applied to each of the inverse algorithms discussed above. In line (5) the solver returns a feasible solution which is not worse than the previous one. The time in the \mathcal{SP} (line 15) refers to the time since the most recent master solution was found.

6 Models

We test our counterfactual explanation approach for two forward problems: the 0-1 knapsack (KP) and single machine scheduling with release dates, $1|r_j|\sum w_j C_j$. The KP was selected because it is NP-complete [20], has a simple structure, and is easy to understand. While it is of practical importance, including as part of many more complex problems, its simplicity (yet NP-completeness) allowed us to develop our methods before moving to more complex hard combinatorial problems. The scheduling problem was selected because CP often performs well in scheduling, matching a potential use case for CP based explanation techniques (i.e. explainable scheduling). It is also a relatively simple (though strongly NP-Hard [17]) scheduling problem to test our methods on. Both problems have finite domains and are therefore guaranteed to have a bounded objective.

6.1 0-1 Knapsack Problem

In the 0-1 KP, we are given a set of $n \in \mathbb{N}$ items, a profit vector $c \in \mathbb{N}_0^n$, a weight vector $w \in \mathbb{N}_0^n$, and a knapsack capacity $W \in \mathbb{N}_0$. The objective of the 0-1 KP is to maximize the sum of the profits of the items that are included in the knapsack, without having the sum of the weights of those items exceed W .

CP Model. The CP model uses a packing global constraint, specifically *binPackingCapa* [12]. The first argument of this constraint is a set of bins, with each bin $\langle l, W_l \rangle$ associated with an index $l \in \mathbb{N}_0$ and a capacity $W_l \in \mathbb{N}$. The second argument is a set of items, with each item $\langle x_i, w_i \rangle$ corresponding to decision variable $x_i \in \mathbb{N}_0$ identifying which container the item is placed in and an item weight $w_i \in \mathbb{N}$. The constraint ensures that all items are placed in a container such that the sum of item weights in any container does not exceed its capacity. The CP model for 0-1 KP is

$$\max c^T x \quad (13)$$

$$\text{s.t. } \text{binPackingCapa}(\{\langle 0, \infty \rangle, \langle 1, W \rangle\}, \{\langle x_i, w_i \rangle | i \in \{1, \dots, n\}\}) \quad (14)$$

$$x \in \{0, 1\}^n. \quad (15)$$

The choice of whether to place an item in container 1 or container 0 is equivalent to the decision of including or excluding that item in the knapsack, respectively.

MIP Model. Let $x \in \{0, 1\}^n$ be a decision vector where $x_i = 1$ if an item is included in the knapsack and 0 otherwise. The MIP model is

$$\min_x \{c^T x : w^T x \leq W, x \in \{0, 1\}^n\}. \quad (16)$$

6.2 Single Machine Scheduling with Release Dates, $1|r_j|\sum w_j C_j$

In the $1|r_j|\sum w_j C_j$ problem, we are given $n \in \mathbb{N}$ jobs, with each job $i \in \{1, \dots, n\}$ having a processing time $p_i \in \mathbb{N}$, a weight¹ $c_i \in \mathbb{N}$, and a release date $r_i \in \mathbb{N}$. The objective is to minimize the weighted sum of completion times of all jobs given that no two jobs can be processed at the same time, no jobs can start before their release dates, and no jobs can be interrupted (no preemption).

¹ This problem is typically defined with w representing the job weights. To be consistent with our notation, we replace w with c , though we continue to refer to the problem by its typical name, $1|r_j|\sum w_j C_j$.

CP Model. We represent the jobs with a set of interval variables $\{I_i\} \forall i \in \{1, \dots, n\}$, defined with the notation $intervalVar(p_i, [s_i, e_i])$, where the possible values of I_i are the intervals $\{[s_i, e_i] : s_i, e_i \in \mathbb{N}_0, s_i + p_i = e_i\}$. The model is

$$\min \sum_{i=1}^n c_i e_i \quad (17)$$

$$\text{s.t. } NoOverlap(\{I_1, \dots, I_n\}) \quad \forall i \in \{1, \dots, n\} \quad (18)$$

$$s_i \geq r_i \quad \forall i \in \{1, \dots, n\} \quad (19)$$

$$I_i = intervalVar(p_i, [s_i, e_i]) \quad \forall i \in \{1, \dots, n\}. \quad (20)$$

Constraint (18) is the *NoOverlap* global constraint that forces jobs to be processed one at a time. Constraints (19) ensure that jobs do not start before they are released.

Time-Indexed MIP Model. As several MIP formulations exist for $1|r_j|\sum w_j C_j$, we use a time-indexed formulation due to its strong performance over a variety of instances [15]. Let $x_{i,t} \in \{0, 1\}$ be a binary decision variable which is 1 if job i is scheduled to start at time t , and 0 otherwise. With T as the time horizon, an upper bound on latest completion time of any job, the model is

$$\min \sum_{i=1}^n \sum_{t=0}^{T-p_i} c_i(t + p_i)x_{i,t} \quad (21)$$

$$\text{s.t. } \sum_{t=0}^{T-p_i} x_{i,t} = 1 \quad \forall i \in \{1, \dots, n\} \quad (22)$$

$$\sum_{i=1}^n \sum_{s=\max(0, t-p_i+1)}^t x_{i,s} \leq 1 \quad \forall t = 0, 1, \dots, T-1 \quad (23)$$

$$\sum_{t=0}^{r_i-1} x_{i,t} = 0 \quad \forall i \in \{1, \dots, n\} \quad (24)$$

$$x_{i,t} \in \{0, 1\} \quad \forall i \in \{1, \dots, n\}, \forall t \in \{1, \dots, T-1\}. \quad (25)$$

Constraints (22) force each job to start exactly once. Constraints (23) ensure no two jobs are processed at the same time. Finally, constraints (24) enforce the release dates.

7 Experimental Setup

The goal of our experiments is to test the generation of counterfactual explanations for the PA-NCE. We do this by solving an initial forward problem, generating a user query, and solving the resulting PA-NCE, focusing on the latter.

7.1 Problem Instance Generation

To generate PA-NCE instances, we create and solve a forward problem instance and then generate a set of foil constraints that form the user query.

0-1 KP Instances. Each forward problem consists of $n \in \{20, 30, 40\}$ items. For all instances, profit c_i and weight w_i are both drawn independently from the random uniform distribution $[1, R]$ with $R = 1000$. The knapsack capacity is $W = \max\{\lfloor P \sum_{i=1}^n w_i \rfloor, R\}$, with $P = 0.5$. Each instance was solved to produce an optimal solution, x^* .

To generate the user query, $m \in \{5, 10, 15\}$ items were randomly selected from $\{1, \dots, n\}$ to create the set \mathcal{M} . Each user assignment $x_i^p, i \in \mathcal{M}$ was set to the opposite value of x_i^* : 0 if $x_i^* = 1$ and 1 if $x_i^* = 0$. We obtain X_ψ by adding the corresponding set of assignment constraints $x_i = x_i^p, \forall i \in \mathcal{M}$, to the original constraint set X for MIP and CP, respectively. Recall that in the PA-NCE, any restrictions on the feasible values of the modified parameters d are expressed through the feasible set \mathcal{C} . In addition to specifying that only the parameters $d_i, i \in \mathcal{M}$, can change, we also add the constraint $d \in \mathbb{N}_0^n$. We generated 20 problem instances for each combination of (n, m) .

This instance generation procedure may result in an infeasible query if \mathcal{M} forces the knapsack to contain items that exceed its capacity. In this case, a new random set \mathcal{M} was generated until a non-empty foil set X_ψ was found.

$1|r_j| \sum w_j C_j$ Instances. Forward instances of size $n = \{5, 10, 15\}$ were generated with the random uniform distributions $p_i \in [10, 100]$, $c_i \in [1, 10]$, and $r_i \in [0, \lfloor qP \rfloor]$, where $q = 0.4$ and $P = \sum_{i=1}^n p_i$. The time horizon T was calculated as $T = \lfloor qP \rfloor + P$. We generate 20 instances for each value of (n, m) , with tuple values given in Section 8.

The generation of a feasible set of foil constraints to assign the start times of a subset of jobs is non-trivial for this problem. In an optimal solution for a given complete sequence of jobs, all jobs are left-shifted subject to the release time constraints. Therefore, an arbitrarily chosen start time for a job will not form part of an optimal solution unless it happens to be equal to the job's release date or to the completion time of another job in some optimal sequence. Following the simple query generation approach used with the knapsack problem is therefore likely to result in many infeasible explanation problems.

Therefore, to generate instances more likely to have feasible explanations, we follow a different approach, although the infeasibility of some PA-NCEs remains an issue (see Section 9). We create a random permutation $(a_i)_{i \in \mathcal{M}}$ of m jobs in a randomly chosen subset \mathcal{M} . We then solve the original forward problem to optimality, constraining the jobs in \mathcal{M} to follow the selected permutation. Finally, we select the start times in the user query to be the start times of the jobs in \mathcal{M} from this solution.

Specifically, the constraints added to the forward problem were, for CP,

$$\text{endBeforeStart}(I_j, I_i) \quad \forall i, j \in \mathcal{M}, a_i > a_j, \quad (26)$$

which forces the end e_j of interval variable I_j to be less than or equal to the start s_i of interval variable I_i , $e_j \leq s_i$. For MIP, the constraints added were

$$\sum_{t=0}^{T-p_j} tx_{j,t} < \sum_{t=0}^{T-p_i} tx_{i,t} \quad \forall i, j \in \mathcal{M}, a_i > a_j. \quad (27)$$

Finally, we add $d \in \mathbb{N}^n = \{1, 2, \dots\}^n$ as one of the constraints that define \mathcal{C} in the PA-NCE, restricting the modified weights to be positive integers.

7.2 PA-NCE Solution

Having generated our PA-NCE instances, $\langle c^T x, c, \mathcal{C}, X, \|\cdot\|_1, \psi \rangle$, we solve them using our two-step approach (Section 4): first finding the optimal foil x^ψ by solving $\langle c, X_\psi \rangle$, and then solving the corresponding inverse problem using x^ψ as the known solution. We test two groups of algorithms, based on whether the forward problems were solved with CP or with MIP. Notice that there are multiple forward problems involved in solving each PA-NCE instance. First, there is the optimal foil problem $\langle c, X_\psi \rangle$. Then, each iteration of the inverse

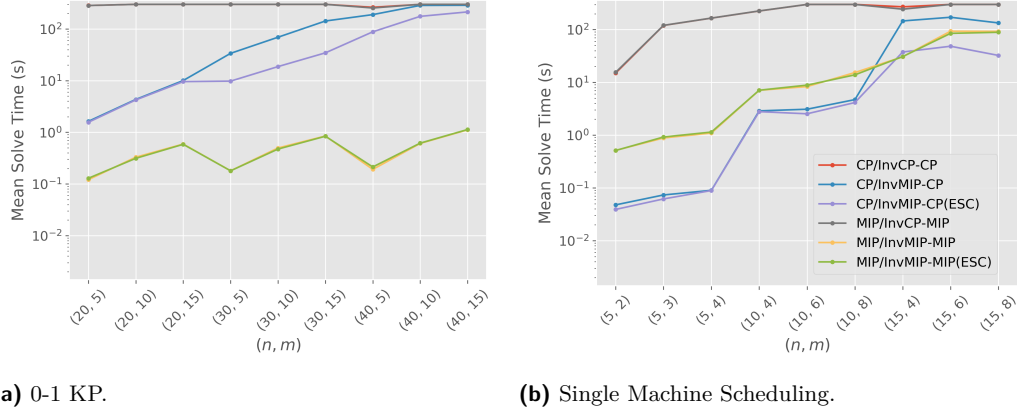


Figure 1 PA-NCE Mean Solve Times.

algorithm uses a new d to solve a subproblem $\langle d, X \rangle$. We refer to the algorithms that use CP for these forward problems as forward CP based, and the algorithms that use MIP for these forward problems as forward MIP based.

For each of these two algorithm groups, we tested three inverse algorithms. The first used CP in the master problem, while the second used MIP. The third inverse algorithm applied the ESC to the subproblem when MIP was used for the master. We name the complete two-step algorithms by first specifying the technique used to solve the optimal foil problem (CP or MIP), and then specifying the inverse algorithm. For example, we call *CP/InvMIP-CP* the algorithm that uses CP to find the optimal foil (step one) and *InvMIP-CP* to solve the inverse problem (step two). We tested six such two-step algorithms in total.

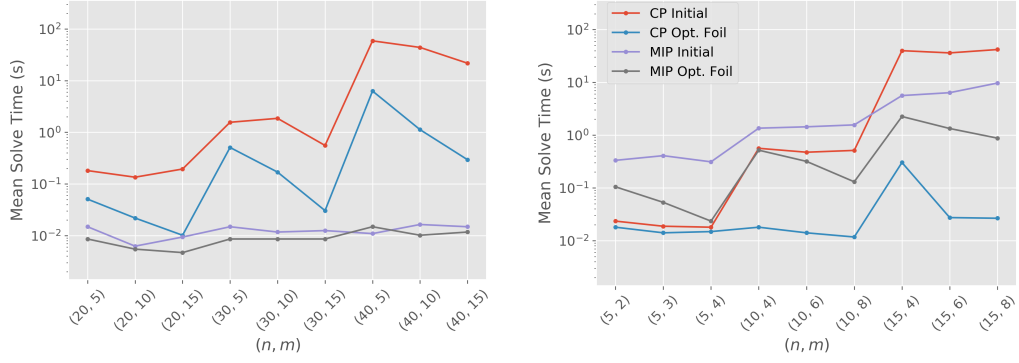
We also track the performance of CP and MIP for the initial forward problem $\langle c, X \rangle$. While this is part of instance generation and not explanation generation, it is a useful proxy for the performance of the solvers in the subproblem in the inverse algorithms. Additionally, for all inverse algorithms, we take advantage of having an initial forward solution x^* to initialize the set of known solutions as $S^0 = \{x^*\}$.

7.3 Computational Details

All two-stage algorithms were run for a *global* time limit t_{max} of 300 seconds (for both stages together). If a PA-NCE instance was not solved within the global time limit, then t_{max} was recorded as the solve time. For all inverse algorithms that used the ESC, the subproblem time limit γ was set to 1 second. The MIP solver used was ILOG CPLEX V12.10 and the CP solver was ILOG CPOptimizer V12.10. Experiments were run on a single core of a 2.5 GHz Intel Core i7-6500U CPU and all reported times are CPU times.

8 Experimental Results

Overall Performance. Figure 1 shows the solution times for the two-stage algorithms for PA-NCEs and Figure 2 the solution times for the optimal foil problem and the initial forward problem – note the log-scale on the y-axes. For the 0-1 KP, the *MIP/InvMIP-MIP* and *MIP/InvMIP-MIP(ESC)* algorithms are by far the most effective, likely due to the strong MIP performance for the forward problem. As mentioned, initial forward problem



(a) 0-1 KP.

(b) Single Machine Scheduling.

■ **Figure 2** Mean Solve Times for Initial Forward and Optimal Foil Problems.

performance, shown in Figure 2a, is a good proxy for subproblem performance. This result is not surprising given that MIP solvers are typically very good at working with knapsack constraints.

For single-machine scheduling, the best performing algorithm overall is *CP/InvMIP-CP(ESC)*, partly due to the superiority of CP over MIP on the forward problem for instances with $n \leq 10$ (Figure 2b). However, for instances with $n = 15$, the success of *CP/InvMIP-CP(ESC)* is due to the ESC modification, as demonstrated by its improvement over the unmodified *CP/InvMIP-CP* algorithm in Figure 1b.

Early Stopping Criteria. For both problems, the early stopping criteria was clearly beneficial for the forward CP based algorithm, *InvMIP-CP*, when n was sufficiently large. It had no effect for smaller problems, because when a subproblem is solved to optimality within γ seconds, the ESC is never met. Interestingly, the ESC did *not* produce improvements the forward MIP based algorithm, *InvMIP-MIP*. Recall that the motivation behind the ESC was to avoid solving the *SP* to optimality at each iteration. We speculate that the ESC improves *InvMIP-CP* because it prevents CP from spending an excessive amount of time *proving* that the subproblem solution is optimal. In contrast, we speculate that much less time is spent proving subproblem optimality in *InvMIP-MIP*.

CP for Master Problem. When CP is used to solve the master problem, performance is very poor: *CP/InvCP-CP* and *MIP/InvCP-MIP* both reached the time limit on most instances (Figure 2a and 2b). The simple linear structure of the master problem lends itself particularly well to MIP solving. However, we anticipate that CP may be useful in the master problem for more complex explanation problems, for example to express more complicated constraints on \mathcal{C} , providing more control over how the objective parameters are allowed to change.

Instance Breakdown. Tables 1 and 2 provide more detailed data on the performance of the two-stage algorithms in terms of the number of instances solved optimally, proved infeasible, and timed-out. There were a large number of infeasible PA-NCEs for larger scheduling instances even with our approach for generating feasible foils. Even though these instances have feasible foils by construction, there is no guarantee that a cost vector exists that makes

■ **Table 1** Number of 0-1 KP PA-NCE Solutions Optimal (O) and Timeout (T). Algorithm names are split into the first (optimal foil) and second (inverse) stage components.

Foil Alg		CP						MIP					
Inv Alg		CP-CP		MIP-CP		MIP-CP (ESC)		CP-MIP		MIP-MIP		MIP-MIP (ESC)	
n	m	O	T	O	T	O	T	O	T	O	T	O	T
20	5	1	19	20	0	20	0	1	19	20	0	20	0
	10	0	20	20	0	20	0	0	20	20	0	20	0
	15	0	20	20	0	20	0	0	20	20	0	20	0
30	5	0	20	20	0	20	0	0	20	20	0	20	0
	10	0	20	19	1	20	0	0	20	20	0	20	0
	15	0	20	14	6	20	0	0	20	20	0	20	0
40	5	3	17	11	9	18	2	3	17	20	0	20	0
	10	0	20	1	19	15	5	0	20	20	0	20	0
	15	0	20	2	18	10	10	0	20	20	0	20	0

■ **Table 2** Number of $1|r_j|\sum w_j C_j$ PA-NCE Solutions: Optimal (O), Infeasible (I), Timeout (T). Algorithm names are split into the first (optimal foil) and second (inverse) stage components.

Foil Alg		CP									MIP								
Inv Alg		CP-CP			MIP-CP			MIP-CP (ESC)			CP-MIP			MIP-MIP			MIP-MIP (ESC)		
n	m	O	I	T	O	I	T	O	I	T	O	I	T	O	I	T	O	I	T
5	2	19	0	1	19	1	0	19	1	0	19	0	1	19	1	0	19	1	0
	3	12	0	8	18	2	0	18	2	0	12	0	8	18	2	0	18	2	0
	4	9	0	11	15	5	0	15	5	0	9	0	11	15	5	0	15	5	0
10	4	5	0	15	12	8	0	12	8	0	5	0	15	12	8	0	12	8	0
	6	0	0	20	5	15	0	5	15	0	0	0	20	5	15	0	5	15	0
	8	0	0	20	4	16	0	4	16	0	0	0	20	4	16	0	4	16	0
15	4	3	0	17	6	8	6	7	12	1	4	0	16	8	12	0	8	12	0
	6	0	0	20	0	11	9	3	15	2	0	0	20	4	15	1	5	14	1
	8	0	0	20	1	12	7	1	18	1	0	0	20	2	16	2	2	17	1

the foil optimal. This issue is discussed further in Section 9. In contrast, no instances were infeasible for the inverse knapsack problems as any inverse 0-1 KP has a feasible solution in which any items which are not included in the knapsack by x^d have their profits set to 0.

Optimal Foil Problem. Results for finding the optimal foil are included in Figures 2a and 2b. Similar to most initial problems, MIP performed better for KP while CP performed better for scheduling.

9 Discussion and Limitations

This paper develops methods to find nearest counterfactual explanations for a class of optimization problems and user queries and introduces inverse constraint programming as part of the solution methodology. Here we address the limitations of our approaches.

Counterfactual explanations are independent of the algorithmic decision making process as they identify *changes* in the problem that would result in the user's specifications being met. However, they require that the changeable parameters are meaningful to the user. If

this is not the case, then the user requires a higher level explanation of why those parameters are used and how their factual values were derived: questions that touch on ethics, fairness, and the broader human system surrounding the optimization problem [16].

Our approach cannot generate nearest counterfactual explanations for foil sets described by more expressive constraints, as defined in the general NCE. The core challenge is that the constraints admit a set of solutions and the NCE seeks a cost vector such that *any one* of those solutions is optimal. As far as we are aware, there exist no inverse optimization approaches to such generalizations. We were able to solve the PA-NCE because Theorem 3 guarantees the existence of an optimal foil and so standard inverse optimization can be used. As the NCE is a bi-level optimization problem, one direction for future work on the more general problem is to investigate techniques from discrete bi-level optimization [22].

Similarly, the NCE definition and our approach are limited to counterfactual values for parameters in the objective function. Clearly, a user may want explanations in terms of changes to constraints. The challenge is that such changes modify the feasible set of the forward problem, complicating the inverse optimization formulation. We are not aware of any general inverse combinatorial optimization approaches that can handle changes to constraint parameters, though formulations have been investigated for specific inverse problems [27]. One direction for work on such problems may be to combine the approach here with existing work in CP on explaining infeasibilities (Section 10). In continuous optimization, inverse methods allowing changes to constraint parameters exist for linear programs [4].

In our formulation, a counterfactual query is an assignment of a subset of variables. However, we required that an explanation consist only of changes to the parameters corresponding to the variables in the user query. While we argued above that such a restriction is often useful due to parameter relevance and privacy, without it Theorem 3 does not hold and our solution approach does not work. The challenge, as above, is that without this restriction, the NCE requires finding a parameter vector such that any one of a set of solutions is optimal.

Finally, as described in Section 7.1 and shown in Table 2, the restrictions of PA-NCE may make it difficult to generate queries with a feasible explanation. From a user’s perspective, just as with forward optimization, a result that says that there is no world in which the user’s decisions are possible is not particularly helpful. As far as we are aware this issue has not been addressed in the broader literature of counterfactual explanations. However, the generalization noted above of allowing constraint coefficients to change may be worth pursuing for this challenge as well.

In spite of these limitations, the work in this paper substantially extends the scope of counterfactual explanations for optimization-based decisions from a counterfactual query on a single variable to queries on some or all variables, albeit with a restricted query form. Furthermore, for the first time we have defined inverse constraint programming and solved such problems through an adaptation and extension of work in inverse integer programming.

10 Related Work

We build directly on our previous work [16], which formulates the NCE and proposes the connection to inverse combinatorial optimization. However, that work does not develop an inverse optimization based solution approach, relying on the restriction that the query must be a linear constraint over a single variable. This restriction allows NCEs with binary decision variables to be solved in closed form given the solution to a modified problem, and with binary search over a series of modified problems when the decision variables are integers.

In ML, there has been a significant amount of research on counterfactual explanations for classifiers [24], with highly influential work by Wachter *et al.* [25]. Although still nascent, the work has already developed more advanced concepts such as diverse counterfactual explanations [21] which could be extended to counterfactual explanations for optimization.

In AI Planning, a counterfactual explanation approach has been adopted in a number of contexts, for instance to enumerate the shared properties of all possible counterfactual plans [7] and to identify the differences between counterfactual plans and factual plans [10]. Further, Brandao and Maggazzini [2] recently used inverse optimization to generate explanations for path planning, for which there exists a polynomial inverse algorithm.

In CP, the majority of work on explanations has focused on explaining infeasibility with work on optimality being sparse [11]. To our knowledge there have been no attempts to explain optimality using counterfactual explanations. For constraint satisfaction problems which are solvable with inference only (no search), Sqalli and Freuder [23] generated explanations by tracing the inference steps, and observed that, for the logic puzzles used in their experiments, these explanations were very similar to those generated by humans. Subsequent research has built on this approach [1] while acknowledging the limitation that, for problems that also require search, tracing solver steps does not currently provide understandable explanations.

Explanation of infeasibility in CP has largely dealt with finding minimal sets of unsatisfiable constraints [14], and a parallel literature exists in mathematical programming [5]. We have previously proposed [16] that such explanations could also be viewed as counterfactuals (i.e., a set of constraints that must change) but this connection has not been developed. Freuder [11] provides a recent discussion and overview of explainability in CP.

11 Conclusion

Counterfactual explanations answer a user query asking why, given a set of facts, an initial decision did not satisfy some desired characteristics. The explanation is a hypothetical set of facts that would have satisfied the user's characteristics. Because they do not require the user to understand the inner workings of increasingly complex and uninterpretable algorithms, counterfactual explanations are drawing considerable research attention in AI. We build on recent work on counterfactual explanations for discrete optimization by introducing multi-variate explanation problems and solving them with the help of inverse optimization.

When a user is interested in an alternative, partial set of variable assignments, we show how to generate an explanation in terms of changes to the objective parameters associated with those variables. A counterfactual explanation can be found by first solving the original problem with the addition of the user's partial assignment constraints, and then solving a corresponding inverse optimization problem. We solve the inverse problem with constraint programming through a modification of an existing MIP cutting plane algorithm to develop both pure and hybrid inverse constraint programming algorithms. In addition, we develop a novel early stopping criteria that significantly improves inverse CP on larger problem instances. Finally, through numerical experiments we demonstrate our solution approaches for the 0-1 knapsack problem and a single machine scheduling problem, and show that a hybrid MIP-CP approach can achieve superior performance compared to a pure MIP approach, particularly when CP is state of the art for the underlying optimization problem.

References

- 1 B. Bogaerts, E. Gamba, J. Claes, and T. Guns. Step-wise explanations of constraint satisfaction problems. In *24th European Conference on Artificial Intelligence (ECAI)*, 2020.
- 2 M. Brandao and D. Magazzini. Explaining plans at scale: scalable path planning explanations in navigation meshes using inverse optimization. In *Workshop on XAI, IJCAI-PRICAI*, 2020.

- 3 J. Burrell. How the machine ‘thinks’: Understanding opacity in machine learning algorithms. *Big Data & Society*, 3(1), 2016.
- 4 T. C. Y. Chan and N. Kaw. Inverse optimization for the recovery of constraint parameters. *European Journal of Operational Research*, 282(2):415–427, 2020.
- 5 J. W. Chinneck. *Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods*. Number 118 in International Series in Operations Research and Management Sciences. Springer, 2008.
- 6 M. Demange and J. Monnot. An introduction to inverse combinatorial problems. *Paradigms of Combinatorial Optimization: Problems and New Approaches*, pages 547–586, 2014.
- 7 R. Eiffer, M. Cashmore, J. Hoffmann, D. Magazzeni, and M. Steinmetz. A New Approach to Plan-Space Explanation: Analyzing Plan-Property Dependencies in Oversubscription Planning. In *AAAI*. AAAI, 2020.
- 8 K. Epstude and N. J. Roese. The functional theory of counterfactual thinking. *Personality and social psychology review*, 12(2):168–192, 2008.
- 9 V. Eubanks. *Automating inequality: How high-tech tools profile, police, and punish the poor*. St. Martin’s Press, 2018.
- 10 M. Fox, D. Long, and D. Magazzeni. Explainable Planning. *CoRR*, abs/1709.10256, 2017. [arXiv:1709.10256](https://arxiv.org/abs/1709.10256).
- 11 E. Freuder. Explaining ourselves: human-aware constraint reasoning. In *AAAI*, 2017.
- 12 Global Constraint Catalogue. <https://sofdem.github.io/gccat/>. Accessed: 2021-08-5.
- 13 C. Heuberger. Inverse combinatorial optimization: A survey on problems, methods, and results. *Journal of combinatorial optimization*, 8(3):329–361, 2004.
- 14 U. Junker. Preferred explanations and relaxations for over-constrained problems. In *AAAI*, 2004.
- 15 A. B. Keha, K. Khowala, and J. W. Fowler. Mixed integer programming formulations for single machine scheduling problems. *Computers & Industrial Engineering*, 56(1):357–367, 2009.
- 16 A. Korikov, A. Shleyfman, and C. Beck. Counterfactual explanations for optimization-based decisions in the context of the GDPR. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, 2021.
- 17 J. K. Lenstra, A. R. Kan, and P. Brucker. Complexity of machine scheduling problems. In *Annals of discrete mathematics*, volume 1, pages 343–362. Elsevier, 1977.
- 18 T. Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence*, 267:1–38, 2019.
- 19 Parliament and Council of the European Union. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation), 2016.
- 20 D. Pisinger, H. Kellerer, and U. Pferschy. Knapsack problems. *Handbook of Combinatorial Optimization*, page 299, 2013.
- 21 C. Russell. Efficient search for diverse coherent explanations. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*, pages 20–28, 2019.
- 22 A. Sinha, P. Malo, and K. Deb. A review on bilevel optimization: from classical to evolutionary approaches and applications. *IEEE Transactions on Evolutionary Computation*, 22(2):276–295, 2017.
- 23 M. H. Sqalli and E. C. Freuder. Inference-based constraint satisfaction supports explanation. In *AAAI/IAAI, Vol. 1*, pages 318–325, 1996.
- 24 S. Verma, J. Dickerson, and K. Hines. Counterfactual explanations for machine learning: A review. *arXiv preprint*, 2020. [arXiv:2010.10596](https://arxiv.org/abs/2010.10596).
- 25 S. Wachter, B. Mittelstadt, and C. Russell. Counterfactual explanations without opening the black box: Automated decisions and the GDPR. *Harv. JL & Tech.*, 31:841, 2017.
- 26 L. Wang. Cutting plane algorithms for the inverse mixed integer linear programming problem. *Operations Research Letters*, 37(2):114–116, 2009.
- 27 C. Yang, J. Zhang, and Z. Ma. Inverse maximum flow and minimum cut problems. *Optimization*, 40(2):147–170, 1997.

Utilizing Constraint Optimization for Industrial Machine Workload Balancing

Benjamin Kovács

Universität Klagenfurt, Austria

Pierre Tassel

Universität Klagenfurt, Austria

Wolfgang Kohlenbrein

Kostwein Holding GmbH, Klagenfurt, Austria

Philipp Schrott-Kostwein

Kostwein Holding GmbH, Klagenfurt, Austria

Martin Gebser

Universität Klagenfurt, Austria

Technische Universität Graz, Austria

Abstract

Efficient production scheduling is an important application area of constraint-based optimization techniques. Problem domains like flow- and job-shop scheduling have been extensive study targets, and solving approaches range from complete and local search to machine learning methods. In this paper, we devise and compare constraint-based optimization techniques for scheduling specialized manufacturing processes in the build-to-print business. The goal is to allocate production equipment such that customer orders are completed in time as good as possible, while respecting machine capacities and minimizing extra shifts required to resolve bottlenecks. To this end, we furnish several approaches for scheduling pending production tasks to one or more workdays for performing them. First, we propose a greedy custom algorithm that allows for quickly screening the effects of altering resource demands and availabilities. Moreover, we take advantage of such greedy solutions to parameterize and warm-start the optimization performed by integer linear programming (ILP) and constraint programming (CP) solvers on corresponding problem formulations. Our empirical evaluation is based on production data by Kostwein Holding GmbH, a worldwide supplier in the build-to-print business, and thus demonstrates the industrial applicability of our scheduling methods. We also present a user-friendly web interface for feeding the underlying solvers with customer order and equipment data, graphically displaying computed schedules, and facilitating the investigation of changed resource demands and availabilities, e.g., due to updating orders or including extra shifts.

2012 ACM Subject Classification Applied computing → Command and control

Keywords and phrases application, production planning, production scheduling, linear programming, constraint programming, greedy algorithm, benchmarking

Digital Object Identifier 10.4230/LIPIcs.CP.2021.36

Supplementary Material *Dataset:* <https://github.com/prosysscience/CPWorkloadBalancing>
archived at `swh:1:dir:eab82a06d181ac3ff1b5d6c5e11f9d8d9700b0bd`

Funding This work was partially funded by KWF project 28472, cms electronics GmbH, FunderMax GmbH, Hirsch Armbänder GmbH, incubed IT GmbH, Infineon Technologies Austria AG, Isovolta AG, Kostwein Holding GmbH, and Privatstiftung Kärntner Sparkasse.

1 Introduction

High customization and flexibility of modern production processes increase the need for efficient and performant scheduling methods in order to optimize the utilization of required equipment and resources [13]. Well-studied problem domains like flow- and job-shop scheduling [7, 19, 24], or even tardiness minimization for jobs running on a single machine [11],



© Benjamin Kovács, Pierre Tassel, Wolfgang Kohlenbrein, Philipp Schrott-Kostwein, and Martin Gebser;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 36; pp. 36:1–36:17



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

turn out to be NP-hard [23]. This elevated complexity makes constraint-based optimization techniques based on integer linear programming (ILP) [4], constraint programming (CP) [22], answer set programming (ASP) [17], or Boolean satisfiability (SAT) [5] attractive means for tackling scheduling tasks, while problem size and specifics of industrial domains necessitate customizations when adapting such solving methods from lab to real-world environments [14].

Industrial domains impose particular challenges due to irregularities and exceptions, such as maintenance downtimes or extra shifts, uncertainties and variances, e.g., due to machine breakdowns or manual handling times, as well as dynamics and policies, as evolving from production order updates or customer service duties. While ILP and CP solvers proved to be highly effective on benchmark scheduling problems [3, 9, 18, 21], which refer to abstract domain models that do not incorporate the specifics encountered in industrial practice, local dispatching [16], machine learning [25], and metaheuristic methods [27] are devised for reactive decision making but cannot guarantee (near-)optimal quality of online schedules.

In this paper, we devise and empirically study optimization techniques for scheduling specialized manufacturing processes of Kostwein Holding GmbH in the build-to-print business. Unlike with large-scale assembly line production, each product is made and customized to order, and lot size one is a frequent scenario. This goes along with the manual steps of preparing machines and tools for particular production tasks in order to process daily work plans at the discretion of experienced engineers. Hence, scheduling consists of deciding which production tasks shall be performed at each workday such that resource bottlenecks are projectively avoided and customer orders are completed in time as good as possible. Moreover, we have to take dynamic factors like varying processing times and production order updates into account, which can make schedules obsolete and necessitate quick changes.

In order to address these challenges, we furnish several approaches for scheduling pending production tasks to one or more workdays, considering that long processes may exceed daily machine capacities, for performing them. The four main contributions of our work are:

- We provide an abstract formulation of our production scheduling problem from industry and contribute an instance set based on real production data.
- We propose a greedy custom algorithm as well as ILP and CP models, enabling constraint optimization by state-of-the-art ILP and CP solvers like Gurobi¹ and Google OR-tools².
- We empirically investigate our scheduling methods in realistic setups and show that constraint-based optimization is highly beneficial for further improving greedy solutions.
- We present a user-friendly web interface for launching solvers and inspecting their results to facilitate rescheduling, e.g., with updated orders or extra shifts, and decision making.

As a result, we demonstrate the practical applicability of our scheduling methods to instances of industrial size and relevance. In particular, our devised techniques support a timely reaction to deviations, such as process delays and customer order updates, as well as an upfront identification of resource bottlenecks, thus assisting production managers to take appropriate measures like increasing machine capacities by including extra shifts or delegating some of the pending production tasks to external suppliers.

The rest of this paper is organized as follows. Section 2 introduces an abstract formulation of the production scheduling problem at Kostwein Holding GmbH. In Section 3, we present a greedy custom algorithm as well as ILP and CP models for solving the problem. We evaluate the devised techniques on instances extracted from real production data in Section 4. Section 5 describes the web interface used to deploy our scheduling methods in industrial practice. Finally, conclusions and future work are discussed in Section 6.

¹ <https://www.gurobi.com/products/gurobi-optimizer/>

² <https://developers.google.com/optimization/>

2 Problem formulation

The specialized manufacturing processes at Kostwein Holding GmbH go along with manual steps like preparing machines and tools for particular production tasks, where experienced engineers organize the workflows at their operating units. Given such independent and non-prescribed task execution, fine-grained sequencing of tasks competing for a resource, e.g., performed in flow- and job-shop scheduling, is out of scope and scheduling consists of deciding which of the pending production tasks shall be performed at each workday within the available machine capacities. The overall goal is thus to balance the workload of machines over several weeks from a global perspective such that bottlenecks are projectively avoided, customer orders can be completed in time, and as few additional resources as possible have to be utilized otherwise, e.g., by including extra shifts or delegating pending tasks to external suppliers. In the following, we formally specify the application scenario we are dealing with.

We consider a set M of *machines* and a set J of *jobs*, where each job $j \in J$ is a sequence $t_1^j, \dots, t_{n_j}^j$ of *tasks* to be successively performed on *days* denoted by integers $D = \{0, \dots, h\}$. Every job j has an associated *earliest start* day $e_j \in D$ such that $1 \leq e_j$, a *deadline* $d_j \in D$, and a *weight* $w_j \in \mathbb{N}$ used for penalizing tardiness. While day 0 can be viewed as the date of today and is admitted as deadline for jobs that should have been accomplished already, the positive earliest start days express that tasks require some lead time and can only be scheduled from tomorrow on or even later, e.g., in case raw materials need to be supplied first. The machines $m \in M$ are characterized by daily *capacities* $q_{m,k} \in \mathbb{Q}^+ \cup \{0\}$ for $1 \leq k \leq h$, where $q_{m,k} = 0$ means that m is *unavailable* on day k , such as on weekends or maintenance.

Each task $t = t_i^j$ of some job $j \in J$ is further characterized by the following attributes:

- the *machine* $m_t \in M$ to be used for processing t ,
- the *processing time* $p_t \in \mathbb{Q}^+$ required for performing t ,
- the number $g_t \in \mathbb{N}$ of *gap days* that must lie in-between the day of performing $t = t_i^j$ and t_{i-1}^j if $1 < i \leq n_j$, while we take $g_t = 0$ as fixed when $t = t_1^j$ has no preceding task, and
- a *coupling flag* $c_t \in \{0, 1\}$, where $c_t = 1$ indicates that $t = t_i^j$ has to be performed directly after t_{i-1}^j for $1 < i \leq n_j$, while $c_t = 0$ does not impose any such condition and is always the case when $t = t_1^j$.

Note that the machine to process a task is fixed, which reflects that specialized products are customized to order, so that the specification of jobs may involve CAD design as well as CNC programming and task allocation cannot easily be automated or adjusted. The possible gap days between a task and its predecessor are included to leave time for intermediate steps like specialized processes performed by external suppliers or transports between separate manufacturing sites. In general, we assume that operating units organize their daily work independently in order to perform their pending tasks efficiently, which precludes specific assumptions about the sequencing of tasks on a machine and time slots of processing within a day. Hence, a task $t = t_i^j$ must be performed $g_t + 1$ or more days later than its predecessor t_{i-1}^j (if any) and cannot be scheduled to the same day. On the other hand, the coupling flag $c_t = 1$ expresses that there must not be any delay between processing t_{i-1}^j and t_i^j , apart from days $k \in D$ such that $q_{m_t,k} = 0$ signals unavailability of the machine m_t . We use such coupling for modeling long production tasks that exceed the daily capacity of their machine and are thus broken up into parts to be processed directly after each other, which circumvents unintended and inoperative preemptive scheduling of (long) tasks.

A *schedule* is an assignment $a : T \rightarrow D$ of tasks $T = \{t_i^j \mid j \in J, 1 \leq i \leq n_j\}$ to days such that, for every job j , $1 < i \leq n_j$, and $t = t_i^j$, we have that

- $e_j \leq a(t_1^j)$,
- $a(t_{i-1}^j) + g_t < a(t_i^j)$, and
- if $c_t = 1$, $q_{m_t,k} = 0$ for each day $k \in D$ with $a(t_{i-1}^j) < k < a(t_i^j)$,

while the capacities $q_{m,k}$ of machines $m \in M$ for days $1 \leq k \leq h$ have to be respected, i.e., $\sum_{t \in T, m_t=m, a(t)=k} p_t \leq q_{m,k}$ must hold. That is, machine capacities, earliest start days of jobs, and requirements about the days between successive tasks impose hard constraints on feasible schedules. Clearly, the scheduling *horizon* h , i.e., the number of days to which tasks can be scheduled, must be large enough to allow for allocating all tasks within the available machine capacities, and in Section 3.1 we present a greedy algorithm to determine a sufficient horizon. Also note that the rational numbers for processing times and machine capacities are merely considered for a convenient representation of timeframes, e.g., in terms of fractions of hours or minutes, while they can always be scaled to integers without loss of precision, so that off-the-shelf ILP and CP solvers can be applied to the corresponding models described in Section 3.2 and Section 3.3.

The deadlines d_j and weights w_j for jobs $j \in J$ are used to assess the *quality* of schedules by a weighted sum $\sum_{j \in J, a(t_{n_j}^j) > d_j} w_j \cdot (\omega \cdot (a(t_{n_j}^j) - d_j) + \omega')$ subject to *penalties* $\omega, \omega' \in \mathbb{N}$ per day a job is delayed or per delayed job, respectively. Both penalties are multiplied by job weights to incorporate priorities, the lower the weighted sum the better the quality of a schedule is, and the numbers taken for ω and ω' allow for balancing specific objectives obtained when either penalty is set to zero: if $\omega' = 0$, the total weighted tardiness of a schedule is to be minimized, or the weights of delayed jobs should be minimal in case $\omega = 0$.

Our scheduling problem is a specific version of the Resource-Constrained Project Scheduling Problem (RCPSP) [15], which considers the scheduling of activities under precedence and resource constraints such that particular objectives are optimized. The broad RCPSP framework embraces plenty problem variants with diverse features and solving approaches proposed in the literature. Our application can here be categorized as a single-mode [8], partially renewable [6], cumulative [20] resource allocation task with release dates and deadlines [10] subject to time-based objectives [2]. Machine unavailability days, usually standing for weekends or bank holidays, constitute a specific phenomenon of our scenario leading to variable time periods from start to completion for long production tasks, which we model by splitting tasks stretching over several days up into several coupled parts.

2.1 Example

Table 1 provides the input parameters of an example problem instance with four jobs and three machines. The earliest start days e_j , deadlines d_j , and weights w_j of the jobs $j \in \{1, 2, 3, 4\}$ are listed in Table 1a, showing that the first job cannot be started before day 2 and the others immediately on the first workday. Considering a horizon of five days, the daily machine capacities are given in Table 1b. Note that the positive capacities $q_{m,k}$ for $m \in \{1, 2, 3\}$ are uniform, i.e., $q_{1,k} = 8$, $q_{2,k} = 20$, and $q_{3,k} = 4$, while $q_{m,k} = 0$ signals unavailability of a machine m on day k otherwise. Currently we build on such uniform capacity patterns for production scheduling at Kostwein Holding GmbH, since we statically split long processes into coupled tasks such that the processing time of all but the last part amounts to the uniform capacity of the allocated machine. The tasks t_1^1 and t_2^1 of the first job, listed together with the other jobs' tasks in Table 1c, constitute a corresponding example, where both tasks are to be processed by machine 1, the processing time 8 of t_1^1 matches $q_{1,1} = q_{1,2} = q_{1,4} = q_{1,5} = 8$, and $c_t = 1$ for $t = t_2^1$ indicates that the second part obtained by splitting a long task of 10 time units is coupled. Clearly, a coupled task like $t = t_2^1$

■ **Table 1** Input parameters of an example problem instance.

(a) Earliest start days, deadlines, and weights of jobs.

Job j	Earliest e_j	Deadline d_j	Weight w_j
1	2	4	3
2	1	4	2
3	1	2	2
4	1	1	2

(b) Daily capacities of machines.

Machine m	Day 1 $q_{m,1}$	2 $q_{m,2}$	3 $q_{m,3}$	4 $q_{m,4}$	5 $q_{m,5}$
1	8	8	0	8	8
2	20	0	0	20	20
3	4	4	4	4	4

(c) Machines, processing times, and requirements of production tasks.

Job j	Task t	Machine m_t	Processing time p_t	Gap days g_t	Coupled c_t
1	1	1	8	0	0
1	2	1	2	0	1
2	1	2	10	0	0
2	2	1	2	0	0
2	3	3	3	1	0
3	1	2	0.5	0	0
4	1	3	2.5	0	0

always comes with $g_t = 0$ gap days, as conflicting requirements to postpone and proceed with processing t would be imposed otherwise. Unlike that, $g_t = 1$ for $t = t_3^2$, i.e., the third task of the second job, means that at least one day must lie in-between performing t_2^2 and t_3^2 , e.g., for transport to a different manufacturing site.

Partial schedules considering only the first or second job, respectively, are displayed in Figure 1a. Regarding the first job and its two coupled tasks in the upper part, t_1^1 is scheduled to the earliest start day $e_1 = 2$ and occupies the full capacity $q_{1,2} = 8$ of machine 1 on day 2. The coupled task t_2^1 must be scheduled to the next day on which machine 1 is available, which is day 4 in view of $q_{1,3} = 0$, and 6 time units of $q_{1,4} = 8$ are left for processing any other jobs' tasks by machine 1 on the same day. Turning to the second job and its three tasks in the lower part, t_1^2 is scheduled to day $e_2 = 1$ and utilizes half of the capacity $q_{2,1} = 20$ of machine 2. The successor task t_2^2 is processed by machine 1 directly on the next day 2, occupying 2 time units of the capacity $q_{1,2} = 8$. Given the gap days $g_t = 1$ for the remaining task $t = t_3^2$, the earliest day for performing t_3^2 is 4, and it takes 3 time units of the capacity $q_{3,4} = 4$ of machine 3. Scheduling t_3^2 to day 4 is required for finishing the second job within its deadline $d_2 = 4$, while a penalty weighted by $w_2 = 2$ would apply if delaying t_3^2 to day 5.

An entire schedule for the example instance in Table 1 is shown in Figure 1b. Here the three tasks of the second job are scheduled as discussed before, so that the second job is finished within its deadline. The short tasks t_1^3 and t_1^4 of the third and fourth job are additionally processed by machine 2 or 3, respectively, on the earliest start day $e_3 = e_4 = 1$. This is possible because the sum of processing times 10 and 0.5 of t_1^2 and t_1^3 stays within the capacity $q_{2,1} = 20$ of machine 2, and $q_{3,1} = 4$ also yields the availability of machine 3 to process t_1^4 for the required 2.5 time units. Different from the previous partial schedule for the first job, its first task t_1^1 cannot be scheduled to day 2 anymore because t_2^2 takes part of the capacity $q_{1,2} = 8$ of machine 1, while the full capacity would be required for processing t_1^1 . Given that machine 1 is only available again on day 4, t_1^1 is performed then, and its coupled task t_2^1 on the next day 5. That is, the first job is finished one day later than its deadline $d_1 = 4$, which in view of the weight $w_1 = 3$ leads to the quality $3 \cdot (\omega \cdot (5 - 4) + \omega') = 3 \cdot (\omega + \omega')$ w.r.t. the penalties ω, ω' of the schedule displayed in Figure 1b.

Job 1	Day 1	Day 2	Day 3	Day 4	Day 5
Machine 1	/ 8	8 / 8	/ 0	2 / 8	/ 8
Machine 2	earliest start is day two	/ 0	0 / 0	/ 20	/ 20
Machine 3	/ 4	/ 4	/ 4	/ 4	/ 4

Job 2	Day 1	Day 2	Day 3	Day 4	Day 5
Machine 1	/ 8	2 / 8	/ 0	/ 8	/ 8
Machine 2	10 / 20	/ 0	one rest day before task three	/ 20	/ 20
Machine 3	/ 4	/ 4	/ 4	3 / 4	/ 4

(a) Scheduling two jobs separately based on earliest availability of machines.

All jobs	Day 1	Day 2	Day 3	Day 4	Day 5
Machine 1	/ 8	2 / 8	/ 0	8 / 8	2 / 8
Machine 2	10 / 20	/ 0	/ 0	/ 20	/ 20
Machine 3	2.5 / 4	/ 4	/ 4	3 / 4	/ 4

(b) Schedule assigning the tasks of all four jobs to workdays.

■ Figure 1 Schedules for the example instance in Table 1.

3 Solving approaches

Our main goal is to utilize constraint optimization for solving the production scheduling problem specified in the previous section. This, however, requires the choice of a sufficient horizon in terms of workdays, so that all jobs can be scheduled and their tardiness inspected in order to assess the solution quality. To this end, we start by proposing a greedy algorithm able to quickly produce a sensible custom solution. Beyond the option to timely screen the effects of and react to deviations in resource availabilities and demands, we use greedy solutions to derive a feasible scheduling horizon along with strict limits on the completion of jobs. Exhaustive optimization can then be performed by applying state-of-the-art solvers to the ILP and CP models also presented in this section, where greedy solutions help to warm-start the optimization and converge to high-quality schedules in shorter solving time.

3.1 Greedy algorithm

We have devised a greedy algorithm to heuristically determine a sensible custom solution that yields a feasible scheduling horizon and can also be used to parameterize the constraint optimization performed by ILP and CP solvers. The basic idea is to proceed day by day to greedily schedule pending tasks, whose predecessors (if any) have been processed before, according to some priority until all tasks are scheduled. Letting $a(t_0^j) = e_j - 1$ and $f_i^j = \sum_{i' < i' \leq n_j, t' = t_{i'}^j} (g_{t'} + 1)$ for each job $j \in J$ and $1 \leq i \leq n_j$, the *priority* function we use for pending tasks $t = t_i^j$ and days $k \in \mathbb{N}$ is calculated as follows:

$$priority(t_i^j, k) = \begin{cases} -\infty & \text{if } k \leq a(t_{i-1}^j) + g_t \\ \infty & \text{if } a(t_{i-1}^j) + g_t < k \text{ and } c_t = 1 \\ w_j \cdot \frac{\omega + \omega'}{\exp(d_j - (k + f_i^j))} & \text{if } a(t_{i-1}^j) + g_t < k, c_t = 0, \text{ and } k + f_i^j \leq d_j \\ w_j \cdot \frac{\omega}{\exp(n_j - i + 1)} & \text{otherwise} \end{cases}$$

Algorithm 1 Greedy task scheduling.

```

 $T \leftarrow \{t_1^j \mid j \in J\};$ 
 $k \leftarrow 0;$ 
while  $T \neq \emptyset$  do
   $k \leftarrow k + 1;$ 
   $S \leftarrow$  list of tasks  $t \in T$  in decreasing order of  $\text{priority}(t, k)$ ;
  foreach  $t \in S$  such that  $\text{priority}(t, k) \neq -\infty$  do
    if  $p_t \leq q_{m_t, k}$  then
       $a(t) \leftarrow k;$ 
       $q_{m_t, k} \leftarrow q_{m_t, k} - p_t;$ 
       $T \leftarrow T \setminus \{t\};$ 
      if  $i < n_j$  for  $t = t_i^j$  then  $T \leftarrow T \cup \{t_{i+1}^j\};$ 
    end
  end
end

```

The distinguished priority $-\infty$ is taken for (successor) tasks t that cannot be scheduled to the day k in view of the (positive) number g_t of required gap days. In turn, a coupled successor task t for which $c_t = 1$ must be scheduled to the next day on which its allocated machine is available, and in this case we use the distinguished priority ∞ .

When no hard constraints forbid or force $t = t_i^j$ to be scheduled, we approximate the feasibility of finishing the job j within its deadline d_j according to the condition $k + f_i^j \leq d_j$, which applies if and only if the gap days and workdays needed for successor tasks of t are still left after the day k . If so, we optimistically assume the availability of allocated machines and consider the sum $\omega + \omega'$ of penalties as cost to safe by processing t on day k . To also reflect the criticality of jobs, we reduce this cost exponentially based on the number $d_j - (k + f_i^j)$ of remaining buffer days by which pending tasks of j can be further postponed within the deadline d_j . The outcome is then scaled by the weight w_j to incorporate the importance of j .

The case that remains is that the job j can certainly not be finished within its deadline d_j , so that the penalty $w_j \cdot \omega'$ applies no matter whether $t = t_i^j$ is scheduled to day k or later. This means that only the penalty ω is of further interest, we reduce it exponentially according to the number $n_j - i + 1$ of pending tasks of j , and again scale the outcome by the weight w_j . Among jobs that will certainly be delayed, the priority calculation thus prefers those with fewer remaining tasks, which are presumably easier to complete soon. Arguably, such a scheme may seem unbalanced and at risk to delay jobs needing more work for even longer, but our empirical investigation of greedy heuristics led to best schedules with the described prioritization strategy, and constraint optimization later goes for improvements.

Algorithm 1 outlines our greedy method by pseudo-code, whose central part is to traverse the pending tasks $t \in T$ per day k in decreasing order of priority. When the daily capacity $q_{m_t, k}$ of the allocated machine m_t suffices to process t , we schedule t and subtract its processing time p_t from the machine capacity before checking whether other tasks of lower priority can be performed in addition. If a scheduled task $t = t_i^j$ has the successor t_{i+1}^j , the latter is added to the set T of pending tasks and can be processed from day $k + 1$ on, where the greedy scheduling proceeds when no further task can be performed within the available machine capacities on day k . Given our assumption of uniform positive capacities for days on which machines are available, the splitting of longer tasks into coupled parts with processing times up the capacity of their allocated machine, and weekly repeating machine capacity patterns (except for occasional holidays, maintenance, or extra shifts), Algorithm 1 will eventually succeed to schedule all tasks and thus yield a sufficient scheduling horizon.

Reconsidering our example instance discussed in Section 2.1, all tasks that can be processed on day 1, i.e., t_1^2 , t_1^3 , and t_1^4 , are scheduled greedily, as shown in Figure 1b. The pending tasks on day 2, which matches the earliest start $e_1 = 2$ of job 1, are $T = \{t_1^1, t_2^2\}$, both tasks compete for machine 1, and we have to inspect their priorities to decide whether to process t_1^1 or t_2^2 . As displayed in Figure 1a, it is feasible to finish both tasks within their common deadline $d_1 = d_2 = 4$, so that we have to consider the gap days and workdays needed for successor tasks: $f_1^1 = 0 + 1 = 1$ and $f_2^2 = 1 + 1 = 2$ in view of t_2^1 or t_3^2 , respectively. Together with the weights $w_1 = 3$ and $w_2 = 2$, this yields $\text{priority}(t_1^1, 2) = 3 \cdot (\omega + \omega')/e \leq 2 \cdot (\omega + \omega')/1 = \text{priority}(t_2^2, 2)$, where the priority of t_2^2 for day 2 is strictly greater whenever $\omega + \omega' \neq 0$. Hence, we greedily schedule t_2^2 to day 2, and then the remaining tasks t_1^1 , t_2^1 , and t_3^2 to the next days on which they can be processed w.r.t. the availability of machine 1 and the gap day required before t_3^2 . This reproduces the schedule shown in Figure 1b by means of our greedy scheduling strategy.

3.2 ILP model

In order to fix a feasible scheduling horizon to be investigated by means of constraint-based optimization techniques, we take advantage of the greedy solution determined by our heuristic strategy. For each job $j \in J$, we restrict the processing of its tasks to the latest day $r_j = a(t_{n_j}^j) + b$, where $a(t_{n_j}^j)$ is the completion day of j in the greedy solution and $b \in \mathbb{N}$ is some constant. This yields the *range* from the earliest start day e_j until r_j as period of workdays to which tasks of j can possibly be scheduled, and globally leads to $h = \max\{r_j \mid j \in J\}$ as sufficient scheduling horizon. For our experiments in Section 4, we use $b = 10$ to admit up to 10 days later job completion than in the greedy solution, with the hope that any such local degradations allow for schedules of better overall quality. Notably, restricting the range of days for performing jobs based on the greedy solution avoids infeasibility due to enforcing too tight deadlines and may also benefit optimization performance by not overstressing the scheduling horizon in view of potentially far deadlines. Moreover, the derived range r_j implies $l_j = \max\{0, r_j - d_j\}$ as *limit* on the number of days by which a job j can be delayed beyond its deadline d_j in the worst case.

Our ILP model for task scheduling is shown in Figure 2, starting with a summary of instance parameters as introduced in Section 2, augmented by auxiliary functions u_m mapping a day k to the number of days up to k on which the machine $m \in M$ is available, the range r_j for job $j \in J$ derived from a greedy solution, and its limit l_j on delayed completion. The decision variables include Booleans $a_{t_i^j, k}$ to indicate that the task t_i^j is scheduled to a day k in the period from the earliest start e_j to the range r_j of its job j , a numerical variable z_j whose natural value up to l_j provides the delay days of j , and a Boolean z'_j signaling that the completion of j is delayed by at least one day. Hence, the objective function (7) matches the weighted sum $\sum_{j \in J, a(t_{n_j}^j) > d_j} w_j \cdot (\omega \cdot (a(t_{n_j}^j) - d_j) + \omega')$ subject to penalties $\omega, \omega' \in \mathbb{N}$.

The first constraint (1) expresses that each task t_i^j must be scheduled to exactly one day between the earliest start e_j and range r_j of j . Daily machine capacities $q_{m,k}$ are checked by the constraint (2), making sure that the sum of processing times p_t over all tasks t processed by machine m on day k does not exceed $q_{m,k}$. The constraint (3) addresses the gap days $g_{t_i^j}$ that must lie in-between the predecessor t_{i-1}^j and a task t_i^j with $1 < i$. For example, we obtain $\sum_{1 \leq k \leq 5} k \cdot a_{t_2^2, k} + 1 < \sum_{1 \leq k \leq 5} k \cdot a_{t_3^2, k}$ for the tasks t_2^2 and t_3^2 of the instance discussed in Section 2.1, where $g_{t_3^2} = 1$ indicates that t_3^2 cannot be scheduled directly to the next day after performing t_2^2 . Coupled tasks t_{i-1}^j and t_i^j are handled by the constraint (4), requiring that t_i^j is processed on the next day such that its allocated machine $m_{t_i^j}$ is available, where coefficients $u_{m_{t_i^j}}(k)$ map the availability days k of $m_{t_i^j}$ to consecutive natural numbers. Regarding

Parameters

$k \in D \setminus \{0\}$	day / set of days
$m \in M$	machine / set of machines
$q_{m,k} \in \mathbb{Q}^+ \cup \{0\}$	capacity of machine m for day k
$j \in J$	job / set of jobs
$e_j \in D \setminus \{0\}$	earliest start day of job j
$d_j \in D$	deadline of job j
$w_j \in \mathbb{N}$	weight of job j in objective function
$t_i^j \in T$	i -th task of job j / set of tasks
$m_t \in M$	allocated machine of task t
$p_t \in \mathbb{Q}^+$	processing time of task t
$g_t \in \mathbb{N}$	gap days of task t
$c_t \in \{0, 1\}$	coupling flag of task t
ω, ω'	penalty for each delay day / each delayed job
$u_m : D \rightarrow D$	function defined by $k \mapsto \{1 \leq k' \leq k \mid q_{m,k'} \neq 0\} $
$r_j \in D \setminus \{0\}$	range specifying the latest day to complete job j
$l_j \in \mathbb{N}$	limit on the delay days of job j beyond its deadline d_j

Decision variables

$a_{t_i^j,k} \in \{0, 1\}$	1 if task t_i^j is scheduled to day k with $e_j \leq k \leq r_j$, 0 otherwise
$z_j \in \{0, \dots, l_j\}$	delay days of job j , limited by l_j
$z'_j \in \{0, 1\}$	1 if job j is delayed, 0 otherwise

Constraints

$\sum_{k \in D \setminus \{0\}} a_{t_i^j,k} = 1 \quad \forall t_i^j \in T$	task assignment (1)
$\sum_{t \in T, m_t = m} p_t \cdot a_{t,k} \leq q_{m,k} \quad \forall m \in M, \forall k \in D \setminus \{0\}$	machine capacities (2)
$\sum_{k \in D \setminus \{0\}} k \cdot a_{t_{i-1}^j,k} + g_{t_i^j} < \sum_{k \in D \setminus \{0\}} k \cdot a_{t_i^j,k} \quad \forall t_i^j \in T, 1 < i$	gap days (3)
$\sum_{k \in D \setminus \{0\}} u_{m_{t_i^j}}(k) \cdot a_{t_{i-1}^j,k} + 1 = \sum_{k \in D \setminus \{0\}} u_{m_{t_i^j}}(k) \cdot a_{t_i^j,k} \quad \forall t_i^j \in T, c_{t_i^j} = 1$	coupled tasks (4)
$\sum_{k \in D \setminus \{0\}} k \cdot a_{t_{n_j}^j,k} - d_j \leq z_j \quad \forall j \in J$	delay days (5)
$z_j \leq l_j \cdot z'_j \quad \forall j \in J$	delayed jobs (6)

Objective function

$\min \omega \cdot \sum_{j \in J} w_j \cdot z_j + \omega' \cdot \sum_{j \in J} w_j \cdot z'_j$	weighted sum of delay days and delayed jobs (7)
--	---

■ **Figure 2** ILP model for task scheduling.

the coupled tasks t_1^1 and t_2^1 of our example in Section 2.1, this scheme gives the constraint $1 \cdot a_{t_1^1,1} + 2 \cdot a_{t_1^1,2} + 2 \cdot a_{t_1^1,3} + 3 \cdot a_{t_1^1,4} + 4 \cdot a_{t_1^1,5} + 1 = 1 \cdot a_{t_2^1,1} + 2 \cdot a_{t_2^1,2} + 2 \cdot a_{t_2^1,3} + 3 \cdot a_{t_2^1,4} + 4 \cdot a_{t_2^1,5}$, in which the coefficients do not increase for day 3 in view of the unavailability of machine $m_{t_2^1} = m_{t_1^1} = 1$. The remaining constraints (5) and (6) impose lower bounds on the variables z_j and z'_j , reflecting the delay days or delayed completion, respectively, of a job j . For the first job of the instance in Section 2.1 with its deadline $d_1 = 4$, we obtain the specific

constraints $\sum_{1 \leq k \leq 5} k \cdot a_{t_2^1, k} - 4 \leq z_1$ and $z_1 \leq 1 \cdot z_1'$. This necessitates $z_1 = z_1' = 1$ when $a_{t_2^1, 5}$ signals that the last task t_2^1 of the job is processed on day 5, as done according to the schedule in Figure 1b, so that the cost $3 \cdot (\omega + \omega')$ is included in the objective function (7). Any further constraints imposing upper bounds and thus fixing the values of z_j and z_j' would be redundant, since minimization of the objective function also aims at assigning smallest feasible values and optimal solutions for the ILP model in Figure 2 readily give best schedules.

Recalling the role of a greedy solution, we use it for restricting the days to process jobs, thus reducing the representation size of our ILP model and targeting solvers towards better schedules in the neighborhood of the greedy solution, while even better schedules that entirely differ may be excluded. Given the high complexity of our industrial scheduling domain with thousands of tasks to be scheduled over several weeks, as empirically studied in Section 4, we make this trade-off and do not insist on schedules of theoretically best quality. However, we observed that admitting up to 10 days later job completion than in the greedy solution gives loose ranges for our instances, where constraint optimization yields schedules such that by far most jobs are completed earlier than in the greedy solution. The quality of schedules obtainable in reasonable solving time is also higher than with further relaxed ranges that increase likewise the representation size and the search space of instances. That is, we could not empirically confirm potential theoretical advantages due to extended limits on the completion of jobs, and thus up to 10 more days than in the greedy solution appear sufficiently cautious to us. Moreover, our experiments demonstrate that warm-starting solvers with a greedy solution, giving an initial hint on promising task assignments, significantly improves their optimization performance.

3.3 CP model

Given that linear constraints over finite-domain variables are supported by CP solvers (and rational coefficients can be scaled to integers without loss of precision), our CP model for task scheduling is primarily a syntactic reformulation of the constraints and objective function in Figure 2. However, rather than taking Booleans $a_{t_i^j, k}$ to represent that a task t_i^j is scheduled to day k , we use interval variables $a_{t_i^j}$ with associated duration 1 within the period from the earliest start e_j to the range r_j of the job j . This enables a convenient modeling of the constraint (2) for machine capacities in terms of the *cumulative* global constraint [12]. Since *cumulative* assumes the capacity $q_{m, k}$ of a machine $m \in M$ to be the same on each day k in the scheduling horizon, we use the uniform positive capacity on availability days of m as constant threshold, and model unavailability on a day k by adding a fixed task t taking the full capacity of m as processing time p_t . The solutions and objective function values then correspond one-to-one between our ILP and CP models, where either a Boolean $a_{t_i^j, k}$ or a variable assignment $a_{t_i^j} = k$ indicates the day k for performing a task t_i^j .

Our motivation for devising two models of similar functionality is that decision variables, linear constraints, and objective functions can be conveniently expressed in both formats, so the extra effort for modeling is modest, while the respective state-of-the-art solvers feature complementary constraint-based optimization techniques. A major advantage of ILP solvers is the estimation of solution quality based on the duality gap to relaxed real-valued solutions for a model [1]. Support of global constraints with dedicated propagation methods is a particular strength of CP solvers, where we make use of *cumulative* to limit machine capacities more compactly than by separate linear constraints for each day in the scheduling horizon.

4 Experimental analysis

We empirically evaluate the optimization performance and solution quality achieved with our scheduling methods on problem instances extracted from production data by Kostwein Holding GmbH, operating in the build-to-print business. The used instance sets and solver settings are described first, and then we present the results of our empirical evaluation.

4.1 Experimental setup

The production data supplied by Kostwein Holding GmbH contains a complete list of customer orders taken as snapshot from the company's ERP system. While these orders may have substantial lead times with planned delivery dates several months ahead, the workflows can possibly change during the production process due to customer requests, and new orders regularly come in between one data export and another. Hence, a production schedule incorporating all present orders will have a horizon of several months, yet be subject to revision in a few days at latest, so that filtering orders and their planned workflows to focus on a shorter scheduling horizon up to a few weeks is advisable in practice. A list of daily machine capacities constitutes the second kind of input, which usually follow a weekly pattern apart from bank holidays, planned maintenance, and occasionally extra shifts on weekends to manage peak loads. This matches our current assumption of uniform machine capacities on availability days in order to split long tasks into coupled parts occupying a machine for days.

The problem instances for our experiments are based on six customer order lists along with production workflows exported at different weeks. Following the idea that schedules should focus on the near to mid-term future, we extracted the jobs whose earliest start days lie within the next two, four, or six weeks, respectively, thus obtaining three instance sets with six realistic scenarios of roughly same size in each. In fact, the extracted jobs amount to about 6000 tasks per 2-weeks period, so that the average number of tasks to be scheduled is around 6000, 12000, or 18000 depending on the instance set.

In preliminary experiments, we compared the ILP solvers Gurobi¹ and IBM CPLEX Optimizer³ as well as the CP solvers Google OR-tools² and IBM ILOG CP Optimizer⁴. Both the two ILP and the two CP solvers showed comparable performance on small problem instances, with a slight tendency in favor of Gurobi or Google OR-tools, respectively, when the instance size grows. We thus run Gurobi and Google OR-tools in our systematic experiments.

We conducted our experiments on a machine equipped with two Intel Xeon 6138 CPUs, providing 40 cores and offering 80 parallel threads. While Google OR-tools (version 8.2.8710) is configured to exploit all 80 threads, Gurobi (version 9.1.1) runs 32 threads, which is the default recommended by the developers, as more threads can in some cases deteriorate the optimization performance. The penalties for each delay day or delayed job, respectively, are fixed to $\omega = 1$ and $\omega' = 3$, so that entirely avoiding a delay counts more than reducing the delay length just by single days. Aiming at few delayed jobs makes practical sense because the delays point out bottlenecks that may a posteriori be resolved by including extra shifts or delegating critical production tasks to external suppliers. We report average objective function values along with the standard deviation relative to greedy solutions, determined by means of the heuristic algorithm in Section 3.1, for time limits of 120, 600, and 1800 seconds when the solvers are warm-started with greedy solutions. Without warm-start, we restrict

³ <https://www.ibm.com/analytics/cplex-optimizer/>

⁴ <https://www.ibm.com/analytics/cplex-cp-optimizer/>

■ **Table 2** Objective function values relative to the greedy solution for warm-started solvers with 120s, 600s, and 1800s solving time limit, based on six instances per number of weeks, where respective gaps are additionally given for ILP. Results without warm-start, for 1800s solving time limit, include the number of instances for which a solution was found. No run terminated before the time limit.

Weeks			Greedy	CP (solved by Google OR-tools)				
	Time limit, start		N/A	120s, warm	600s, warm	1800s, warm	1800s, no warm	
	Number of tasks		Objective	Objective			Objective	Solved
2	6543 \pm 1020		1.00	0.84 \pm 0.04	0.65 \pm 0.08	0.53 \pm 0.11	0.59 \pm 0.11	5
4	12204 \pm 1128		1.00	0.96 \pm 0.01	0.87 \pm 0.03	0.78 \pm 0.03	-	0
6	17387 \pm 455		1.00	0.98 \pm 0.00	0.94 \pm 0.01	0.89 \pm 0.02	-	0
Weeks	ILP (solved by Gurobi)							
	120s, warm		600s, warm		1800s, warm		1800s, no warm	
	Objective	Gap	Objective	Gap	Objective	Gap	Objective	Solved
2	0.55 \pm 0.11	0.40 \pm 0.18	0.41 \pm 0.16	0.25 \pm 0.10	0.35 \pm 0.18	0.18 \pm 0.08	2.00 \pm 0.00	1
4	0.88 \pm 0.08	0.55 \pm 0.13	0.73 \pm 0.03	0.43 \pm 0.09	0.66 \pm 0.07	0.25 \pm 0.10	-	0
6	1.00 \pm 0.02	0.60 \pm 0.05	0.99 \pm 0.03	0.55 \pm 0.05	0.99 \pm 0.03	0.52 \pm 0.05	-	0

the comparison to 1800 seconds because the optimization performance declines dramatically and plenty runs do not even return a feasible solution within the solving time limit. For the ILP solver Gurobi, which reports the duality gap to relaxed real-valued solutions, we also indicate average gaps and the standard deviation relative to the quality of greedy solutions. Our instance sets and instructions for running the compared solvers are available in the supplementary material.

4.2 Experimental results

Table 2 summarizes the results of our empirical evaluation. The instance sets based on jobs with the earliest start day up to two, four, or six weeks in the future are listed in separate rows, and average objective function values together with further measurements (where applicable) for solvers and their setups are given in respective columns. The average number of tasks for the instance sets including jobs starting differently many weeks ahead is provided first, and the normalized quality 1.00 of greedy solutions is then indicated for reference.

For both the CP solver Google OR-tools and the ILP solver Gurobi, where results for the latter are displayed below the former, we observe that the solution quality improves substantially with increasing solving time limit when the solvers are warm-started with greedy solutions. However, Gurobi has a significant edge on Google OR-tools for the instance sets with jobs starting up to two or four weeks ahead, yielding 18% and 12% better quality of schedules relative to the greedy solution with 1800s time limit for both solvers. These percentages increase even more when shorter solving time limits are taken, such as 29% difference between the best solutions of Gurobi and Google OR-tools in 120s for the 2-weeks instances, and still 14% in 600s for the 4-weeks instances. Unlike that, Google OR-tools performs better than Gurobi on the 6-weeks instances, where its solutions are of 10% better quality with 1800s time limit. We checked that Gurobi here deals with roughly 500,000 linear constraints, and we conjecture that the handling of the *cumulative* global constraint by Google OR-tools is advantageous for instances of such large size. The box plot of average objective function values in Figure 3 also illustrates these clear trends visually.

Regarding the duality gaps provided by Gurobi, they range from 18% for 2-weeks instances to 52% for 6-weeks instances, so that the best schedules found within 1800s solving time limit cannot be claimed optimal, but constitute a trade-off between solving time and solution quality. This indicates that provably optimal schedules for our instances of industrial size



■ **Figure 3** Plotted objective function values for ILP and CP solvers relative to the greedy solution.

are beyond reach, and compromises have to be made. Limiting the considered jobs to those starting at most two to four weeks in the future already yields substantial improvements by the ILP solver Gurobi in comparison to greedy solutions deemed sensible. Moreover, the poor results of Gurobi and Google OR-tools without warm-start in Table 2, whose runs fail to return any feasible solution except for some 2-weeks instances, emphasize that a heuristic algorithm and constraint-based optimization techniques form a worthwhile combination to come to high-quality schedules in reasonable solving time.

5 Web interface

The presented scheduling methods are integrated as back-ends of a web application, whose user interface is illustrated in Figure 4, supporting the analysis of production planning scenarios at Kostwein Holding GmbH. To this end, production managers can upload customer order lists including the workflows of manufacturing processes exported from the company's ERP system. A second file provides the daily machine capacities by calendar dates, and the screenshot in Figure 4a indicates such input files in the upper left menu.

Before running solvers to compute schedules, the jobs can be filtered based on their earliest start days or deadlines to restrict the scope of the considered problem instance. As displayed in Figure 4a, our web interface allows for visually inspecting instance properties like the length distribution of tasks to be scheduled and the accumulated workloads of machines, which is helpful to spot critical resources and potential bottlenecks independently of specific production schedules. For example, peak loads of the allocated machines may be rebalanced by modifying the planned workflows and delegating tasks to alternative resources able to process them, or extra shifts on weekends may be included to temporarily increase the machine capacity and compensate the additional working hours by free days at another time. Taking appropriate measures to rebalance high workloads requires specific human experience about the involved manufacturing processes and can thus not be performed automatically in a meaningful way, yet our web application aims to support decision making by facilitating the exploration of possible scenarios like, e.g., the effects of increasing machine capacities.

Once a problem instance has been configured, the main functionality, however, consists of picking back-end solvers and settings to perform the task scheduling. In particular, the penalties ω and ω' for delay days or delayed jobs, respectively, can be adjusted, solving time limits be fixed, and for Gurobi also a duality gap below which the optimization is stopped can

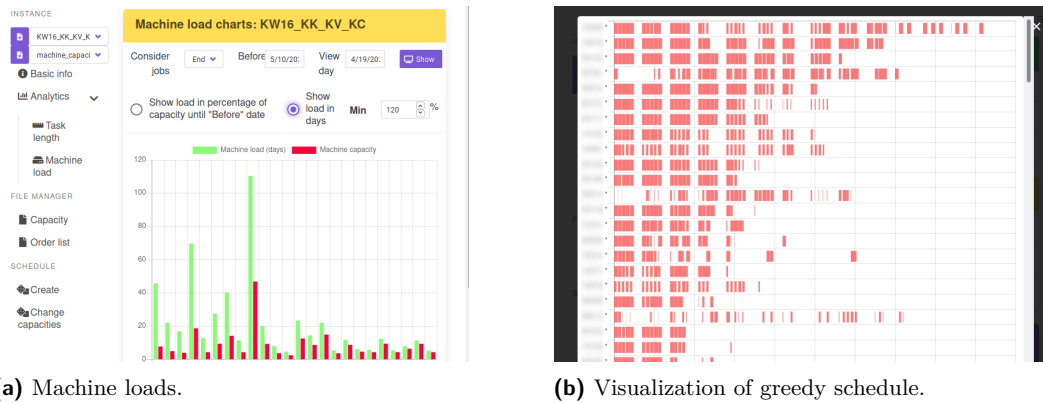


Figure 4 User interface of the web application.

be given. As the experiments in Section 4 show, warm-starting Gurobi or Google OR-tools with a greedy solution virtually always benefits their optimization performance, so that the list of solver settings to launch one after the other, which can be compiled through the web interface, should usually include the warm-start option in each of its entries.

While solvers are run, our interface provides feedback about the optimization progress, including the objective function value of the best solution found so far and also the current duality gap for Gurobi. When a run is finished, the best schedule found can be visualized by a day chart, as exemplarily shown for a greedy solution in Figure 4b. Here the machines are sorted in decreasing order of their workloads, recurring idle periods of two days represent weekends, yet idleness of highly loaded machines on other days is presumably due to shortcomings of our greedy strategy and can be improved by constraint-based optimization. While the day charts give an overview of the distribution of machine workloads, the detailed schedules with the days for performing each task are available as editable spreadsheets.

We are currently about to deploy the web application at Kostwein Holding GmbH on a regular basis, with two main use cases in mind: strategic production scheduling for several weeks, resembling the scopes of the 2-weeks and 4-weeks instance sets in Section 4, to be run over night as well as reactive rescheduling during a day, where the number of jobs to consider will be much smaller to give quick feedback and possibly even optimal short-term schedules.

6 Conclusion

Our paper presents an industrial production scheduling problem and proposes three dedicated solving methods. We have devised a greedy algorithm to come up with a feasible custom solution quickly. Constraint optimization by state-of-the-art solvers can benefit the production scheduling process based on the provided ILP and CP models. Notably, we consult greedy solutions to derive feasible ranges of days for performing production tasks, while the deadlines given for jobs may be too tight for allocating all tasks within the available machine capacities. The deadlines are used to assess the quality of schedules in terms of delay days and delayed jobs, where production managers can then decide on measures to resolve resource bottlenecks.

Regarding the optimization performance, our experiments on problem instances of industrial size and relevance indicate that provably optimal schedules for thousands of production tasks are beyond reach. Nevertheless, the ILP solver Gurobi and the CP solver Google OR-tools successfully exploit the hints by greedy solutions taken to warm-start them and then manage to substantially improve the solution quality in reasonable solving time. While

some care must be taken about the representation size of instances, where around 12000 tasks in our 4-weeks instance set constitute a limit that should better not be exceeded, our empirical results demonstrate the clear advantages of combining a greedy algorithm with constraint-based optimization techniques. The synergy of the greedy and constraint-based methods thus proves to be a practically successful approach. Also taking into account that customer orders and production schedules are frequently subject to revision in our application scenario, longer scheduling horizons than considered in our experiments are of little practical interest, so that the developed scheduling methods scale well to realistic problem sizes.

For applying our scheduling methods in industrial practice, we have developed a user-friendly web interface through which production data and machine capacity profiles can be uploaded and filtered to conveniently specify problem instances. The interface also allows for configuring the penalties used within the objective function for assessing the quality of solutions as well as parameterizing the solvers to run for the optimization of schedules. Both instance properties and returned production schedules can be visually inspected for an accumulated overview of their key features. Since appropriate measures to resolve resource bottlenecks, such as increasing machine capacities, reallocating or delegating tasks, require expert knowledge that is beyond the scope of production scheduling, the web interface is meant to support production managers in exploring possible scenarios and making decisions.

We are currently in the trial phase of confronting our web application regularly with the real production data at Kostwein Holding GmbH, where the evaluation is performed by business experts who are not supposed to need in-depth understanding of the supplied solving methods. The goal is to gather user experience and practical feedback whether the provided functionality and performance are serviceable in the production scheduling process and help to complete customer orders without running into resource bottlenecks. In this respect, our scheduling methods and the encapsulating web application contribute prototypes for experimentation and the further refinement of requirements, where a few immediately compelling directions of future work are discussed in the remainder of this paper.

6.1 Future work

There are a number of opportunities to improve the performance and extend the applicability of the presented scheduling methods. The first consideration is that our greedy algorithm is still ad hoc and based on limited experiments with a handful of heuristics. Arguably, the instances of our scheduling problem are related to each other, as rescheduling with partially overlapping jobs is frequently needed in practice. Hence, there is a good chance that machine learning methods can be trained to typical resource demands and availabilities, and thus lead to better custom solutions than our greedy algorithm with manually selected heuristics. As one particularly promising approach, we are investigating natural evolution strategies [26] for training neural networks to provide the priority for greedy task scheduling. It is then an interesting question we did not explore yet whether warm-starting ILP and CP solvers with feasible solutions of better quality further improves their optimization performance.

Long production tasks that occupy a machine for several days are currently split into coupled parts with fixed processing times, based on the assumption of uniform machine capacities on availability days. This working hypothesis has been adopted to keep the initial modeling approaches simple, yet sacrifices flexibility regarding the machine capacity profiles that can be handled properly. Extending our constraint models to support a dynamic splitting mechanism, where the number and processing times of coupled tasks adjust to the available machine capacities, may allow for addressing richer application scenarios. For example, such

features would enable an automatic allocation of extra shifts declared as optional in the input, e.g., for switching between one- and two-shift operation modes based on demands, which at the moment requires the separate inspection of eligible machine capacity profiles.

A third direction of future work for tuning the optimization performance and achieving tighter (near-)optimality guarantees in terms of a small duality gap is to study worthwhile problem decompositions. For example, we may narrow down constraint-based optimization to tasks processed by highly loaded machines and use gap days as abstractions of skipped tasks. The abstracted tasks would then be inserted again in a post-processing phase.

References

- 1 Edward Anderson. A review of duality theory for linear programming over topological vector spaces. *Journal of Mathematical Analysis and Applications*, 97(2):380–392, 1983. doi:10.1016/0022-247X(83)90204-4.
- 2 Francisco Ballestín, Vicente Valls, and Sacramento Quintanilla. Due dates and RCPSP. In Joanna Józefowska and Jan Weglarz, editors, *Perspectives in Modern Project Scheduling*, volume 92 of *International Series in Operations Research & Management Science*, pages 79–104. Springer, 2006. doi:10.1007/978-0-387-33768-5_4.
- 3 Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*, volume 39 of *International Series in Operations Research & Management Science*. Springer, 2001. doi:10.1007/978-1-4615-1479-4.
- 4 Michel Bénichou, Jean-Michel Gauthier, Paul Girodet, Gerard Hentges, Gerard Ribière, and O. Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1(1):76–94, 1971. doi:10.1007/BF01584074.
- 5 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-3.
- 6 Jan Böttcher, Andreas Drexler, Rainer Kolisch, and Frank Salewski. Project scheduling under partially renewable resource constraints. *Management Science*, 45(4):543–559, 1999. doi:10.1287/mnsc.45.4.543.
- 7 Jacques Carlier and Éric Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, 1989. doi:10.1287/mnsc.35.2.164.
- 8 Ripon Chakraborty, Ruhul Sarker, and Daryl Essam. Single mode resource constrained project scheduling with unreliable resources. *Operational Research*, 20(3):1–35, 2020. doi:10.1007/s12351-018-0380-7.
- 9 Giacomo Da Col and Erich Teppan. Industrial size job shop scheduling tackled by present day CP solvers. In Thomas Schiex and Simon de Givry, editors, *Proceedings of the Twenty-fifth International Conference on Principles and Practice of Constraint Programming (CP’19)*, volume 11802 of *Lecture Notes in Computer Science*, pages 144–160. Springer, 2019. doi:10.1007/978-3-030-30048-7_9.
- 10 Laure Drezet and Jean-Charles Billaut. A project scheduling problem with labour constraints and time-dependent activities requirements. *International Journal of Production Economics*, 112(1):217–225, 2008. doi:10.1016/j.ijpe.2006.08.021.
- 11 Jianzhong Du and Joseph Leung. Minimizing total tardiness on one machine is NP-hard. *Mathematics of Operations Research*, 15(3):483–495, 1990. doi:10.1287/moor.15.3.483.
- 12 Steven Gay, Renaud Hartert, and Pierre Schaus. Simple and scalable time-table filtering for the cumulative constraint. In Gilles Pesant, editor, *Proceedings of the Twenty-first International Conference on Principles and Practice of Constraint Programming (CP’15)*, volume 9255 of *Lecture Notes in Computer Science*, pages 149–157. Springer, 2015. doi:10.1007/978-3-319-23219-5_11.
- 13 Iiro Harjunkoski. Deploying scheduling solutions in an industrial environment. *Computers & Chemical Engineering*, 91:127–135, 2016. doi:10.1016/j.compchemeng.2016.03.029.

- 14 Iiro Harjunkoski, Christos Maravelias, Peter Bongers, Pedro Castro, Sebastian Engell, Ignacio Grossmann, John Hooker, Carlos Méndez, Guido Sand, and John Wassick. Scope for industrial applications of production scheduling models and solution methods. *Computers & Chemical Engineering*, 62:161–193, 2014. doi:10.1016/j.compchemeng.2013.12.001.
- 15 Sönke Hartmann and Dirk Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 207(1):1–14, 2010. doi:10.1016/j.ejor.2009.11.005.
- 16 Oliver Holthaus. Efficient dispatching rules for scheduling in a job shop. *International Journal of Production Economics*, 48(1):87–105, 1997. doi:10.1016/S0925-5273(96)00068-0.
- 17 Vladimir Lifschitz. *Answer Set Programming*. Springer, 2019. doi:10.1007/978-3-030-24658-7.
- 18 Leilei Meng, Chaoyong Zhang, Yaping Ren, Biao Zhang, and Chang Lv. Mixed-integer linear programming and constraint programming formulations for solving distributed flexible job shop scheduling problem. *Computers & Industrial Engineering*, 142:Article ID 106347, 2020. doi:10.1016/j.cie.2020.106347.
- 19 Muhammad Nawaz, Emory Enscore, and Inyong Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1):91–95, 1983. doi:10.1016/0305-0483(83)90088-9.
- 20 Klaus Neumann and Christoph Schwindt. Project scheduling with inventory constraints. *Mathematical Methods of Operations Research*, 56:513–533, 2003. doi:10.1007/s001860200251.
- 21 Yves Pochet and Laurence Wolsey. *Production Planning by Mixed Integer Programming*. Springer, 2010. doi:10.1007/0-387-33477-7.
- 22 Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. doi:10.5555/2843512.
- 23 Yuri Sotskov and Natalia Shakhlevich. NP-hardness of shop-scheduling problems with three jobs. *Discrete Applied Mathematics*, 59(3):237–266, 1995. doi:10.1016/0166-218X(95)80004-N.
- 24 Eric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993. doi:10.1016/0377-2217(93)90182-M.
- 25 Bernd Waschneck, André Reichstaller, Lenz Belzner, Thomas Altenmüller, Thomas Bauernhansl, Alexander Knapp, and Andreas Kyek. Optimization of global production scheduling with deep reinforcement learning. *Procedia CIRP*, 72:1264–1269, 2018. doi:10.1016/j.procir.2018.03.212.
- 26 Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *Journal of Machine Learning Research*, 15(1):949–980, 2014. URL: <http://dl.acm.org/citation.cfm?id=2638566>.
- 27 Fatos Xhafa and Ajith Abraham, editors. *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*, volume 128 of *Studies in Computational Intelligence*. Springer, 2008. doi:10.1007/978-3-540-78985-7.

Minimizing Cumulative Batch Processing Time for an Industrial Oven Scheduling Problem

Marie-Louise Lackner¹ ✉

Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, DBAI, TU Wien, Austria

Christoph Mrkvicka ✉

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 GmbH, Wien, Austria

Felix Winter ✉

Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, DBAI, TU Wien, Austria

Abstract

We introduce the Oven Scheduling Problem (OSP), a new parallel batch scheduling problem that arises in the area of electronic component manufacturing. Jobs need to be scheduled to one of several ovens and may be processed simultaneously in one batch if they have compatible requirements. The scheduling of jobs must respect several constraints concerning eligibility and availability of ovens, release dates of jobs, setup times between batches as well as oven capacities. Running the ovens is highly energy-intensive and thus the main objective, besides finishing jobs on time, is to minimize the cumulative batch processing time across all ovens. This objective distinguishes the OSP from other batch processing problems which typically minimize objectives related to makespan, tardiness or lateness.

We propose to solve this NP-hard scheduling problem via constraint programming (CP) and integer linear programming (ILP) and present corresponding CP- and ILP-models. For an experimental evaluation, we introduce a multi-parameter random instance generator to provide a diverse set of problem instances. Using state-of-the-art solvers, we evaluate the quality and compare the performance of our CP- and ILP-models, which could find optimal solutions for many instances. Furthermore, using our models we are able to provide upper bounds for the whole benchmark set including large-scale instances.

2012 ACM Subject Classification Computing methodologies → Planning and scheduling

Keywords and phrases Oven Scheduling Problem, Parallel Batch Processing, Constraint Programming, Integer Linear Programming

Digital Object Identifier 10.4230/LIPIcs.CP.2021.37

Supplementary Material *Software (Source Code, Benchmark Set, and Results):*
<https://cdlab-artis.dbai.tuwien.ac.at/papers/ovenscheduling/>

Funding The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

¹ Corresponding author



1 Introduction

In the electronics industry, many components need to undergo a hardening process which is performed in specialised heat treatment ovens. As running these ovens is a highly energy-intensive task, it is advantageous to group multiple jobs that produce compatible components into batches for simultaneous processing. However, creating an efficient oven schedule is a complex task as several cost objectives related to oven processing time, job tardiness, setup costs and setup times need to be minimized. Furthermore, a multitude of constraints that impose restrictions on the availability, capacity, and eligibility of ovens have to be considered. Due to the inherent complexity of the problem and the large number of jobs that usually have to be batched in real-life scheduling scenarios, efficient automated solution methods are thus needed to find optimized schedules.

Over the last three decades, a wealth of scientific papers investigated batch scheduling problems. Several early problem variants using single machine and parallel machine settings were categorized in [19] and shown to be NP-hard; a more recent literature review can be found in [15]. Batch scheduling problems share the common goal that jobs are processed simultaneously in batches in order to increase efficiency. Besides this common goal, a variety of different problems with unique constraints and solution objectives arise from different applications in the chemical, aeronautical, electronic and steel-producing industry where batch processing machines can appear in the form of autoclaves [13], ovens [12] or kilns [25].

For example, a just-in-time batch scheduling problem that aims to minimize tardiness and earliness objectives has been recently investigated in [18]. Another recent study [25] introduced a batch scheduling problem from the steel industry that includes setup times, release times, as well as due date constraints. Furthermore, a complex two-phase batch scheduling problem from the composites manufacturing industry has been solved with the use of CP and hybrid techniques [20].

Exact methods used for finding optimal schedules on batch processing machines involve dynamic programming [2] for the simplest variants as well as CP- and mixed integer programming (MIP) models. CP-models have e.g. been proposed in [13] and in [10], where both publications consider batch scheduling on a single machine with non-identical job sizes and due dates but without release dates. A novel arc-flow based CP-model for minimizing makespan on parallel batch processing machines was recently proposed in [21]. Branch-and-Bound [1] and Branch-and-Price [17] methods have been investigated as well. As the majority of batch scheduling problems are \mathcal{NP} -hard, exact methods are often not capable of solving large instances within a reasonable time-limit and thus (meta-)heuristic techniques are designed in addition. These range from GRASP approaches [6] and variable neighbourhood search [3], over genetic algorithms [14, 5], ant colony optimization [4] and particle swarm optimization [26] to simulated annealing [7].

In this paper, we introduce the Oven Scheduling Problem (OSP), which is a new real-life batch scheduling problem from the area of electronic component manufacturing. The OSP defines a unique combination of cumulative batch processing time, tardiness, setup cost, and setup time objectives that needs to be minimized. To the best of our knowledge, this objective has not been studied previously in batch scheduling problems. Furthermore, we take special requirements of the manufacturing industry into account. Thus, the problem considers specialized constraints concerning the availability of ovens as well as constraints regarding oven capacity, oven eligibility and job compatibility.

The main contributions of this paper are:

- We introduce and formally specify a new real-life batch scheduling problem.
- We propose solver independent CP- and ILP-models that can be utilized with state-of-the-art solver technology to provide an exact solution approach. In addition we provide on OPL-model for CP Optimizer using interval variables.
- To generate a large instance set, we introduce an innovative multi-parameter random instance generation procedure.
- We provide a construction heuristic that can be used to quickly obtain feasible solutions.
- All our solution methods are extensively evaluated through a series of benchmark experiments, including the evaluation of several search strategies and a warm-start approach. For a sample of 80 benchmark instances, we obtain optimal results for 37 instances, and provide upper bounds on the objective for all instances.

In the following, we first provide a description of the OSP (Section 2) before we introduce the CP model (Section 3). Then we present alternative models and search strategies (Section 4). Afterwards, we introduce a random instance generator and the construction heuristic (Section 5). Finally, we present and discuss experimental results (Section 6).

2 Description of the Oven Scheduling Problem (OSP)

The OSP consists in creating a feasible assignment of jobs to batches and in finding an optimal schedule of these batches on a set of ovens, which we refer to as machines in the remainder of the paper.

Jobs that are assigned to the same batch need to have the same *attribute*; in the context of heat treatment this can be thought of as the temperature at which components need to be processed. Moreover, a batch cannot start before the *release date* of any job assigned to this batch. The *batch processing time* may not be shorter than the minimal processing time of any assigned job and must not be longer than any job's maximal processing time, as this could damage the produced components. Every job can only be assigned to a set of *eligible machines* and machines are further only available during machine-dependent *availability intervals*. Moreover, machines have a maximal *capacity*, which may not be exceeded by the cumulative size of jobs in a single batch. When determining the start and processing times of batches, *setup times* between consecutive batches must also be taken into account. Setup times depend on the ordered pair of attributes of the jobs in the respective batches and are independent of the machine assignments. In the context of heat treatment, this can be thought of as the time required to switch from one temperature to another.

The main objective of the OSP is to minimize the cumulative batch processing time, total setup times and setup costs, as well as the number of tardy jobs. As the minimization of job tardiness usually has the highest priority in practice, the tardiness objective is weighted higher than the other objectives.

In practice the cumulative batch processing time should be minimized as the cost of running an oven depends merely on the processing time of the entire oven batch and not on the number of jobs within a batch. Therefore, running an oven containing a single small order incurs the same costs as running the oven filled to its maximal capacity.

Furthermore, we note that setup costs and setup times are not necessarily correlated. In fact, cooling down an oven from a high to a low temperature might not incur any (energy) costs, but still might require a certain amount of processing time. Setup costs can also capture costs related to personnel involved in the setup operation.

Using the three-field notation introduced by Graham et. al. [8], the OSP can be classified as $\tilde{P}|r_j, \bar{d}_j, maxt_j, b_i, ST_{sd,b}, SC_{sd,b}, \mathcal{E}_j, Av_m|obj$, where \mathcal{E}_j stands for eligible machines and Av_m for availability of machines. A more formal description of the problem constraints and the objective function obj is given in Section 3.

As shown by Uzsoy [22], minimizing makespan on a single batch processing machine is an \mathcal{NP} -hard problem. It follows that the OSP is \mathcal{NP} -hard as well, as minimizing makespan on a single batch processing machine can easily be expressed within an instance of the OSP.

2.1 Instance parameters of the OSP

An instance of the OSP consists of a set $\mathcal{M} = \{1, \dots, k\}$ of machines, a set $\mathcal{J} = \{1, \dots, n\}$ of jobs and a set $\mathcal{A} = \{1, \dots, a\}$ of attributes as well as the length $l \in \mathbb{N}$ of the scheduling horizon. Every machine $m \in \mathcal{M}$ has a maximum capacity c_m and machine availability is further specified in the form of time intervals $[as(m, i), ae(m, i)] \subseteq [0, l]$ where $as(m, i)$ denotes the start- and $ae(m, i)$ the endtime of the i -th interval on machine m . W.l.o.g. we assume that every machine has the same number of availability intervals and denote this number by I where some of these intervals might be empty (i.e. $as(m, i) = ae(m, i)$). Moreover, availability intervals have to be sorted in increasing order (i.e. $as(m, i) \leq ae(m, i) \leq as(m, i+1)$) for all $i \leq I-1$).

Every job $j \in \mathcal{J}$ is specified by the following list of properties:

- A set of eligible machines $\mathcal{E}_j \subseteq \mathcal{M}$.
- An earliest start time (or release time) $et_j \in \mathbb{N}$ with $0 \leq et_j < l$.
- A latest end time (or due date) $lt_j \in \mathbb{N}$ with $et_j < lt_j \leq l$.
- A minimal processing time $mint_j \in \mathbb{N}$ with $min_T \leq mint_j \leq max_T$, where $min_T > 0$ is the overall minimum and $max_T \leq l$ is the overall maximum processing time.
- A maximal processing time $maxt_j \in \mathbb{N}$ with $mint_j \leq maxt_j \leq max_T$.
- A size $s_j \in \mathbb{N}$.
- An attribute $a_j \in \{1, \dots, a\}$.

Moreover, an $(a \times a)$ -matrix of setup times $st = (st(a_i, a_j))_{1 \leq a_i, a_j \leq a}$ and an $(a \times a)$ -matrix of setup costs $sc = (sc(a_i, a_j))_{1 \leq a_i, a_j \leq a}$ are given to denote the setup times (resp. costs) incurred between a batch using attribute a_i and a subsequent batch using attribute a_j . Setup times (resp. costs) are integers in the range $[0, max_{ST}]$ (resp. $[0, max_{SC}]$), where $max_{ST} \leq l$ (resp. $max_{SC} \in \mathbb{N}$) denotes the maximal setup time (resp. maximal setup cost). Note that these matrices are not necessarily symmetric.

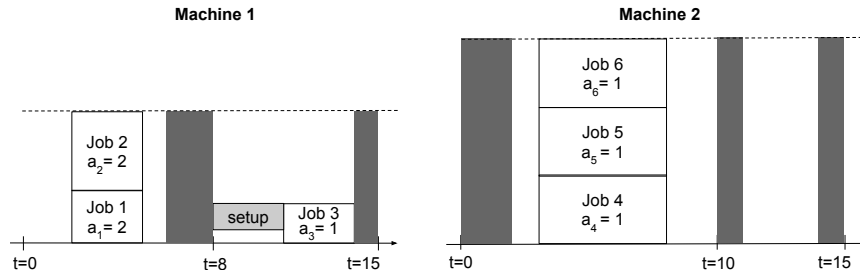
2.2 Example instance with six jobs

Consider the following example for an OSP instance consisting of six jobs, two attributes, two machines and a scheduling horizon of length $l = 15$. The instance parameters are summarized in the following tables and matrices:

Machine	M_1	M_2
c_m	100	150
Availability intervals $[as, ae]$	$[0, 6]$ $[8, 14]$	$[2, 10]$ $[11, 14]$

Job j	1	2	3	4	5	6
\mathcal{E}_j	M_1	M_1	M_1	M_1		
		M_2		M_2	M_2	M_2
et_j	2	0	0	3	0	2
lt_j	10	10	20	20	20	
$mint_j$	3	3	3	5	5	5
$maxt_j$	3	5	5	8	8	10
s_j	40	60	30	50	50	50
a_j	2	2	1	1	1	1

$$st = \begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix} \quad sc = \begin{pmatrix} 0 & 20 \\ 10 & 0 \end{pmatrix}$$



■ **Figure 1** An optimal solution to the OSP for a small example problem.

An optimal² solution of this instance consists of three batches and is visualised in Figure 1. In this visualisation, the dark grey areas correspond to time intervals for which the machine is not available, the gray rectangle before the second batch on machine 1 is the setup time between the first and second batch and the dashed lines represent the machine capacities.

3 CP-model for the OSP

In this section we provide a formal definition of the OSP that will also serve as CP-model. We first explain how batches are modeled and afterwards define decision variables, objective function, and the set of constraints.

In the worst case we need as many batches as there are jobs; we thus define the set of potential batches as $\mathcal{B} = \{B_{1,1}, \dots, B_{1,n}, \dots, B_{k,1}, \dots, B_{k,n}\}$ to model up to n batches for machines 1 to k . In order to break symmetries in the model, we further enforce that batches are sorted in ascending order of their start times and empty batches are scheduled at the end. That is, $B_{m,b+1}$ is the batch following immediately after batch $B_{m,b}$ on machine m for $b \leq n-1$. Clearly, at most n of the $k \cdot n$ potential batches will actually be used and the rest will remain empty.

3.1 Variables

We define the following decision variables:

- Machine assigned to job: $M_j \in \mathcal{M} \quad \forall j \in \mathcal{J}$
- Batch number assigned to job:³ $B_j \in [n] \quad \forall j \in \mathcal{J}$
- Start times of batches: $S_{m,b} \in [0, l] \subset \mathbb{N} \quad \forall m \in \mathcal{M} \quad \forall b \in [n]$
- Processing times of batches: $P_{m,b} \in [0, max_T] \subset \mathbb{N} \quad \forall m \in \mathcal{M} \quad \forall b \in [n]$

Note that M_j and B_j determine to which batch job j is assigned ($B_{(M_j), (B_j)}$). We additionally define the following auxiliary variables:

- Attribute of batch: $A_{m,b} \in [a] \quad \forall m \in \mathcal{M} \quad \forall b \in [n]$
- Availability interval for batch: $I_{m,b} \in [I] \quad \forall m \in \mathcal{M} \quad \forall b \in [n]$
- Number of batches per machine: $b_m \in [n] \quad \forall m \in \mathcal{M}$

² This solution is optimal with respect to weights $\alpha = 4$, $\beta = \gamma = 1$ and $\delta = 100$ as defined in Section 3.2.

³ Throughout the paper, we write $[n]$ for the interval of integers $\{1, \dots, n\}$.

3.2 Objective function

The objective function consists of four components: the cumulative batch processing time across all machines p , the number of tardy jobs t , the cumulative setup times st and the cumulative setup costs sc :

$$\begin{aligned} p &= \sum_{m \in \mathcal{M}, 1 \leq b \leq n} P_{m,b} & t &= |\{j \in \mathcal{J} : S_{M_j, B_j} + P_{M_j, B_j} > lt_j\}| \\ st &= \sum_{\substack{m \in \mathcal{M} \\ 1 \leq b \leq b_m - 1}} st(A_{m,b}, A_{m,b+1}) & sc &= \sum_{\substack{m \in \mathcal{M} \\ 1 \leq b \leq b_m - 1}} sc(A_{m,b}, A_{m,b+1}) \end{aligned} \quad (1)$$

Normalization of cost components. In practice it is important to define an objective function that is highly flexible and configurable. We thus decided in consultation with our industrial partner to define the objective function as a linear combination of the four components. Therefore, the components p , st , sc and t need to be normalized to \tilde{p} , \tilde{st} , \tilde{sc} and $\tilde{t} \in [0, 1]$:

$$\begin{aligned} \tilde{p} &= \frac{p}{avg_t \cdot n} \text{ with } avg_t = \left\lceil \frac{\sum_{j \in \mathcal{J}} mint_j}{n} \right\rceil & \tilde{t} &= \frac{t}{n} \\ \tilde{st} &= \frac{st}{\max(max_{ST}, 1) \cdot n} & \tilde{sc} &= \frac{sc}{\max(max_{SC}, 1) \cdot n} \end{aligned} \quad (2)$$

In the worst case, every batch processes a single job. In this case the total batch processing time is n times the average job processing time avg_t . The setup times and costs are bounded by the maximum setup time resp. cost multiplied with the number of jobs.⁴ We take the maximum of 1 and max_{ST} resp. max_{SC} since it is possible that $max_{ST} = 0$ or $max_{SC} = 0$. The number of tardy jobs is clearly bounded by the total number of jobs.

Finally, the objective function obj is a linear combination of the four normalized components:

$$obj = (\alpha \cdot \tilde{p} + \beta \cdot \tilde{st} + \gamma \cdot \tilde{sc} + \delta \cdot \tilde{t}) / (\alpha + \beta + \gamma + \delta) \in [0, 1] \subset \mathbb{R} \quad (3)$$

where the weights α , β , γ and δ take integer values. Together with our industrial partner, we chose the default values to be $\alpha = 4$, $\beta = 1$, $\gamma = 1$, and $\delta = 100$, which captures the requirements of typical practical scheduling applications.

Integer-valued objective. As some state-of-the-art CP solvers can only handle integer domains, we propose an alternative objective function obj' , where we additionally multiply \tilde{p} , \tilde{st} , \tilde{sc} , and \tilde{t} by the number of jobs and the least common multiple of avg_t , max_{ST} and max_{SC} :

$$\begin{aligned} obj' &= C \cdot n \cdot (\alpha + \beta + \gamma + \delta) \cdot obj \in \mathbb{N} \\ &= \frac{\alpha \cdot C}{avg_t} \cdot p + \frac{\beta \cdot C}{\max(max_{ST}, 1)} \cdot st + \frac{\gamma \cdot C}{\max(max_{SC}, 1)} \cdot sc + \delta \cdot C \cdot t, \\ &\text{where } C = \text{lcm}(avg_t, \max(max_{ST}, 1), \max(max_{SC}, 1)). \end{aligned} \quad (4)$$

Preliminary experiments using the MIP solver Gurobi showed that using obj and obj' both lead to similar results. We therefore used only obj' in our final experimental evaluation.

⁴ Actually, $st \leq (n-1) \cdot max_{ST}$ and $sc \leq (n-1) \cdot max_{SC}$ since no setup is necessary before the first batch. However, we want to keep the least common multiple of the denominators in equation (2) small in favor of the definition of obj' .

3.3 Constraints

In what follows, we formally define the constraints of the OSP using a high-level CP modeling notation. Most of these constraints can directly be handled by CP solvers, however we implicitly make use of constraint reification to express conditional sums and additionally use the maximum global constraint. Furthermore, we implicitly utilize the element constraint to use variables as array indices.

- Jobs may not start before their earliest start time: $S_{M_j, B_j} \geq et_j \quad \forall j \in \mathcal{J}$.
- Batch processing times must lie between the minimal and maximal processing time of all assigned jobs:

$$\begin{aligned} P_{m,b} &= \max(\min_{j: j \in \mathcal{J} \text{ with } B_j = b \wedge M_j = m} p_j) \quad \forall m \in \mathcal{M}, b \in [b_m] \\ P_{M_j, B_j} &\leq \max_{j \in \mathcal{J}} p_j \end{aligned}$$

- Batches on the same machine may not overlap and setup times must be considered between consecutive batches:

$$S_{m,b} + P_{m,b} + st_{(A_{m,b}, A_{m,b+1})} \leq S_{m,b+1} \quad \forall b \in [b_m - 1],$$

- Batches and the preceding setup times must lie entirely within one machine availability interval. In practice an interval for which a machine is unavailable can also represent a period for which the personnel required for the setup and running of ovens is unavailable. Therefore, setup times also need to fall completely within the associated availability interval. This is modeled with auxiliary variables $I_{m,b}$ which encode in which availability interval batch $B_{m,b}$ lies:

$$\begin{aligned} I_{m,b} &= \max(i \in [I] : S_{m,b} \geq as(m, i)) & \forall m \in \mathcal{M} \forall b \in [b] \\ S_{m,b} + P_{m,b} &\leq ae(m, I_{m,b}) & \forall m \in \mathcal{M} \forall b \in [b_m] \\ S_{m,b} - st_{(A_{m,b-1}, A_{m,b})} &\geq as(m, I_{m,b}) & \forall m \in \mathcal{M}, \forall b \in \{2, \dots, b_m\}. \end{aligned}$$

- Total batch size must be less than machine capacity:

$$\sum_{j \in \mathcal{J}: M_j = m \wedge B_j = b} s_j \leq c_m \quad \forall m \in \mathcal{M} \text{ with } b_m > 0, \forall b \in [b_m]$$

- Jobs in one batch must have the same attribute, which we model with auxiliary variables $A_{m,b}$ to set the attribute of a batch: $A_{M_j, B_j} = a_j \quad \forall j \in \mathcal{J}$.
- The assigned machine must be eligible for a job: $M_j \in \mathcal{E}_j \quad \forall j \in \mathcal{J}$.
- Set the number of batches per machine variables: $b_{M_j} \geq B_j \quad \forall j \in \mathcal{J}$
- Set variables for empty batches (i.e. batches $B_{m,b}$ with $b > b_m$):

$$S_{m,b} = l \quad P_{m,b} = 0 \quad A_{m,b} = 1 \quad I_{m,b} = I \quad \forall m \in \mathcal{M}, b_m < b \leq n$$

4 Alternative models and search strategies

4.1 ILP-model for the OSP

We propose a ILP-formulation, where batches are modeled similarly as in the CP-model ($B_{m,b}$ with $m \in \mathcal{M}, b \in [n]$), but we use a different set of decision variables: Binary variables $X_{m,b,j}$ encode whether job j is assigned to batch $B_{m,b}$ ($X_{m,b,j} = 1 \Leftrightarrow (B_j = b \wedge M_j = m)$), and integer variables $S_{m,b}$ and $P_{m,b}$ encode the start and processing times of batches.

37:8 Minimizing Batch Processing Time for an Oven Scheduling Problem

To handle empty batches, we define an additional attribute with value 0 and extend the matrices of setup times \bar{st} and setup costs \bar{sc} so that no costs occur when transitioning from an arbitrary batch to an empty batch: $\bar{st}(a_i, a_j) = \bar{sc}(a_i, a_j) = 0$ if $a_i = 0$ or $a_j = 0$. Moreover, we add a machine availability interval $[l, l]$ of length 0 to the list of availability intervals so that empty batches can be scheduled for this interval (the maximum number of intervals per machine therefore becomes $I + 1$). We then model the problem as follows:⁵

$$\text{Min. } \text{obj}' = \tilde{\alpha} \cdot p + \tilde{\beta} \cdot st + \tilde{\gamma} \cdot sc + \tilde{\delta} \cdot t, \text{ where} \quad (5)$$

$$p = \sum_{\substack{m \in \mathcal{M} \\ 1 \leq b \leq n}} P_{m,b}, \quad t = \sum_{\substack{j \in \mathcal{J}, m \in \mathcal{M} \\ 1 \leq b \leq n}} T_{m,b,j},$$

$$st = \sum_{\substack{m \in \mathcal{M} \\ 1 \leq b \leq n-1}} st_{m,b}, \quad sc = \sum_{\substack{m \in \mathcal{M} \\ 1 \leq b \leq n-1}} sc_{m,b},$$

$$\text{s.t. } \sum_{m \in \mathcal{M}, 1 \leq b \leq n} X_{m,b,j} = 1 \quad \forall j \quad (6)$$

$$\sum_{m \in \mathcal{E}_j, 1 \leq b \leq n} X_{m,b,j} = 1 \quad \forall j \quad (7)$$

$$S_{m,b} \geq et_j \cdot X_{m,b,j} \quad \forall m, \forall b, \forall j \quad (8)$$

$$mint_j \cdot X_{m,b,j} \leq P_{m,b} \quad \wedge$$

$$P_{m,b} \leq max_j \cdot X_{m,b,j} + max_T \cdot (1 - X_{m,b,j}) \quad \forall m, \forall b, \forall j \quad (9)$$

$$S_{m,b+1} \geq S_{m,b} + P_{m,b} + st_{m,b} \quad \forall m, \forall b \leq n-1 \quad (10)$$

$$\sum_{j \in \mathcal{J}} s_j \cdot X_{m,b,j} \leq c_m \quad \forall m, \forall b \quad (11)$$

$$a_j \cdot X_{m,b,j} \leq A_{m,b} \quad \wedge$$

$$A_{m,b} \leq a_j \cdot X_{m,b,j} + a \cdot (1 - X_{m,b,j}) \quad \forall m, \forall b, \forall j \quad (12)$$

$$as(m, i) \cdot I_{m,b,i} \leq S_{m,b} \quad \wedge$$

$$S_{m,b} \leq ae(m, i) \cdot I_{m,b,i} + l \cdot (1 - I_{m,b,i}) \quad \forall m, \forall b, \forall i \quad (13)$$

$$\sum_{1 \leq i \leq I+1} I_{m,b,i} = 1 \quad \forall m, \forall b \quad (14)$$

$$as(m, i) \cdot I_{m,b,i} \leq S_{m,b} - st_{m,b-1} \quad \forall m, \forall b \geq 2, \forall i \quad (15)$$

$$S_{m,b} + P_{m,b} \leq ae(m, i) \cdot I_{m,b,i} + l \cdot (1 - I_{m,b,i}) \quad \forall m, \forall b, \forall i \quad (16)$$

$$T_{m,b,j} \leq X_{m,b,j} \quad \forall j, \forall m, \forall b \quad (17)$$

$$S_{m,b} + P_{m,b} \leq (X_{m,b,j} - T_{m,b,j}) \cdot (lt_j - l) + l \quad \forall j, \forall m, \forall b \quad (18)$$

$$S_{m,b} + P_{m,b} + (1 - T_{m,b,j}) \cdot (l + 1) > lt_j \quad \forall j, \forall m, \forall b \quad (19)$$

$$\sum_{j \in \mathcal{J}} X_{m,b,j} \geq 1 - E_{m,b} \quad \forall m, \forall b \quad (20)$$

$$X_{m,b,j} \leq 1 - E_{m,b} \quad \forall m, \forall b, \forall j \quad (21)$$

$$S_{m,b} \geq l \cdot E_{m,b} \quad \forall m, \forall b \quad (22)$$

$$P_{m,b} \leq max_T \cdot (1 - E_{m,b}) \quad \forall m, \forall b \quad (23)$$

$$E_{m,b} \leq I_{m,b,I+1} \quad \forall m, \forall b \quad (24)$$

$$A_{m,b} \leq a \cdot (1 - E_{m,b}) \quad \forall m, \forall b \quad (25)$$

$$E_{m,b} \leq E_{m,b+1} \quad \forall m, \forall b \leq n-1 \quad (26)$$

⁵ If not stated otherwise, $\forall m$ is short for $\forall m \in \mathcal{M}$, $\forall b$ for $\forall b \in [1, n]$, $\forall i$ for $\forall i \in [1, I + 1]$ and $\forall j$ for $\forall j \in \mathcal{J}$.

$$st_{m,b} = st(A_{m,b}, A_{m,b+1}) \quad \forall m, \forall b \leq n-1 \quad (27)$$

$$sc_{m,b} = sc(A_{m,b}, A_{m,b+1}) \quad \forall m, \forall b \leq n-1 \quad (28)$$

$$\begin{aligned} X_{m,b,j} &\in \{0, 1\}, S_{m,b} \in [0, l], P_{m,b} \in [0, max_T], \\ A_{m,b} &\in [0, a], I_{m,b,i} \in \{0, 1\}, E_{m,b} \in \{0, 1\}, \\ st_{m,b} &\in [0, max_{ST}], sc_{m,b} \in [0, max_{SC}] \quad \forall m, \forall b, \forall j \end{aligned} \quad (29)$$

The weights of the objective function (5) are as described in equation (4) in Section 3.2. Constraint (6) ensures that every job is assigned to exactly one batch. Moreover, constraint (7) ensures that jobs can only be assigned to eligible machines. Constraint (8) specifies that a batch may not start before the earliest start of any job in the batch. The processing time of a batch is constrained by equations (9). Constraint (10) imposes additional restrictions on the starting times and ensures that the correct setup times are considered between consecutive batches. Constraint (11) ensures that the machine capacities are not exceeded for any batch. Constraint (12) ensures that jobs in the same batch have the same attribute. The binary auxiliary variables $I_{m,b,i}$ in constraint (13) encode whether batch $B_{m,b}$ is scheduled within the i -th availability interval $[as(m, i), ae(m, i)]$ of machine m . Therefore, if $I_{m,b,i} = 1$, it must hold that $as(m, i) \leq S_{m,b} \leq ae(m, i)$. The redundant constraint (14) ensures that every batch is scheduled within exactly one availability interval. Constraints (15) and (16) ensure that the entire processing time of batch $B_{m,b}$ as well as the preceding setup times $st_{m,b-1}$ (see (27)) lie within a single availability interval.

The binary auxiliary variables $T_{j,m,b}$ encode whether job j in batch $B_{m,b}$ finishes after its latest end date and is used to calculate the number of tardy jobs t . Constraint (17) ensures that $T_{j,m,b} = 1$ is only possible if job j is assigned to batch $B_{m,b}$. If $T_{j,m,b} = 0$, job j must finish before lt_j (Constraint (18)) and if $T_{j,m,b} = 1$, it must hold that $S_{m,b} + P_{m,b} > lt_j$ (Constraint (19)). The binary variables $E_{m,b}$ in equations (20) to (26) encode whether batch $B_{m,b}$ is empty or not. Constraints (20) and (21) ensure that $E_{m,b} = 1$ iff no job is scheduled for batch $B_{m,b}$. The constraints (22) to (24) set the start times, processing times, availability intervals, and attributes for empty batches. Moreover, in order to break symmetries, the list of batches $(B_{m,b})_{1 \leq b \leq n}$ per machine $m \in \mathcal{M}$ is sorted so that all non-empty batches appear first (constraint (26)). Constraint (27) defines the setup times $st_{m,b}$ and (28) the setup costs $sc_{m,b}$ between consecutive batches on the same machine. Finally, equation (29) defines the domains of all decision and helper variables.

We further investigated an alternative CP model using the Optimization Programming Language (OPL) [23]. More details for this alternative model can be found in Appendix A.

4.2 Programmed Search Strategies

We evaluated the performance of our models with the use of several programmed search strategies, which are based on variable- and value selection heuristics. For our experiments, we implemented the search strategies directly in the MiniZinc language using search annotations.

Variable Ordering: In our implemented search strategies we select at first an auxiliary variable that captures the total number of batches. For this variable we always use a minimum value first heuristic to encourage the solver to look for low cost solutions early in the search. Afterwards, we sequentially select decision variables related to a job by assigning the associated batch, machine, batch start time, and batch duration for the job (i.e., $B_1, M_1, S_{(M_1, B_1)}, P_{(M_1, B_1)}, \dots, B_{|\mathcal{J}|}, M_{|\mathcal{J}|}, S_{(M_{|\mathcal{J}|}, B_{|\mathcal{J}|})}, P_{(M_{|\mathcal{J}|}, B_{|\mathcal{J}|})}$).

Variable Selection Heuristics: We use three different variable selection strategies on the set of decision variables that are related to job assignments: *input order* (select variables based on the specified order), *smallest* (select variables that have the smallest values in their domain first, break ties by the specified order), and *first fail* (select variables that have the smallest domains first, break ties by the specified order).

Value Selection Heuristics: We experimented with two different value selection heuristics for the set of variables which is related to job assignments: *min* (the smallest value from a variable domain is assigned first), and *split* (the variable domain is bisected to first exclude the upper half of the domain).

Evaluated Search Strategies: Using the previously defined heuristics we evaluated 8 different programmed search strategies:

1. *default*: Use the solver's default search strategy.
2. *search1*: Assign number of batches first, then continue with the solver's default strategy.
3. *search2*: Assign number of batches first, then continue with *input order* and *min* value selection on the job variables.
4. *search3*: Assign number of batches first, then continue with *smallest* and *min* value selection on the job variables.
5. *search4*: Assign number of batches first, then continue with *first fail* and *min* value selection on the job variables.
6. *search5*: Assign number of batches first, then continue with *input order* and *split* value selection on the job variables.
7. *search6*: Assign number of batches first, then continue with *smallest* and *split* value selection on the job variables.
8. *search7*: Assign number of batches first, then continue with *first fail* and *split* value selection on the job variables.

5 Random instance generator and construction heuristic

5.1 Construction of random instances

The random instance generator we propose is based on random instance generation procedures for related problems from the literature [14, 24]. However, as the existing variants were designed for batch scheduling problems which neither include machine eligibility constraints, machine availability times nor setup costs and times, the random generation of the associated instance parameters is a novel contribution of this paper. The list of parameters for this instance generator is given in Table 1.

Jobs. The list of n jobs is generated as follows. First, for every job j , the minimal processing time $mint_j$ is chosen using a discrete uniform distribution $U(1, max_T)$. For the maximum processing time, there are two options: either there is no upper limit on the processing time of jobs (`max_time = false`), in which case the maximum processing time is set to max_T for all jobs. Or, if `max_time = true`, the maximum processing time for a job is chosen using a discrete uniform distribution $U(mint_j, max_T)$. Next, the earliest start and latest end times are determined for every job. The earliest start time et_j is chosen similarly as in [24] according to a discrete uniform distribution $U(0, \lceil \rho \cdot Z \rceil)$ where $\rho \in [0, 1]$ and $Z = \sum mint_j$ is the total processing time of all jobs. If $\rho = 0$, all jobs are available right at the beginning and as ρ grows, the jobs are released over a longer interval. The latest end time lt_j is chosen as in [14] according to $lt_j = et_j + \lfloor U(1, \phi) \cdot mint_j \rfloor$ where $\phi \geq 1$. If $\phi = 1$, the latest end time is equal to the sum of the earliest start time and the minimum processing time, meaning that

■ **Table 1** List of parameters of the random instance generator.

Name	Description	Values
Parameters relating to jobs		
n	number of jobs	10, 25, 50, 100
max_T	overall maximum processing time	10, 100
max_time	true if jobs have a max. processing time	true, false
ρ	determines spread of earliest start times	0.1, 0.5
ϕ	determines time from earliest start to latest end of job	2, 5
σ	determines number of eligible machines per job	0.2, 0.5
s	maximum job size	5, 20
Parameters relating to attributes		
a	number of attributes	2, 5
s_time	type of setup-time matrix	constant, arbitrary,
= s_cost	type of setup-cost matrix	realistic, symmetric
Parameters relating to machines		
k	number of machines	2, 5
$min_C = s$	lower bound for max. machine capacity (=max. job size)	5, 20
max_C	upper bound for maximum machine capacity	20, 100
τ	lower bound for the fraction of time machines are available	0.25, 0.75
max_I	max. number of availability intervals	5

all jobs must be processed immediately in order to finish on time. As ϕ grows, more time is given for every job to be completed and tardy jobs are less likely. Regarding the set of eligible machines for a job, one machine is chosen at random among all machines. Additional machines are then added to this set with probability σ each. The size s_j and attribute a_j of a job are both chosen at random between 1 and the maximum job size s or number of attributes respectively.

Attributes. The setup times and setup costs matrices can be of four different types: Constant, arbitrary, realistic and symmetric. For the type “constant”, setup times/costs are all equal to a randomly chosen constant between 0 and $\lceil max_T/4 \rceil$. For the type “arbitrary”, every entry is chosen independently at random between 1 and $\lceil max_T/4 \rceil$. For the type “realistic”, setup times/costs between two batches of the same attribute are lower and are chosen independently at random between 0 and $\lceil max_T/8 \rceil$, whereas setup times/costs between different attributes are higher and are chosen between $\lceil max_T/8 \rceil + 1$ and $\lceil max_T/4 \rceil$. For the type “symmetric” a symmetric matrix is generated with random entries between 0 and $\lceil max_T/4 \rceil$.

Machines. The maximum machine capacity c_m is randomly chosen between min_C and max_C where the lower bound min_C is set to the maximum job size s to ensure that every job fits into every machine. For the machine availability times, we first fix the length of the scheduling horizon l . If we assume that every job is processed in a batch of its own and that all jobs are processed on the same machine, the total runtime is at most equal to the sum of all processing times Z plus n times the maximal setup time max_{st} . The parameter $\tau \in (0, 1]$ is a lower bound for the fraction of time that every machine is available. Thus, if max_{et} is the latest earliest start time, all jobs should – on average – be finished at time

$$l = max_{et} + \lceil (Z + n \cdot max_{st}) / (\tau) \rceil$$

which we use to set the length of the scheduling horizon. Note that if the latest end date of a job is greater than this upper bound we simply use it instead. Now, for every one of the k machines, we pick the number of availability intervals I randomly between 1 and max_I . Every interval $[start_i, end_i]$ should be long enough to accommodate at least a single job with minimal processing time $min_T = \min(min_{t_j} : j \in \mathcal{J})$ (plus the necessary setup times). Thus, the minimum distance between two interval start times $start_i$ and $start_{i+1}$ is $d = min_T + max_{st}$. We first pick the start time $start_1$ of the first interval: In order to guarantee that every machine is available at least a fraction τ of the time, $0 \leq start_1 \leq \lfloor l \cdot (1 - \tau) \rfloor$ must hold and in order to leave enough time for all availability intervals, it has to hold $start_1 \leq l - I \cdot d$. Next, for the start times of the remaining intervals, we pick $I - 1$ random integers between $(start_1 + d)$ and $(l - d)$ that are at least d apart. Finally, we determine the end time end_i of the i -th interval:

$$end_i = start_i + \max(d, \lceil U(\tau, 1) \cdot (start_{i+1} - start_i) \rceil)$$

5.2 Construction heuristic

We designed a simple construction heuristic that finds initial solutions for instances of our problem. The heuristic starts at time 0. At every time step, the list of currently available machines and the list of remaining jobs that have already been released and can be processed on one of the machines is generated. Among these jobs, the one with the earliest due date is chosen and assigned to one of the machines if it fits into an availability interval. Once a job is scheduled, the algorithm adds other jobs that are currently available to the same batch if the job's attributes and maximal processing time as well as the machine's capacity allows so. If no job can be scheduled, the time is increased by one and the above procedure is repeated until the end of the scheduling horizon is reached or all jobs have been scheduled.

6 Experimental evaluation

Using our random instance generator, we created a large set of benchmark instances to evaluate the performance of our proposed models. First, we executed the random generator once for every possible configuration of the 15 parameter values specified in Table 1. Thereby we produced 1024 instances for each of the 16 combinations of the parameters n , k and a . Then we randomly selected 5 instances from every set of 1024 instances, creating a set of 80 instances which we used throughout our experiments. This set thus consists of 20 instances each with 10 (instances 1-20), 20 (21-40), 50 (41-60) and 100 jobs (61-80). All benchmark instances turn out to be satisfiable and solvable by the construction heuristic described in Section 5.2. This reflects our real-life industrial application, for which feasible solutions usually can be found heuristically and the main aim is to find cost-minimal schedules. Furthermore, note that in the particular real-life application scheduling scenarios consisting of roughly 50 jobs are considered to be the average use case.

We implemented both the CP- and ILP-model presented in Sections 3 and 4.1 using the high-level constraint modeling language MiniZinc [16] and used recent versions of Chuffed, OR-Tools, CP Optimizer and Gurobi. For Chuffed, OR-Tools and Gurobi, we used all 7 search strategies described in Section 4.2 and compared them with the solvers default search strategy (for CP Optimizer, search strategies are currently not supported by MiniZinc). For Chuffed, we activated the free search parameter which allows the solver to interleave between the given search strategy and its default search. Furthermore, we investigated a warm-start approach with Gurobi (for the other solvers, warm-start is currently not supported by MiniZinc): the

construction heuristic described in Section 5.2 was used to find an initial solution which was then provided to the model. Finally, the OPL-model presented in Section A was run using CP Optimizer in IBM ILOG CPLEX Studio. This results in a total of 53 different combinations of models, solvers and search strategies per instance; the time limit for every one of these combinations was set to one hour per instance. Experiments were run on single cores, using 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.

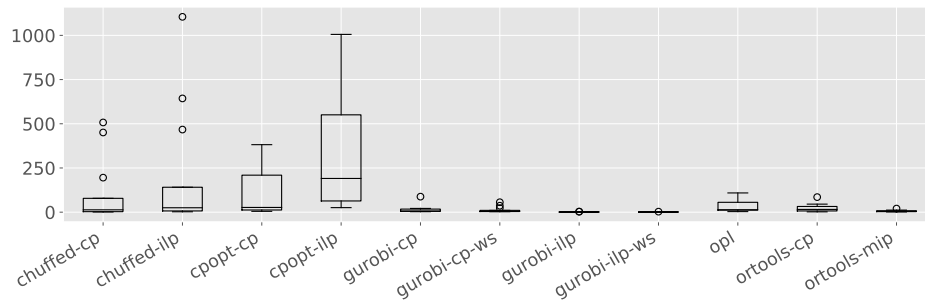
In the following we summarize our findings.⁶ Table 2 provides an overview of the final results produced on the 80 benchmark instances with all evaluated methods. Based on initial experiments with different search strategies we selected the following search strategies solver pairings in the final experiments: *chuffed-cp* with *search3*, *chuffed-ilp* with *search2*, *ortools-cp* with *search6*, *ortools-ilp* with *search2*, and for all other solvers we used the *default* search strategy. The first column in each row denotes the evaluated solver and model. From left to right, columns 2–5 display: the number of solved instances, the number of instances where overall best cost results could be achieved, the number of obtained optimal solutions and the number of optimality proofs. Column 6 shows the number of fastest proofs. Columns 7–10 further present information for the 13 instances for which all solvers could deliver optimality proofs: The average number of nodes visited in the search process, the average runtime, the standard deviation of the number of visited nodes, and the standard deviation of the runtime.

Table 2 Overview of the final computational results based on 80 benchmark instances. Columns marked with * are based on the subset of instances for which all solvers could prove optimality.

solver	solved	best	opt	proof	fastest	avg nd*	avg rt*	std nd*	std rt*
chuffed-cp	66	25	24	14	1	3.87E+05	134.5	8.94E+05	240
chuffed-ilp	67	24	24	14	0	3.69E+05	259.8	7.75E+05	467.6
ortools-cp	72	20	20	20	0	n/a	47.2	n/a	48.5
ortools-ilp	78	20	20	20	0	n/a	12	n/a	11.9
cpopt-cp	64	37	32	14	0	3.25E+06	158.1	3.86E+06	201.1
cpopt-ilp	69	40	33	13	0	1.17E+07	487.9	1.20E+07	535.4
gurobi-cp	46	36	32	25	0	5.01E+02	30.7	4.43E+02	56.3
gurobi-ilp	65	52	37	32	23	4.74E+02	2.3	1.03E+03	3.1
gurobi-cp-ws	80	37	34	25	1	4.26E+02	26	4.04E+02	39.3
gurobi-ilp-ws	80	66	37	36	12	3.14E+02	2	5.07E+02	2.6
opl	51	22	22	19	0	1.05E+06	49.1	9.75E+05	57.3

Finding solutions. The warm-start approach for Gurobi finds solutions for all 80 instances. Even without warm-start, *ortools-ilp* was capable of finding solutions for 78 instances (72 for *ortools-cp*), followed by *cpopt-ilp* (69 instances) and *chuffed-ilp* (67 instances). Regarding the quality of found solutions, the best results were achieved by *gurobi-ilp-ws* (66 best results), followed by *gurobi-ilp* (52 best results). With *cpopt-ilp* and *cpopt-cp* as well as *gurobi-cp-ws* and *gurobi-cp*, best results could be achieved for roughly half of the instance set.

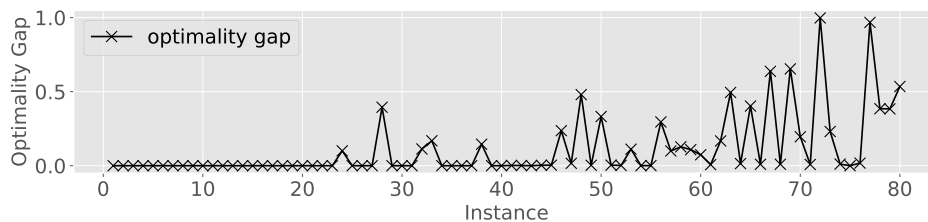
⁶ The entire benchmark set as well as the detailed experimental results and the MiniZinc code are publicly available at <https://cdlab-artis.dbai.tuwien.ac.at/papers/ovenscheduling/>.



■ **Figure 2** Comparison of proof times.

Finding optimal solutions. Using all evaluated methods, optimal solutions could be found for 37 instances: all instances with 10 jobs, 14 instances with 20 jobs, 2 with 50 jobs and one with 100 jobs. Most optimality proofs were provided by Gurobi (36 proofs with the ILP-model), followed by OR-Tools (20 proofs) and OPL (19 proofs). Even though cpopt-ilp (cpopt-cp) could only provide 13 (14) optimality proofs, it did find 33 (32) optimal solutions; chuffed-cp (chuffed-ilp) provided 14 optimality proofs and could find 24 optimal solutions. Figure 2 takes a closer look at the 13 instances for which all evaluated methods provided optimal solutions within the runtime limit (instances 1–7, 9, 10, 12, 15, 17 and 19) and compares the relative proof times for these instances. To calculate the relative proof time for an instance, we divide the absolute proof time by the overall fastest absolute proof time for that instance. Gurobi and OR-Tools deliver fastest proofs and outperform Chuffed and CP Optimizer; the OPL model lies in between. Moreover, it can be noted that Gurobi and OR-Tools perform best with the ILP-model, whereas Chuffed and CP Optimizer can provide faster proofs with the CP model.

Optimality gap. Figure 3 visualizes the overall smallest optimality gap per instance. That is, if $s(I)$ is the objective value of the overall best solution found (i.e., the minimal solution cost) and $b(I)$ is the best (i.e. maximal) dual bound found by Gurobi for instance I , the optimality gap is given by $g(I) = 1 - b(I)/s(I)$. The optimality gap generally increases with



■ **Figure 3** Overall smallest optimality gap per instance.

the number of jobs per instance: while the dual bounds are tight for all instances with 10 jobs and for most instances with 20 jobs, this is no longer the case for instances with 50 or 100 jobs. However, the size of the optimality gap is not purely determined by the number of jobs; there are 15 instances with 50 jobs or more for which the optimality gap is less than 1%.

Search strategies. The results of the comparison of search strategies for chuffed-cp, chuffed-ilp, ortools-cp and ortools-ilp can be found in Table 3. The first column in each row denotes the evaluated search strategy and the following columns contain the respective numbers of solved instances, best solution results and optimality proofs achieved by each search strategy in comparison to all other search strategies with the same solver. We can see that

■ **Table 3** Comparison of 8 search strategies for CP- and ILP-models with Chuffed and OR-Tools.

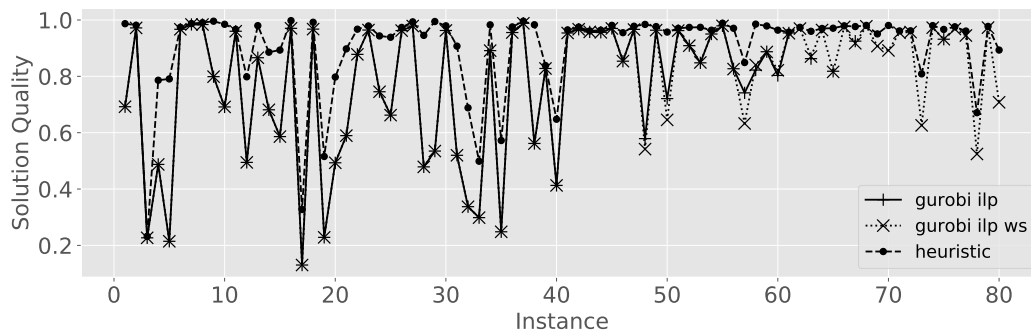
search strategy	chuffed-cp			chuffed-ilp			ortools-cp			ortools-ilp		
	solved	best	proof	solved	best	proof	solved	best	proof	solved	best	proof
default	49	22	10	41	20	10	38	26	20	21	20	20
search1	64	27	14	64	22	14	37	27	20	25	22	20
search2	64	32	14	67	25	14	71	27	19	78	31	20
search3	66	29	14	64	25	13	71	33	20	78	31	20
search4	65	30	14	64	28	13	72	30	19	78	31	20
search5	65	30	14	67	24	14	70	28	19	78	31	20
search6	64	26	14	66	26	14	72	34	20	78	31	20
search7	65	32	14	65	24	14	71	31	19	78	31	20

using search strategies could greatly improve the number of solved instances for all four compared methods. The improvement was most significant for ortools-ilp, which could solve only 21 instances with the default strategy and 78 instances with search strategies 2–7 (all instances except 77 and 80). For Chuffed, all search strategies had a comparable impact on the number of solved instances. The number of best solutions found (in comparison to the other search/model configurations with the same solver) could be improved as well for all four methods; again, the most notable improvement was for ortools-ilp and search strategies 2–7 (20 vs. 31 best solutions). For Chuffed, the number of provided optimality proofs could also be improved using the suggested search strategies. Experiments were also run for gurobi-ilp and gurobi-cp with all search strategies, but no significant differences could be observed for this solver.

Warm-start with Gurobi. The construction heuristic could find feasible solutions for all 80 instances within few seconds. Warm-starting Gurobi with these initial solutions obtained bounds for all instances. Figure 4 allows a comparison of the solution quality per instance for the construction heuristic, gurobi-ilp and gurobi-ilp-ws. The heuristic solution could be improved by gurobi-ilp-ws for all 80 instances, thus providing 15 more results than gurobi-ilp. For 12 instances for which gurobi-ilp had previously found solutions, warm-starting could improve the solution quality. For 6 instances warm-starting led to a lower solution quality. Concerning optimality proofs, gurobi-ilp-ws was slightly worse than gurobi-ilp (32 vs. 36 proofs). To sum up, even though warm-starting found solutions that were better than those found by the heuristic, using this approach was not always advantageous.

7 Conclusion

In this paper, we introduced and formally defined the Oven Scheduling Problem and provide new instances for this problem. We propose CP- and ILP-models and investigate various search strategies. Using our models as well as a warm-start approach, we were able to find feasible solutions for all 80 benchmark instances. Provably optimal solutions could be found for nearly half of the instance set and for 50 of the 80 instances, the best optimality gap is



■ **Figure 4** Solution quality per instance using the construction heuristic, gurobi-ilp without warm-start and gurobi-ilp-ws with the heuristically constructed solution for warm-start.

less than 1%. Overall, the best results could be achieved with Gurobi and OR-Tools for the ILP-model. Varying the search strategy had a major impact on the performance of the CP solvers Chuffed and Gurobi. To further improve the solution quality for large instances, we plan to develop meta-heuristic strategies based on local search or large neighborhood search. Moreover, in order to explain which parameters cause instances to be hard, an in-depth instance space analysis could be conducted.

References

- 1 Meral Azizoglu and Scott Webster. Scheduling a batch processing machine with incompatible job families. *Computers & Industrial Engineering*, 39(3-4):325–335, 2001.
- 2 Peter Brucker, Andrei Gladky, Han Hoogeveen, Mikhail Y Kovalyov, Chris N Potts, Thomas Tautenhahn, and Steef L Van De Velde. Scheduling a batching machine. *Journal of scheduling*, 1(1):31–54, 1998.
- 3 Eray Cakici, Scott J Mason, John W Fowler, and H Neil Geismar. Batch scheduling on parallel machines with dynamic job arrivals and incompatible job families. *International Journal of Production Research*, 51(8):2462–2477, 2013.
- 4 Bayi Cheng, Qi Wang, Shanlin Yang, and Xiaoxuan Hu. An improved ant colony optimization for scheduling identical parallel batching machines with arbitrary job sizes. *Applied Soft Computing*, 13(2):765–772, 2013.
- 5 Antonio Costa, Fulvio Antonio Cappadonna, and Sergio Fichera. A novel genetic algorithm for the hybrid flow shop scheduling with parallel batching and eligibility constraints. *The International Journal of Advanced Manufacturing Technology*, 75(5-8):833–847, 2014.
- 6 Purushothaman Damodaran, Mario C Vélez-Gallego, and Jairo Maya. A grasp approach for makespan minimization on parallel batch processing machines. *Journal of Intelligent Manufacturing*, 22(5):767–777, 2011.
- 7 Purushothaman Damodaran and Mario C. Vélez-Gallego. A simulated annealing algorithm to minimize makespan of parallel batch processing machines with unequal job ready times. *Expert Systems with Applications*, 39(1):1451–1458, 2012.
- 8 Ronald L Graham, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Annals of discrete mathematics*, volume 5, pages 287–326. Elsevier, 1979.
- 9 IBM. *IBM ILOG CPLEX Optimization studio, Getting Started with Scheduling in CPLEX Studio*, 2017.
- 10 Sebastian Kosch and J Christopher Beck. A new mip model for parallel-batch scheduling with non-identical job sizes. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 55–70. Springer, 2014.

- 11 Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vili m. IBM ILOG CP optimizer for scheduling. *Constraints*, 23(2):210–250, 2018.
- 12 Chung-Yee Lee, Reha Uzsoy, and Louis A Martin-Vega. Efficient algorithms for scheduling semiconductor burn-in operations. *Operations Research*, 40(4):764–775, 1992.
- 13 Arnaud Malapert, Christelle Gu ret, and Louis-Martin Rousseau. A constraint programming approach for a batch processing problem with non-identical job sizes. *European Journal of Operational Research*, 221(3):533–545, 2012.
- 14 Sujay Malve and Reha Uzsoy. A genetic algorithm for minimizing maximum lateness on parallel identical batch processing machines with dynamic job arrivals and incompatible job families. *Computers & Operations Research*, 34(10):3016–3028, 2007.
- 15 Muthu Mathirajan and Appa Iyer Sivakumar. A literature review, classification and simple meta-analysis on scheduling of batch processors in semiconductor. *The International Journal of Advanced Manufacturing Technology*, 29(9-10):990–1001, 2006.
- 16 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In Christian Bess  re, editor, *Principles and Practice of Constraint Programming – CP 2007*, Lecture Notes in Computer Science, pages 529–543, Berlin, Heidelberg, 2007. Springer.
- 17 N Rafiee Parsa, Behrooz Karimi, and A Husseinazadeh Kashan. A branch and price algorithm to minimize makespan on a single batch processing machine with non-identical job sizes. *Computers & Operations Research*, 37(10):1720–1730, 2010.
- 18 Sergey Polyakovskiy, Dhananjay Thiruvady, and Rym M’Hallah. Just-in-time batch scheduling subject to batch size. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, GECCO ’20, pages 228–235, New York, NY, USA, 2020. Association for Computing Machinery.
- 19 Chris N. Potts and Mikhail Y. Kovalyov. Scheduling with batching: A review. *European Journal of Operational Research*, 120(2):228–249, 2000.
- 20 Tanya Y. Tang and J. Christopher Beck. CP and Hybrid Models for Two-Stage Batching and Scheduling. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Lecture Notes in Computer Science, pages 431–446, 2020.
- 21 Renan Spencer Trindade, Olinto CB de Ara  jo, and Marcia Fampa. Arc-flow approach for parallel batch processing machine scheduling with non-identical job sizes. In *International Symposium on Combinatorial Optimization*, pages 179–190. Springer, 2020.
- 22 Reha Uzsoy. Scheduling a single batch processing machine with non-identical job sizes. *The International Journal of Production Research*, 32(7):1615–1635, 1994.
- 23 Pascal Van Hentenryck. *The OPL optimization programming language*. MIT press, 1999.
- 24 Mario Cesar Velez Gallego. *Algorithms for scheduling parallel batch processing machines with non-identical job ready times*. PhD thesis, Florida International University, 2009.
- 25 Z. Zhao, S. Liu, M. Zhou, X. Guo, and L. Qi. Decomposition Method for New Single-Machine Scheduling Problems From Steel Production Systems. *IEEE Transactions on Automation Science and Engineering*, 17(3):1376–1387, 2020.
- 26 Hongming Zhou, Jihong Pang, Ping-Kuo Chen, and Fuh-Der Chou. A modified particle swarm optimization algorithm for a batch-processing machine scheduling problem with arbitrary release times and non-identical job sizes. *Computers & Industrial Engineering*, 123:67–81, 2018.

A Alternative CP model using OPL

In addition to the solver independent models presented in sections 3 and 4.1, we developed an alternative CP model for IBM ILOG CPLEX Studio, since CP Optimizer is particularly well suited for scheduling problems [11]. This model is written using the Optimization Programming Language (OPL) [23] and makes use of *interval variables* for batches and

37:18 Minimizing Batch Processing Time for an Oven Scheduling Problem

setup times between batches. It is based on the ILP model described in Section 4.1.⁷ In the following, we briefly describe the decision variables and constraints that differ from the ILP model;⁸ for an introduction to the used CP Optimizer concepts see [9]. We use optional interval variables for batches:

interval $B_{m,b}$ optional $\subseteq [0, l]$ size $\in [min_T, max_T]$ intensity $av_m \quad \forall m \in \mathcal{M} \forall b \in [n]$.

Batches are optional since not all $k \times n$ batches will actually be used: Depending on whether any jobs are assigned to a batch or not, the batch interval variable will be present or absent. The intensity function av_m encodes the machine availability times and is modeled using *intensity step functions*; $av_m(t) = 100$ if machine m is available at time t and $av_m(t) = 0$ otherwise. Setup times between batches are also modelled using optional interval variables:

interval $st_{m,b}$ optional $\subseteq [0, l]$ size $\in [0, max_{ST}]$ intensity $av_m \quad \forall m \in \mathcal{M} \forall b \in [n - 1]$.

Besides these interval variables, we use the same decision variables as in Section 4.1: $X_{m,b,j} \in \{0, 1\}$ to encode whether job j is assigned to batch $B_{m,b}$, $A_{m,b} \in [0, a]$ for the attribute of batch $B_{m,b}$ and $sc_{m,b}$ for setup costs.

The following constraints are used for the batch and setup time interval variables:

- **presenceOf** ensures that a batch is present iff some job is assigned to it. Similarly, setup times are present iff the preceding and following batch are present.
- **endBeforeStart** is used to enforce the order of batches and setup times on the same machine and **endAtStart** is used to schedule setup times exactly before the following batch.
- **startOf** restricts the start time of batches to be after the earliest start date of any assigned job and **lengthOf** is used to enforce that batch processing times lie between the minimal and maximal processing time for every assigned job
- **noOverlap** is used as a redundant constraint to ensure that batches on the same machine do not overlap
- **forbidExtent** is used to guarantee that batches and setup times are scheduled entirely within one machine availability interval: whenever $B_{m,b}$ or $st_{m,b}$ is present, it cannot overlap a point t where $av_m(t) = 0$.

⁷ We based the OPL model on our ILP model as it turned out that using CP Optimizer as solver via MiniZinc delivers particularly good results with the ILP model, see the results in Section 6.

⁸ The full model will be made available on our website once this paper has been accepted for publication.

Combining Clause Learning and Branch and Bound for MaxSAT

Chu-Min Li ✉

Huazhong University of Science and Technology, Wuhan, China

Université de Picardie Jules Verne, Amiens, France

Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

Zhenxing Xu ✉

Huazhong University of Science and Technology, Wuhan, China

Jordi Coll ✉

Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

Felip Manyà ✉

Artificial Intelligence Research Institute, CSIC, Bellaterra, Spain

Djamal Habet ✉

Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

Kun He ✉

Huazhong University of Science and Technology, Wuhan, China

Abstract

Branch and Bound (BnB) is a powerful technique that has been successfully used to solve many combinatorial optimization problems. However, MaxSAT is a notorious exception because BnB MaxSAT solvers perform poorly on many instances encoding interesting real-world and academic optimization problems. This has formed a prevailing opinion in the community stating that BnB is not so useful for MaxSAT, except for random and some special crafted instances. In fact, there has been no advance allowing to significantly speed up BnB MaxSAT solvers in the past few years, as illustrated by the absence of BnB solvers in the annual MaxSAT Evaluation since 2017. Our work aims to change this situation and proposes a new BnB MaxSAT solver, called MaxCDCL, by combining clause learning and an efficient bounding procedure. The experimental results show that, contrary to the prevailing opinion, BnB can be competitive for MaxSAT. MaxCDCL is ranked among the top 5 solvers of the 15 solvers that participated in the 2020 MaxSAT Evaluation, solving a number of instances that other solvers cannot solve. Furthermore, MaxCDCL, when combined with the best existing solvers, solves the highest number of instances of the MaxSAT Evaluations.

2012 ACM Subject Classification Software and its engineering → Constraints

Keywords and phrases MaxSAT, Branch&Bound, CDCL

Digital Object Identifier 10.4230/LIPIcs.CP.2021.38

Supplementary Material *Software (Source Code)*: <https://home.mis.u-picardie.fr/~cli/EnglishPage.html>

Funding This work has been partially funded by the French Agence Nationale de la Recherche, reference ANR-19-CHIA-0013-01, and the Spanish AEI project PID2019-111544GB-C2.

Acknowledgements This work has been partially supported by Archimedes Institute, Aix-Marseille University. We thank the anonymous reviewers for their comments and suggestions that helped to improve the manuscript.

1 Introduction

The Maximum satisfiability problem (MaxSAT) is an optimization version of a well-studied and canonical NP-Complete problem, the satisfiability problem (SAT). Although SAT and MaxSAT share many aspects, solving MaxSAT is much harder than solving SAT in practice.



© Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 38; pp. 38:1–38:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Indeed, since several clauses can be falsified in an optimal MaxSAT solution, some fundamental SAT techniques such as unit propagation cannot be used in MaxSAT as they are used in SAT. Despite this difficulty, huge efforts made by researchers make it possible nowadays to solve many interesting real-world and academic NP-hard optimization problems encoded as MaxSAT instances [11, 26]. For this reason, MaxSAT has attracted increasing interest from the academy and industry in recent years.

As for many NP-hard problems, algorithms for MaxSAT are divided into two categories: exact algorithms, which return optimal solutions and prove their optimality; and heuristic algorithms, which quickly find solutions of good quality without guaranteeing their optimality. This paper will focus on exact algorithms for (unweighted) MaxSAT.

We roughly distinguish two types of exact algorithms for MaxSAT: branch-and-bound (BnB) algorithms [26], which directly tackle MaxSAT with a bounding procedure, but without unit propagation and clause learning; and SAT-based algorithms [11], which transform MaxSAT into a sequence of SAT instances and call a CDCL (Conflict-Driven Clause Learning) SAT solver to solve them. The performance of SAT-based MaxSAT algorithms is usually much better than BnB MaxSAT solvers in solving many real-world NP-hard optimization problems, because they indirectly exploit clause learning via the SAT solver. Unfortunately, it is hard for a BnB solver to exploit clause learning. In a CDCL SAT solver, a backtracking happens only when a clause is falsified, from which a sequence of resolution steps is performed to learn a clause explaining the backtracking. However, a BnB MaxSAT solver also need to backtrack when it computes a lower bound equal to the upper bound. In this case, no clause is explicitly falsified, making it hard to learn a clause. Probably because of this difficulty, there has been no advance allowing to significantly speed up BnB MaxSAT solvers in recent years, as illustrated by their absence in the annual MaxSAT Evaluation since 2017.

In this paper, we propose an original approach that allows a BnB MaxSAT solver to learn a clause when it computes a lower bound equal to the upper bound, together with a new bounding procedure, because the one in current BnB MaxSAT solvers is not adequate for large instances. This approach is implemented in a new BnB MaxSAT solver, called MaxCDCL, that combines the new bounding procedure and clause learning. The experimental results show that MaxCDCL is ranked among the top 5 solvers of the 15 solvers that participated in the 2020 MaxSAT Evaluation, solving a number of instances that other solvers cannot solve. Furthermore, MaxCDCL, when combined with the best existing solvers, solves the highest number of instances of the MaxSAT Evaluations.

Combining clause learning and BnB, as clause learning itself, belongs to the general framework consisting in explaining a failure in the search to avoid the same failure in the future. In other fields such as Pseudo-Boolean Optimization (PBO), there are also works in this framework (e.g. [17]). The general framework is not hard to understand. However, making it effective for solving a particular problem such as SAT or MaxSAT is quite challenging, because this requires a deep understanding of the problem and the related solving techniques. So, one important contribution of our work is that we found a configuration and an efficient implementation of this configuration allowing to make the combination of clause learning and BnB effective for MaxSAT, as presented in this paper.

More importantly, our results refute a prevailing opinion in the field stating that, although BnB is a powerful technique that has successfully been used to solve many combinatorial optimization problems, it is not so useful for MaxSAT. Indeed, as the first BnB MaxSAT solver successfully exploiting clause learning, MaxCDCL opens promising research directions.

This paper is organized as follows: Section 2 presents the preliminaries. Section 3 reviews state-of-the-art MaxSAT solvers. Section 4 describes MaxCDCL. Section 5 empirically evaluates and analyzes MaxCDCL. Section 6 concludes.

2 Preliminaries

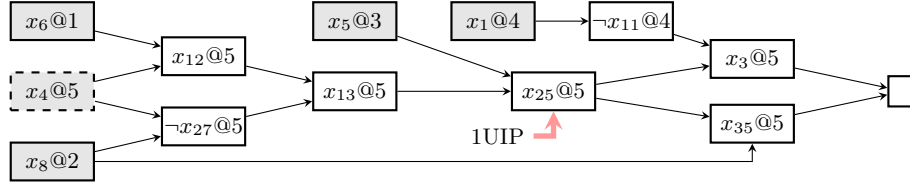
A propositional *variable* x can take values 0 or 1 (false or true). A *literal* is a variable x or its negation $\neg x$. A *clause* is a disjunction of k literals $l_1 \vee \dots \vee l_k$. A *propositional formula in Conjunctive Normal Form* (CNF) is a conjunction (or a set) of m clauses c_1, \dots, c_m . An *assignment* of truth values to the propositional variables satisfies a literal x if $x = 1$, and satisfies a literal $\neg x$ if $x = 0$. A literal x or $\neg x$ is *assigned* if x is assigned a value, otherwise it is *free*. An assignment is *complete* if all the variables are assigned a value, otherwise it is *partial*. A (partial) assignment is usually represented by a (sub)set of satisfied literals. A clause is satisfied if at least one of its literals is satisfied. A CNF is satisfied if all its clauses are satisfied. The *Boolean satisfiability* (SAT) problem for a CNF ϕ is to determine whether there exists an assignment that satisfies ϕ .

MaxSAT is the problem of finding an assignment that satisfies the maximum number of clauses in a given multiset of clauses. In partial MaxSAT, there are *hard* and *soft* clauses, and the goal is to satisfy all the hard clauses and the maximum number of soft clauses. In weighted (partial) MaxSAT, each soft clause has a cost to pay if it is violated. This paper will focus exclusively on unweighted (partial) MaxSAT.

The state-of-the-art SAT solvers implement the CDCL algorithm. CDCL alternates a search phase, where literals are assigned until either a solution or a conflict is found, and a learning phase, which is executed after finding a conflict in order to learn a new clause. *Unit Propagation* (UP) is the main inference rule applied during the search: If there is a unit clause $\{l\}$ in ϕ , literal l must be satisfied (i.e., set to 1). Then, any clause containing l is removed from ϕ , and all the occurrences of $\neg l$ in clauses of ϕ are (implicitly) removed. UP is applied during the search until an empty clause (*conflict*) is found or no unit clause exists in ϕ . If UP finishes without finding a conflict, a new literal is picked following a heuristic and is set to 1 (we make a *decision*), and UP is applied again. If all the variables are assigned without finding a conflict, ϕ is satisfiable. The *decision level* of an assigned literal is the number of decisions made before being assigned. When a conflict is found, a *conflict analysis* is performed on the *implication graph* to derive a new clause explaining the conflict.

Figure 1 shows an example of implication graph. It is a directed acyclic graph where each node represents an assignment $l@dl$, where l is a literal set to 1 and dl is its decision level. The negations of the literals of incoming edges of a node $l@dl$ represent the reason (clause) why UP has set $l = 1$. For instance, node $x_{12}@5$ is propagated due to clause $\neg x_6 \vee \neg x_4 \vee x_{12}$ (i.e., $x_6 \wedge x_4 \rightarrow x_{12}$), given that x_4 and x_6 have been set to 1. A node without incoming edges represents a decision. All decisions are painted in grey, and the last one is dashed. A conflicting clause is represented by incoming edges to \square ; in this example, clause $\neg x_3 \vee \neg x_{35}$. A *Unique Implication Point* (UIP) is a node of the implication graph that belongs to all paths from the last decision to the conflict. Figure 1 contains three UIPs: the last decision x_4 , and literals x_{13} and x_{25} . The most used learning schema in CDCL SAT solvers is called *first UIP (1UIP)*, guided by the closest UIP to the conflict in the implication graph.

When a conflict is found in the decision level dl (i.e., dl decisions were made before the conflict is found), a node in the decision level dl is said *active* if it is not the 1UIP but is in a path from the 1UIP to the conflict. In other words, the active nodes are those nodes that allow to reach the conflict from the 1UIP in the decision level dl . For example, in Figure 1, a conflict is found in the decision level 5, and $x_3@5$ and $x_{35}@5$ are active nodes. The 1UIP learning schema identifies, in each path from a node in a decision level lower than dl to an active node, the last literal in the lower decision level. The new learnt clause is composed of



■ **Figure 1** Example of implication graph.

the negation of these literals and the 1UIP. For instance, in Figure 1, clause $\neg x_8 \vee x_{11} \vee \neg x_{25}$ is learnt, because there are two paths from lower decision levels to an active node, in which x_8 and $\neg x_{11}$ are the last literals in lower decision levels, respectively.

The learnt clause explains the conflict: when all its literals are falsified, unit propagation reproduces the implication graph to derive the same conflict. So, adding the learnt clause prevents the same conflict in the future search. After learning a clause, CDCL backtracks to the second highest decision level of the learnt clause (level 4 in the above example). Unsatisfiability is determined when a conflict is found at decision level 0.

3 Related Work

A major difference between MaxSAT and SAT solvers is that each clause must be satisfied in a SAT solution while a soft clause can be falsified in an optimal MaxSAT solution, making MaxSAT much harder to solve than SAT in practice. Despite this, the MaxSAT community has made huge efforts to implement exact MaxSAT solvers with impressive performance over the last decade [11, 26]. On the other hand, heuristic MaxSAT algorithms such as SatLike [25] have also been proposed.

Roughly speaking, we find two main groups of exact MaxSAT solvers: branch-and-bound (BnB) and SAT-based solvers. BnB MaxSAT solvers implement the branch-and-bound scheme and incorporate a lookahead procedure that detects inconsistent subsets of soft clauses by applying unit propagation and computes a lower bound [26]. They also apply some inference rules at each node of the search tree. Representative BnB solvers are MaxSatz [30], MiniMaxSat [20], Ahmaxsat [1, 14] and Akmaxsat [24]. Closely related to MaxSAT, we can find BnB solvers for the Weighted Constraint Satisfaction Problem (WCSP). Recently, it was presented a technique to improve BnB WCSP solving by avoiding branching on variables which are unlikely to increase the lower bound [43].

SAT-based MaxSAT solvers proceed by reformulating the MaxSAT optimization problem into a sequence of SAT decision problems [11]. These solvers could still be divided into three subgroups: model-guided, core-guided and Minimum Hitting Sets (MHS)-guided. Model-guided approaches reduce to SAT the problem of deciding whether there exists an assignment for the MaxSAT instance with a cost less than or equal to a certain k , and successively decrease k until an unsatisfiable SAT instance is found. Among such solvers we find SAT4J-Maxsat [12], QMaxSat [23, 44], Open-WBO [37] or Pacose [40]. Core-guided and MHS-guided approaches consider a MaxSAT instance as a SAT instance and call a CDCL SAT solver to identify an unsatisfiable subset of soft clauses, called a *core*. Then, they relax this core and solve the relaxed instance with a CDCL SAT solver to identify another core, repeating this process until deriving a satisfiable instance. The difference between them is that core-guided solvers relax a core using cardinality constraints, while MHS-guided solvers remove one clause from each detected core so that the number of different clauses removed from the cores is minimized by solving a minimum hitting set instance with an

integer programming solver. The most representative core-guided solvers include msu1.2 [36], WBO [35], Open-WBO [37], WPM1 [3], PM2 [5], WPM2 [4], WPM3 [6], Eva [39], RC2 [21], and the most representative MHS-guided solvers include MHS [41] and MaxHS [8, 10, 15, 16]. Core-guided search has also been extended to constraint programming [19].

A common point of SAT-based MaxSAT solvers is that they *indirectly* exploit the clause learning technique by repeatedly calling a CDCL SAT solver. Unfortunately, it is hard for BnB solvers to exploit clause learning, which might explain their poor performance on real-world optimization problems. We are aware of only one tentative in [2]: when the number of falsified soft clauses reaches the upper bound, the falsification of these soft clauses is analyzed to learn a clause. Nevertheless, no clause is learnt when the lookahead procedure returns a lower bound equal to the upper bound, and the reported results are not competitive.

4 MaxCDCL: A BnB Algorithm Using CDCL for MaxSAT

This section first presents the general structure of MaxCDCL, and then the different components of our approach implemented in MaxCDCL.

4.1 General Structure of MaxCDCL

We distinguish between hard and soft conflicts in the MaxSAT context. A *hard* conflict occurs when the current partial assignment falsifies a hard clause. Given an upper bound UB, a *soft* conflict occurs when the current partial assignment cannot be extended to a complete one falsifying fewer than UB soft clauses. A CDCL SAT solver only considers hard conflicts, learns a hard clause from each hard conflict and backtracks. A BnB CDCL MaxSAT solver extends the CDCL SAT solver by also learning a hard clause from each discovered soft conflict and backtracks. Algorithm 1 depicts such a BnB CDCL MaxSAT solver called MaxCDCL.

Given a MaxSAT instance with a set of hard clauses and a set of soft clauses, MaxCDCL works with H and S , where H contains the hard clauses and S contains a new literal y for each soft clause sc after adding the hard clauses encoding $y \leftrightarrow sc$ to H . We call such y *soft literal*, because it represents a soft clause (i.e., a soft literal is satisfied if and only if the corresponding soft clause is satisfied). To solve the MaxSAT instance, MaxCDCL is repeatedly called with $UB = 2^0, 2^1, 2^2, 2^3, \dots$ without exceeding $|S| + 1$ or $UB_f + 1$ where UB_f is a feasible upper bound computed by a heuristic solver with a short cutoff. This process stops when it obtains an assignment satisfying all clauses in H and falsifying fewer than UB soft literals in S . After that, MaxCDCL is repeatedly called with UB set to the number of falsified soft literals in the previous call, until no better solution can be found.

MaxCDCL is like a CDCL SAT solver except for two points. First, when UP does not falsify any hard clause in the UPLA procedure, it calls, under certain condition, a lookahead (LA) procedure to compute a lower bound $|cores|$ of the number of soft literals that will be falsified if the current partial assignment is extended. If a soft conflict is discovered, i.e., if $|falseS| + |cores| \geq UB$, it is analyzed to learn a clause for backtracking. Second, if no soft conflict is discovered, but it is discovered that the falsification of any free soft literal y not used in $|cores|$ would result in a soft conflict, y is satisfied by a procedure called *hardening*.

Note that model-guided MaxSAT solvers also set a UB in their search. However, they do not compute a lower bound to discover soft conflicts as MaxCDCL. Consequently, MaxCDCL is able to backtrack much earlier than model-guided MaxSAT solvers for a given UB. See the next subsection for details and illustrative examples.

■ **Algorithm 1** MaxCDCL(H, S, UB), a generic CDCL procedure with lookahead for MaxSAT.

Input: H : a set of hard clauses, S : a set of soft literals, UB : an upper bound.
Output: $|falseS|$, where $falseS$ is the set of falsified soft literals, if $|falseS| < UB$;
or UB otherwise

```

1 begin
2   while true do
3     currentLevel  $\leftarrow$  0; /* start or restart search */
4     while true do
5       ( $cl, falseS, cores, reasons$ )  $\leftarrow$  UPLA( $H, S, UB, currentLevel$ );
6       if  $cl$  is a falsified hard clause or  $|falseS| + |cores| \geq UB$  then
7         if currentLevel = 0 then
8           return UB;
9         else
10          newLearntClause  $\leftarrow$  analyze( $cl, falseS, reasons$ );
11          level  $\leftarrow$  the second highest level in newLearntClause;
12          backtrackTo(level);
13          currentLevel  $\leftarrow$  level;
14       else
15         if all variables are assigned then
16           return  $|falseS|$ ;
17         else if  $|falseS| + |cores| = UB - 1$  then
18           hardening();
19         else if restart condition is satisfied then
20           backtrackTo(0);
21           break; /* restart */
22         else
23           currentLevel++;
24            $H \leftarrow H \cup \{l\}$  where  $l$  is a free literal selected using a heuristic;

```

4.2 Combining Lookahead and Clause Learning

A subset of soft literals $S_i = \{y_1, \dots, y_{|S_i|}\}$ is inconsistent if they cannot be simultaneously satisfied. This inconsistency can be represented by the hard clause $\neg y_1 \vee \dots \vee \neg y_{|S_i|}$. Note that $|S_i|$ can be 1. If the inconsistency is independent of any partial assignment, the subset is called a *global core*. Otherwise, the inconsistency is implied by a subset of literals and the inconsistent subset of soft literals is called a *local core*. The core-guided or MHS-guided SAT-based MaxSAT solvers only detect global cores to relax them, while our approach detects local cores, given a partial assignment, to discover a soft conflict.

► **Example 1.** Let $H = \{\neg y_1 \vee x_1 \vee \neg x_2, \neg x_1 \vee \neg x_3 \vee \neg x_4, \neg y_2 \vee x_3, \neg y_3 \vee x_5\}$, where y_1, y_2 and y_3 are soft literals. If no variable is assigned, all soft literals can be simultaneously satisfied. So, no global core exists. However, if the current partial assignment is $\{x_2 = 1, x_4 = 1\}$, the subset of soft literals $\{y_1, y_2\}$ is a local core implied by the partial assignment. We write the implication by $H \cup \{x_2, x_4\} \rightarrow \neg y_1 \vee \neg y_2$.

Proposition 2 provides the foundation of our approach in the general case.

► **Proposition 2.** *Let H be a set of hard clauses, $S = \{y_1, \dots, y_{|S|}\}$ be the set of all soft literals, k be an integer, and $L_i = \{l_{i1}, \dots, l_{i|L_i|}\}$ ($1 \leq i \leq k$) be a set of literals. If, for every i ($1 \leq i \leq k$), $H \cup L_i$ implies a local core $S_i = \{z_{i1}, \dots, z_{i|S_i|}\} \subset S$ (i.e., $H \cup L_i \rightarrow \neg z_{i1} \vee \dots \vee \neg z_{i|S_i|}$), and S_i and S_j are disjoint for any $j \neq i$ such that $1 \leq j \leq k$, then every assignment that satisfies $H \cup \{\neg y_1 + \dots + \neg y_{|S|} < k\}$ also satisfies the clause $\neg l_{11} \vee \dots \vee \neg l_{1|L_1|} \vee \dots \vee \neg l_{k1} \vee \dots \vee \neg l_{k|L_k|}$.*

Proof. It is easy to see that $H \cup L_1 \cup \dots \cup L_k$ implies $(\neg z_{11} \vee \dots \vee \neg z_{1|S_1|}) \wedge \dots \wedge (\neg z_{k1} \vee \dots \vee \neg z_{k|S_k|})$, meaning that $H \cup L_1 \cup \dots \cup L_k$ falsifies the constraint $\neg y_1 + \dots + \neg y_{|S|} < k$, because each clause $\neg z_{i1} \vee \dots \vee \neg z_{i|S_i|}$ implies at least one different falsified soft literal. Hence, any assignment satisfying $H \cup \{\neg y_1 + \dots + \neg y_{|S|} < k\}$ must falsify at least one literal in $L_1 \cup \dots \cup L_k$, and satisfy the clause $\neg l_{11} \vee \dots \vee \neg l_{1|L_1|} \vee \dots \vee \neg l_{k1} \vee \dots \vee \neg l_{k|L_k|}$. Note that we use “z” (instead “y”) to denote a soft literal in a local core to avoid complex subscripts. ◀

Given a partial assignment F of H , the application of Proposition 2 consists in first detecting a local core S_i and then identifying the smallest $L_i \subset F$ such that $H \cup L_i$ implies S_i . We call L_i the *reason* of S_i . If k is the current upper bound UB and k disjoint local cores are detected, a soft conflict is discovered, and the clause $\neg l_{11} \vee \dots \vee \neg l_{1|L_1|} \vee \dots \vee \neg l_{k1} \vee \dots \vee \neg l_{k|L_k|}$, which is implied by $H \cup \{\neg y_1 + \dots + \neg y_{|S|} < k\}$ and is falsified by the current partial assignment, can be considered by an implicit clause explaining the soft conflict. This clause can be further analyzed using the 1UIP schema in line 10 of Algorithm 1 to learn a clause to be explicitly added to H to prevent the same soft conflict in the future as in the hard conflict case.

The detection of a local core S_i is implemented by using UP in a lookahead procedure as in existing BnB MaxSAT solvers [27, 28, 29]. The advantage of this procedure is that S_i is minimal w.r.t. UP, in the sense that UP cannot detect any local core that is a proper subset of S_i under the same partial assignment [27], which is essential for our approach, because MaxCDCL needs to learn clauses of good quality from the detected local cores. Recall that BnB MaxSAT solvers detect disjoint local cores but do not explain them. When a soft conflict is discovered, they simply backtrack without learning a hard clause from the soft conflict, which is very different from MaxCDCL.

Concretely, MaxCDCL calls Algorithm 2 at decision level dl with a partial assignment F , under which the already falsified soft literals are stored in a set named *falseS*. The lookahead procedure starts at line 5 in Algorithm 2 when no hard conflict is found, and terminates at line 10 or line 24. This procedure proceeds in decision level $dl+1$ by maintaining a set of detected local cores (*cores*). Every iteration of the loop (line 8) propagates a free soft literal y not occurring in *cores*, until a clause h in H is falsified or a soft literal sl not occurring in *cores* is falsified (line 13). These unit propagations construct an implication graph G . The propagated free soft literals y_1, y_2, \dots in the loop can be seen as temporary assumptions at decision level $dl+1$ that have no incoming edge in G .

Inspecting G , let z_1, \dots, z_b be the subset of literals y_1, y_2, \dots at level $dl+1$ and without incoming edge, from which there is a path to h or to $\neg sl$. Then, $\{z_1, \dots, z_b\}$ (resp. $\{z_1, \dots, z_b, sl\}$) is a local core S_i implied by the partial assignment F at decision level dl . Based on G , we can also define L_i as the set containing, from each path to h (resp. $\neg sl$), the last literal assigned before level $dl+1$. Clearly, $H \cup L_i$ implies $\neg z_1 \vee \dots \vee \neg z_b$ (resp. $\neg z_1 \vee \dots \vee \neg z_b \vee \neg sl$).

Note that each already falsified soft literal y in *falseS* constitutes a local core $S_i = \{y\}$ not in *cores*, implied by $H \cup \{\neg y\}$ (i.e. $L_i = \{\neg y\}$). Therefore, when $k = |falseS| + |cores| = \text{UB}$ in MaxCDCL, we have UB disjoint local cores. According to Proposition 2, $\neg l_{11} \vee \dots \vee$

■ **Algorithm 2** UPLA(H, S, UB, dl), Unit propagation followed by lookahead.

Input: H : a set of hard clauses, S : a set of soft literals, UB : an upper bound, dl : the current decision level.

Output: cl : a hard clause; $falseS$: the set of falsified soft literals; $cores$: a set of disjoint local cores; $reasons$: a set of literals that are reasons of $cores$

```

1 begin
2    $(cl, falseS) \leftarrow UP(H)$ ;
3   if  $cl$  is a falsified hard clause or  $|falseS| \geq UB$  or the condition to lookahead is
   not satisfied then
4      $\text{return } (cl, falseS, \emptyset, \emptyset)$ ;
5    $H' \leftarrow H$ ;  $F \leftarrow \{\neg l \mid l \text{ is falsified in } H\}$ ;
6    $reasons \leftarrow \emptyset$ ;  $cores \leftarrow \emptyset$ ;  $S' \leftarrow \emptyset$ ;
7   Increase the decision level to  $dl + 1$ ;
8   while true do
9     if all soft literals of  $S$  either are non-free or occur in  $cores$  then
10       $\text{return } (cl, falseS, cores, reasons)$ ;
11     Let  $y$  be a free soft literal not occurring in  $cores$ ;
12      $H \leftarrow H \cup \{y\}$ ;  $S' \leftarrow S' \cup \{y\}$ ;
13      $(h, sl) \leftarrow UPforLA(H)$ ;
14     if  $h$  is a falsified hard clause or  $sl$  a falsified soft literal not in  $cores$  then
15       if  $h$  is a falsified hard clause then
16          $core \leftarrow \{l \mid l \in S' \text{ and } l \text{ has no incoming edge and}$ 
            $\text{there is a path from } l \text{ to } h \text{ in the implication graph}\}$ ;
17       else
18          $core \leftarrow \{sl\} \cup \{l \mid l \in S' \text{ and } l \text{ has no incoming edge and}$ 
            $\text{there is a path from } l \text{ to } \neg sl \text{ in the implication graph}\}$ ;
19        $reason \leftarrow \{l \mid l \in F \text{ and there is a path from } l \text{ to } h \text{ or } \neg sl$ 
            $\text{in the implication graph, and the literal next to } l \text{ in the path is}$ 
            $\text{of decision level } dl + 1\}$ ;
20        $cores \leftarrow cores \cup \{core\}$ ;
21        $reasons \leftarrow reasons \cup reason$ ;
22        $H \leftarrow H'$ ;  $S' \leftarrow \emptyset$ ; /*Cancel UP done by lookahead*/
23       if  $|cores| + |falseS| \geq UB$  then
24          $\text{return } (cl, falseS, cores, reasons)$ ;

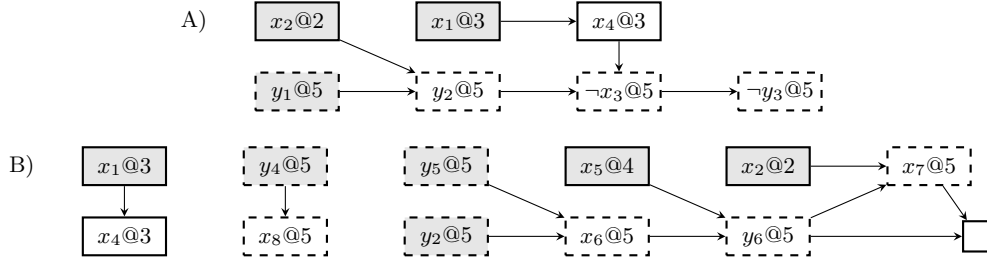
```

$\neg l_{1|L_1|} \vee \dots \vee \neg l_{k1} \vee \neg l_{k2} \vee \dots \vee \neg l_{k|L_k|}$ could be considered as an implicit clause in H , which is falsified by the current partial assignment and is analyzed using the 1UIP schema in line 10 of Algorithm 1 to learn a clause as in the hard conflict case.

► **Example 3.** Let y_1, \dots, y_7 be soft literals, and let H be formed by

$$\begin{array}{lllll}
 \neg y_1 \vee y_2 \vee \neg x_2 & \neg y_2 \vee \neg x_3 \vee \neg x_4 & \neg y_2 \vee \neg y_5 \vee x_6 & \neg x_1 \vee x_4 & \neg y_3 \vee x_3 \\
 \neg x_6 \vee \neg x_5 \vee y_6 & \neg y_6 \vee \neg x_2 \vee x_7 & \neg x_7 \vee \neg y_6 & \neg y_4 \vee x_8 & \neg y_7 \vee x_9
 \end{array}$$

If no variable is assigned, all soft literals can be simultaneously satisfied. So, no global core exists. Let the current partial assignment be $\{y_7=0, x_2=1, x_1=1, x_4=1, x_5=1\}$, where x_4 is propagated due to $\neg x_1 \vee x_4$, and the other literals have been decided. Hence $falseS = \{y_7\}$ and the current decision level is 4. Let also $UB=3$. We show how Algorithm 2 detects $cores$ and reaches UB by propagating the free soft literals y_1, \dots, y_6 , by means of Figure 2.



■ **Figure 2** Implication graphs involved in the detection of *cores* in Example 3. Grey nodes represent decisions or assumptions, and dashed nodes represent assignments made in the lookahead process.

After propagating y_1 , y_2 is satisfied and y_3 is falsified. Hence $sl=y_3$ is detected, the implication graph in Figure 2A is obtained, looking at which $core=\{y_1, y_3\}$ will be identified at line 18 with reason= $\{x_2, x_4\}$, because there are paths from x_2 and x_4 to $\neg y_3$, and the next literals to x_2 and x_4 in the paths are from decision level 5.

Note that since y_2 is not in *cores*, it can be used to detect new cores. After propagating y_2, y_4, y_5 , the implication graph of Figure 2B is obtained, and a conflicting clause $h=\{\neg x_7 \vee \neg y_6\}$ is detected. Then, $core=\{y_2, y_5\}$ is detected at line 16 with reason= $\{x_2, x_5\}$.

Since $|cores| + |falseS| = 2 + 1 = UB$, a soft conflict is found at line 23, with reasons= $\{x_2, x_4, x_5, \neg y_7\}$. According to Proposition 2, clause $\neg x_2 \vee \neg x_4 \vee \neg x_5 \vee y_7$ is implied by $H \cup \{\neg y_1 + \dots + \neg y_{|S|} < 3\}$, which is falsified at decision level 4, and hence can be analysed with the 1UIP scheme at level 4 to learn a new clause and backtrack. Note that a model-guided MaxSAT solver would not discover this soft conflict at this stage, because $UB=3$ but only one soft literal is falsified.

Proposition 2 also provides the basis for hardening. If $k = |falseS| + |cores| = UB - 1$, for each free soft literal z not in the k local cores, $H \cup \{\neg z\}$ implies a new local core $\{z\}$. The hardening procedure (line 18 in Algorithm 1) satisfies z with the reason $z \vee \neg l_{11} \vee \dots \vee \neg l_{1|L_1|} \vee \dots \vee \neg l_{k1} \vee \dots \vee \neg l_{k|L_k|}$. Note that this hardening satisfies a fundamental requirement of CDCL SAT solvers: except the decision (i.e., branching) variables, the value of every variable must be explicitly associated with a reason. In contrast, although existing BnB MaxSAT solvers also implement hardening, no reason is associated with the hardening.

Let $m = |H| + |S|$. UP is in $O(m)$ for detecting one local core. Since the lookahead procedure detects at most UB cores, its whole complexity is in $O(m \times UB)$.

4.3 A Probing Strategy for Lookahead

Existing BnB MaxSAT solvers usually tackle random or crafted instances of limited size and look ahead at each branch. However, such a treatment might be too costly and useless for large instances. If the lower bound is not tight enough to prune the current branch, the time spent to compute the lower bound is lost. When $k = UB - |falseS|$, the lookahead procedure has to detect k disjoint local cores to be successful. Generally, the greater the value of k , the lower the probability of lookahead to be successful.

MaxCDCL uses a probing strategy to determine if lookahead has to be applied at the current branch. With probability p , where p is a parameter intuitively fixed to 0.01, lookahead is applied for probing purpose. The mean *avgp* and the standard deviation *devp* of the number of detected disjoint local cores in a successful probing lookahead are computed (not shown here due to the lack of space) to select the branches where lookahead is applied.

Inspired by the 68-95-99.7 rule in statistics, which says that the values within one (two, three) standard deviation of the mean account for about 68% (95%, 99.7%) of a normal data set, we reasonably assume that the number of cores detected in a successful lookahead is probably lower than $avgp + coef * devp$ when $coef = 3$. So, lookahead is not applied at the current branch when $k > avgp + coef * devp$. However, since the probing may not get exact information and the values may not follow a perfect normal distribution, $coef$ is dynamically adjusted to maintain the success rate of lookahead between $lowRate$ and $highRate$, where $lowRate$ and $highRate$ are parameters intuitively fixed to 0.6 and 0.75, respectively.

Concretely, $coef$ is initialized to 2 for each UB. At each probing, if the success rate of lookahead since the last probing is greater than $highRate$, it is increased by 0.1; and if it is lower than $lowRate$, it is decreased by 0.1. There are no lower and upper bounds on $coef$. When it is too low so that no lookahead is performed between two probings, it is reset to 2.

4.4 Soft literal ordering in lookahead

Existing BnB MaxSAT solvers such as MaxSatz usually propagate soft unit clauses in the ordering these clauses become unit, or in their ordering in the input formula when detecting disjoint cores [28]. MaxCDCL propagates first the soft literals in the cores detected in the previous lookahead. We use this ordering for two intuitive reasons.

- Before backtracking, a local core detected in the previous lookahead remains to be a core. Re-detecting a previous local core allows to obtain a possibly smaller local core due to additional assignments.
- After backtracking, re-detecting local cores in the previous soft conflict may allow to detect a new soft conflict sharing many local cores with the previous conflict. In this way, the clauses learnt from consecutive soft conflicts allow to intensify the search, because the clause learnt from a soft conflict is derived from the reasons of the detected cores.

The quality of a clause learnt from a soft conflict highly depends on the detected cores, which in turn highly depends on the soft literal ordering. How to improve the quality of the learnt clause by further improving the soft literal ordering definitely deserves future study.

4.5 Improving the VSIDS heuristic by lookahead

When there is no unit clause, a CDCL SAT solver chooses a free literal l using a decision heuristic, satisfies l and then performs unit propagation. VSIDS [38] is one of the widely used decision heuristics: it initializes the score of each variable to 0, and then at each conflict, it increases the score of each variable in a path to the conflict in the implication graph by var_inc , where var_inc is initialized to 1 and, after each conflict, it is divided by a parameter usually set to 0.95 to give more importance to the next conflict. Note that VSIDS is a heuristic based on lookback, because it is based on the conflicts in the past.

VSIDS is also used in MaxCDCL by increasing the score of a variable in a soft or hard conflict as in a CDCL SAT solver, but is modified as follows by taking lookahead into account. Every time the lookahead procedure detects a local core, the score of the variables encountered when identifying the reason of the core (see Subsection 4.2) is increased by $var_inc \times \gamma$, where γ is a discount-rate parameter as in reinforcement learning, and is empirically fixed to 0.1. The intuition of this modification is the following. A soft conflict is derived when UB local cores are detected. So a variable contributing to many local cores should be favoured to reach a soft conflict as early as possible. However, a local core represents only a component of a future possible soft conflict but not a soft conflict for sure. So the increase of the score of a variable contributing to a core should be discounted by γ .

4.6 Implementation of MaxCDCL

Since MaxCDCL can be considered as an extension of a CDCL SAT solver, it is implemented on top of the CDCL SAT solver MapleCOMSPS_LRB [32], winner of the application track of SAT competition 2016. We chose MapleCOMSPS_LRB because it was one of the best SAT solvers with deterministic behavior when we started this work in 2017. Nevertheless, some of the recent advances in SAT solving are incorporated, including¹:

- The approach from [31, 34] is applied to minimize the learnt clauses.
- The clause size reduction with the all-UIP learning technique from [18] is applied to reduce the clauses learnt from soft conflicts. This technique is particularly useful for MaxCDCL, because a clause learnt from a soft conflict is usually longer than a clause learnt from a hard conflict. The all-UIP technique allows to significantly reduce the size of a clause learnt from a soft conflict.
- The learnt clause management technique proposed in [22] is incorporated into MaxCDCL, allowing more learnt clauses to be kept in the clause database.

In addition, let $S = \{y_1, \dots, y_{|S|}\}$ be the set of all soft literals. When $|S| \times (\text{UB} - 1) \leq 10^4$, the sequential SAT encoding [42] of the cardinality constraint $\neg y_1 + \neg y_2 + \dots + \neg y_{|S|} < \text{UB}$ is added to the input instance before starting the search. MaxCDCL alternates LRB phases and VSIDS phases for its search as MapleCOMSPS_LRB, using the LRB heuristic and the VSIDS heuristic modified as in subsection 4.5, respectively. Each phase is limited to a number of unit propagations specified by the parameter *phaseLength*, which is initialized to 2×10^7 and is doubled every cycle of LRB phase and VSIDS phase. In the VSIDS phase, the glucose restart strategy [7] is used; in the LRB phase, the Luby restart strategy [33] is used.

We plan to implement MaxCDCL on top of Kissat [13], the winner of the SAT2020 competition, which might further improve its performance.

5 Experimental Evaluation

We report on an experimental investigation to assess the performance of MaxCDCL. We ran all experiments with Intel Xeon CPUs E5-2680@2.40GHz under Linux with 32GB of memory, using the following benchmark sets, unless otherwise stated:

MSE19U20: The union of all the instances used in the complete unweighted track of the MaxSAT Evaluations (MSE) 2019 and 2020, *a total of 1000 distinct instances*.

MC: A subset of the Master Collection of instances from the MaxSAT evaluations held until 2019². It contains 16080 unweighted (partial) MaxSAT instances, classified into 51 families and 76 subfamilies. MC includes all the instances of the 63 subfamilies having 100 instances or less, and the first 100 instances as they occur in the natural order in each of the remaining 13 subfamilies, *considering a total of 3614 instances*. This selection provides a simple, deterministic and objective criterion that does not favor any solver; and the experiments can be easily reproduced. MC contains 726 instances that also belong to MSE19U20.

The cutoff time is one hour (3600s) per instance as in the MaxSAT Evaluation. For MaxCDCL and its variants, this includes 60 seconds to find a feasible upper bound UB_f with SatLike (version 3.0). Note that MaxCDCL and its variants do not start the search

¹ The source code of MaxCDCL is available at <https://home.mis.u-picardie.fr/~cli/EnglishPage.html>

² <https://www.cs.toronto.edu/maxsat-lib/maxsat-instances/master-set/unweighted/>

■ **Table 1** Comparison of MaxCDCL with its variants for MSE19U20 (left) and MC (right).

	#solv	avg	#solv	avg
MaxCDCL\LA	505	255s	2183	194s
MaxCDCL\harden	664	281s	2878	194s
MaxCDCLalwaysLA	681	249s	2962	193s
MaxCDCLioLA	704	268s	2963	168s
MaxCDCL\VSIDSbyLA	724	268s	3003	165s
MaxCDCL	734	256s	3022	156s

from $UB = UB_f$, but from $UB = 2^0$, then $2^1, 2^2, \dots$, until a feasible solution is found or $2^i > UB_f$. In the latter case, UB is set to $UB_f + 1$. Then, UB is gradually decreased until no better solution can be found (see Section 4.1).

The experiments are presented as follows. Firstly, we analyse the impact of the components implemented in MaxCDCL. Secondly, we compare the performance of MaxCDCL with that of the top 5 solvers in MSE2020. Thirdly, we show the complementarity of MaxCDCL with the top 5 solvers by comparing the number of instances solved using a portfolio solver with and without MaxCDCL. Finally, we compare MaxCDCL with a state-of-the-art BnB solver.

5.1 MaxCDCL Components

Table 1 compares MaxCDCL with the following variants:

MaxCDCL\LA. MaxCDCL without lookahead, i.e. the condition to lookahead is never satisfied in line 3 of Algorithm 2. Note that after finding a feasible UB , MaxCDCL\LA performs linear SAT-UNSAT search as a model-guided SAT-based MaxSAT solver.

MaxCDCL\harden. MaxCDCL without hardening (i.e., lines 17 and 18 in Algorithm 1 are removed).

MaxCDCLalwaysLA. MaxCDCL that looks ahead at every branch, i.e., the condition to lookahead is always satisfied in line 3 of Algorithm 2.

MaxCDCLioLA. MaxCDCL that, when detecting cores in lookahead, always propagates the soft literals in the ordering as the corresponding soft clauses occur in the input instance.

MaxCDCL\VSIDSbyLA. MaxCDCL that does not increase the VSIDS score of the variables contributing to a local core detected in lookahead as described in Subsection 4.5.

In Table 1, columns “#solv” give the number of solved instances and columns “avg” give the mean time in seconds (including the 60s used by SatLike) needed to solve these instances. These results indicate that a careful configuration combining clause learning and BnB is crucial for the performance of MaxCDCL, including: hardening based on local core detection and clause learning, the selective and adaptive application of lookahead and the ordering to propagate the soft literals when detecting local cores, because the absence of any of these components makes a significant number of instances out of reach of MaxCDCL. Without this configuration, MaxCDCL\LA solves 229 instances less than MaxCDCL in MSE19U20 and 839 instances less than MaxCDCL in MC. Note that although the hardening of a soft literal requires a distinct learnt clause, it does not increase the total memory usage, because it allows to avoid many learnt clauses by reducing search space.

Recall that the most fundamental feature of MaxCDCL is the clause learning from soft conflicts. We computed the average length of a clause learnt from a soft (hard) conflict for each solved instance by MaxCDCL in MSE19U20, and found that the median average length

■ **Table 2** Results for MSE19U20 (left) and MC (right) with top 5 solvers.

	#solv	avg	#uniq	#win	#solv	avg	#uniq	#win
MaxHS	769	177s	11	36	3037	85.5s	26	116
EvalMaxSAT	759	129s	1	43	3002	69.7s	4	147
UWrMaxSAT	745	128s	3	42	2969	51.6s	7	141
RC2-B	728	164s	0	62	2948	70.1s	1	173
Open-WBO	695	157s	3	71	2906	89.7s	4	190
MaxCDCL	734	256s	16	–	3022	156s	67	–

of a clause learnt from a soft (hard) conflict is 23.67 (19.2) among the 734 instances solved in the set. Note that the learnt clause length averaged across the 734 instances does not make sense because it is biased by few instances with very long learnt clause length.

In addition, the comparison of MaxCDCL with MaxCDCL\VSIDSbyLA suggests that the VSIDS heuristic might be improved by lookahead, indicating a promising research direction.

To complete the subsection, we mention the impact of two other components of MaxCDCL: (1) the local search by SatLike in preprocessing to compute a feasible UB allows MaxCDCL to solve 8 more instances in MSE19U20; (2) the sequential cardinality constraint encoding is applied to about the 20% of the instances in MSE19U20 for at least one UB, helping solve 39 extra instances in MSE from highly symmetric problems such as drmx-at-most-k.

5.2 Comparison with the top 5 Solvers in MSE2020

A total of 15 solvers competed in the complete unweighted track of MSE2020 [9]. We consider the top 5 solvers: *MaxHS* (*mhs* in short), which is MHS-guided; *EvalMaxSAT* (*eval* in short), *RC2-B* (*rc2* in short) and *open-wo-res-mergesat-v2* (*Open-WBO* or *owbo* in short), which are core-guided; and *UWrMaxSAT* (*uwr* in short), which combines both core-guided and model-guided solving. We executed the versions used in MSE2020 in all the experiments.

Table 2 shows the results for MSE19U20 and MC, respectively. Column “#uniq” has the number of instances that were only solved by the solver in the row. Column “#win” has the number of instances solved by MaxCDCL but not by the solver in the row.

We observe that MaxCDCL solves more instances than two top 5 solvers in MSE19U20 and four top 5 solvers in MC. More importantly, MaxCDCL solves a significant number of instances that other solvers cannot solve. For example, MaxCDCL solves 116 instances in MC that MaxHS does not solve. If we consider all the solvers together, there is also a significant number of instances solved by MaxCDCL that no other solver is able to solve: 16 instances in MSE19U20 and 67 instances in MC that mainly come from the subfamilies MaxClique, MaxCut and UAQ. These results show that the existing MaxSAT solvers, especially the model-guided and core-guided ones, are able to solve similar kinds of instances. Nevertheless, MaxCDCL has the potential to solve new kinds of instances that are not solvable with the current MaxSAT techniques. It is important to note that MaxCDCL is far from being as optimized as the other solvers, which are the result of a process of continuous improvements since more than ten years.

5.3 Combining MaxCDCL with existing solvers

Given two deterministic solvers X and Y and a time limit T to solve an NP-hard problem such as MaxSAT, the simplest way to try to solve more instances than X and Y alone within the time limit T is to combine X and Y by running X within the time limit $T/2$, and then Y from scratch within the remaining time $T/2$ if the instance is not solved by X .

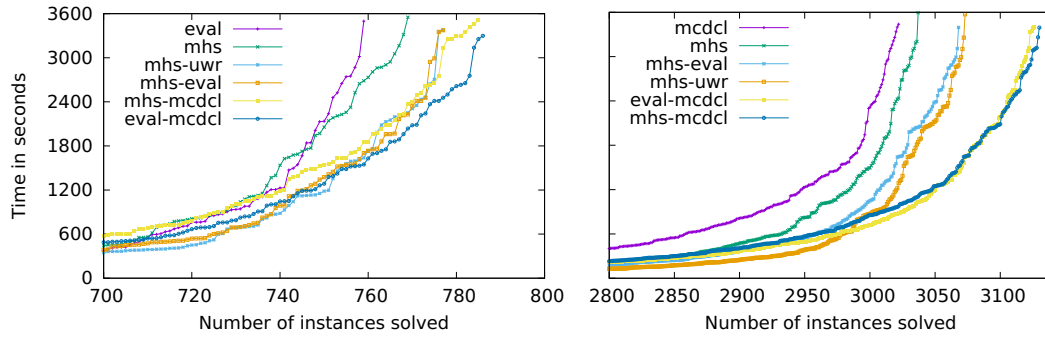
■ **Table 3** Results for MSE19U20 (left) and MC (right). The entry in cell (X, Y) for $X \neq Y$ is the number of instances solved by running solver X for 1800 seconds, and then solver Y from scratch for 1800 seconds if the instance is not solved by X . The entry in cell (X, X) (in the diagonal in grey) is the number of instances solved by running solver X for 1800 seconds. Column X in the last row recalls the results of solver X with 3600 seconds. The best results are in bold.

	mhs	eval	uwr	rc2	owbo	mcdbl		mhs	eval	uwr	rc2	owbo	mcdbl
mhs	747	777	777	770	763	785		3009	3068	3073	3056	3049	3130
eval	777	745	760	751	760	786		3068	2972	3019	2986	3013	3126
uwr	777	760	730	745	746	774		3073	3019	2951	3000	2998	3098
rc2	770	751	745	713	745	778		3056	2986	3000	2921	2981	3105
owbo	763	760	746	745	675	746		3049	3013	2998	2981	2865	3076
mcdbl	785	786	774	778	746	711		3130	3126	3098	3105	3076	2992
3600s	769	759	745	728	695	734		3037	3002	2969	2948	2906	3022

Table 3 shows the results of all possible pairwise combinations of the top 5 solvers and MaxCDCL (*mcdbl*) in MSE2020 for $T = 3600s$. Each cell (X, Y) for $X \neq Y$ contains the number of instances solved by running solver X for 1800s and then solver Y from scratch for 1800s in MSE19U20 (left) and MC (right). Each cell (X, X) (in the diagonal in grey) contains the number of instances solved by X in 1800s. Column X in the last row recalls the results of X with 3600 seconds. Combining any of the top 5 solvers with MaxCDCL solves more instances than this solver and MaxCDCL alone within 3600s, while this is not always true when combining two top 5 solvers. For example, combining MaxHS and Open-WBO solves 763 instances in MSE19U20 within 3600s, while MaxHS alone solves 769 instances within 3600s. This shows that MaxCDCL is more complementary with the top 5 solvers than other solvers.

More importantly, MaxCDCL combined with the top 2 solvers, MaxHS and EvalMaxSAT, solves the highest numbers (785 and 786) of instances in MSE19U20. This result is significantly better than that of MaxHS or EvalMaxSAT alone, and the best combination without MaxCDCL solves only 777 instances. The results are even more striking in MC, where the worst combination of MaxCDCL with a top 5 solver is better than any other combination not including MaxCDCL, and combining MaxHS and MaxCDCL gives the best results, solving 93 instances more than the previous best result achieved by MaxHS alone, and 57 instances more than the best combination without MaxCDCL.

Figure 3 shows cactus plots comparing the best two combinations of solvers with MaxCDCL, the best two combinations without MaxCDCL, as well as the best two mono-solvers, for MSE19U20 (left) and MC (right). Other solvers and combinations are excluded for readability reasons. For any time T ($0 < T \leq 3600s$), a curve gives the number of instances solved by a mono-solver or a combination of two solvers within T , where the number of instances solved by a combination X - Y of solvers X and Y within T is the number of instances solved by running X for $T/2$, and then Y for $T/2$. The solving time of a combination X - Y of solvers X and Y for an instance is twice the minimum solving time among X and Y . This simulates a parallel execution of X and Y by alternating them in small time periods. The plots clearly show the advantage of combining an existing solver with MaxCDCL, allowing to solve the highest number of instances within 3600s, and the advantage becomes greater as the running time is increased.



■ **Figure 3** Cactus plots of the best two combinations with MaxCDCL (*mcdcl* in short), the best two without MaxCDCL, and the best two mono-solvers for MSE19U20 (left) and MC (right). For each point (N, T) in a curve for a mono-solver X , N is the number of instances solved by X in T seconds; and for each point (N, T) in a curve for a combination X - Y of solvers X and Y , N is the number of instances solved by running X for $T/2$ seconds followed by running Y for $T/2$ seconds. The names of the solvers and combinations are listed in the order of the highest points of the corresponding curves from left to right for readability.

■ **Table 4** Results for MSE16 and MSE19U20.

		ahmaxsat			MaxCDCL		
		#ins	#solv	avg	#solv	avg	#win
MSE16	ms_ran	454	259	744s	27	1535s	0
	ms_craf	402	230	33.0s	47	324s	0
	ms_in	55	0	-	37	413s	37
	pms_ran	210	209	148s	141	772s	0
	pms_craf	678	369	242s	645	126s	276
	pms_in	601	183	515s	512	178s	330
MSE19U20		1000	270	422s	734	256s	481

5.4 Comparison with a BnB Solver

We compare MaxCDCL with *ahmaxsat*, which was the best BnB solver in the last MaxSAT evaluation (MSE2016) in which BnB solvers competed. We use the MSE2016 instances of the categories MaxSAT (*ms*) and partial MaxSAT (*pms*). Each category has *random* (ran), *crafted* (craf), and *industrial* (in) instances. Thus, we consider a total of 6 families.

The results are shown in Table 4, where the number of instances for each family is given in column “#ins”. We observe that *ahmaxsat* solves more random and non-partial crafted instances. A learnt clause for these instances with randomness and limited size usually contains most of the variables of the instances and is hardly useful. So, the higher use of lower bounding methods and the lack of clause learning in *ahmaxsat* are adequate for them, and the higher use of lower bounding methods (like in MaxCDCLalwaysLA) does not improve MaxCDCL for them because of clause learning. However, *ahmaxsat* has poor performance on other instances. Instead, MaxCDCL solves a much higher number of such instances.

6 Conclusion

We described MaxCDCL, a MaxSAT solver that combines, for the first time to the best of our knowledge, branch and bound and clause learning. The main differences of MaxCDCL with existing SAT-based MaxSAT solvers are the following:

- SAT-based MaxSAT solvers use a CDCL SAT solver as a black box and do not interfere in the internal operations of the SAT solver when solving an instance, while MaxCDCL itself can be considered a SAT solver extended to handle soft conflicts.
- Both MaxCDCL and model-guided MaxSAT solvers have an upper bound $UB-1$ of the number of soft clauses that can be falsified. The difference lies in how to exploit this UB. Let $falseS$ denote the set of already falsified soft clauses. On the one hand, model-guided MaxSAT solvers call a SAT solver after encoding UB into CNF. If no hard clause is falsified, the SAT solver backtracks only after $|falseS| \geq UB$, because no clause encoding UB is falsified if $|falseS| < UB$. On the other hand, MaxCDCL computes a lower bound LB of the number of soft clauses that will be falsified (but not yet falsified), and backtracks as soon as $LB + |falseS| \geq UB$. Thus, MaxCDCL is able to backtrack much earlier than model-guided MaxSAT solvers.
- MaxCDCL, core-guided or MHS-guided MaxSAT solvers all identify cores. However, a core-guided or MHS-guided MaxSAT solver only identifies *global cores* (i.e., the cores that do not depend on any partial assignment) in order to relax them, while MaxCDCL detects *local cores* by using UP under a partial assignment to derive a soft conflict for learning a clause and backtracking early. Note that identifying a global core is NP-hard, while detecting a local core by applying UP is polynomial.

The extensive experimentation conducted shows that MaxCDCL is ranked among the top 5 exact MaxSAT solvers in the 2020 MaxSAT evaluation. Furthermore, it solves a significant number of instances that other solvers cannot solve, suggesting that combining branch and bound and clause learning has the potential to solve new kinds of instances that are not solvable with current MaxSAT techniques. More importantly, combining MaxCDCL with the existing solvers allows to solve the highest number of MaxSAT instances.

Detailed analyses indicate that the performance of MaxCDCL comes from a careful configuration combining clause learning and BnB, including hardening based on local core detection and clause learning, the selective and adaptive application of lookahead, and the ordering to propagate the soft literals when detecting local cores.

We believe that the proposed approach opens new and promising research directions, including for example: (1) improving the quality of the clauses learnt from soft conflicts by designing new soft literal orderings in lookahead; (2) exploiting the relationship of SAT and MaxSAT for improving SAT and MaxSAT solving; (3) adapting the approach of MaxCDCL to other problems such as pseudo-Boolean optimization and Max-CSP; (4) extending MaxCDCL to weighted MaxSAT, in which each soft clause is weighted. In the extension of MaxCDCL to weighted MaxSAT, each soft clause is represented by a weighted soft literal. A local core detected by lookahead is weighted by the minimum soft literal weight it contains, and other weights are split to be used in other core detections. When the total weight of all detected local cores reaches UB, a soft conflict is discovered. The challenge here is that there can be more local cores to detect than in the unweighted case. Thus, the clause learnt from a soft conflict can be longer. Special strategies in clause and soft literal ordering should be designed to learn shorter clauses from soft conflicts.


References

- 1 André Abramé and Djamal Habet. Ahmaxsat: Description and evaluation of a branch and bound Max-SAT solver. *J. Satisf. Boolean Model. Comput.*, 9:89–128, 2014.
- 2 André Abramé and Djamal Habet. Learning nobetter clauses in Max-SAT branch and bound solvers. In *Proceedings of ICTAI 2016*, pages 452–459, 2016.

- 3 Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. Solving (Weighted) Partial MaxSAT through satisfiability testing. In *Proceedings of SAT 2009*, pages 427–440. Springer LNCS 5584, 2009.
- 4 Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. A new algorithm for weighted partial MaxSAT. In *Proceedings AAAI 2010*, pages 3–8, 2010.
- 5 Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. SAT-based MaxSAT algorithms. *Artificial Intelligence*, 196:77–105, 2013.
- 6 Carlos Ansótegui and Joel Gabàs. WPM3: An (in)complete algorithm for Weighted Partial MaxSAT. *Artificial Intelligence*, 250:37–57, 2017.
- 7 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings IJCAI 2009*, pages 399–404, 2009.
- 8 Fahiem Bacchus. MaxHS in the 2020 MaxSAT Evaluation. In *MaxSAT Evaluation 2020: Solver and Benchmark Descriptions*, pages 19–20, 2020.
- 9 Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins. MaxSAT Evaluation 2020: Solver and benchmark descriptions, 2020.
- 10 Fahiem Bacchus, Antti Hyttinen, Matti Järvisalo, and Paul Saikko. Reduced cost fixing in MaxSAT. In *Proceedings of CP 2017*, Springer LNCS, pages 641–651, 2017.
- 11 Fahiem Bacchus, Matti Järvisalo, and Martins Ruben. Maximum satisfiability. In *Handbook of satisfiability, second edition*, pages 929–991. IOS Press, 2021.
- 12 Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *J. Satisf. Boolean Model. Comput.*, 7(2-3):59–6, 2010.
- 13 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*, page 50, 2020.
- 14 Mohamed Sami Cherif, Djamal Habet, and André Abramé. Understanding the power of Max-SAT resolution through UP-resilience. *Artificial Intelligence*, 289:103397, 2020.
- 15 Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Proceedings of CP 2011*, page 225–239. Springer, 2011.
- 16 Jessica Davies and Fahiem Bacchus. Exploiting the power of MIP solvers in MAXSAT. In *Proceedings of SAT 2013*, pages 166–181. Springer, 2013.
- 17 Jo Devriendt, Ambros Gleixner, and Jakob Nordström. Learn to relax: Integrating 0-1 integer linear programming with pseudo-boolean conflict-driven search. *Constraints*, pages 1–30, 2021.
- 18 Nick Feng and Fahiem Bacchus. Clause size reduction with all-uir learning. In *Proceedings of SAT 2020, Springer LNCS 12178*, pages 28–45, 2020.
- 19 Graeme Gange, Jeremias Berg, Emir Demirović, and Peter J Stuckey. Core-guided and core-boosted search for CP. In *Proceedings of CPAIOR 2020*, pages 205–221, 2020.
- 20 Federico Heras, Javier Larrosa, and Albert Oliveras. MiniMaxSAT: An efficient Weighted Max-SAT solver. *Journal of Artificial Intelligence Research*, 31:1–32, 2008.
- 21 Alexey Ignatiev, António Morgado, and João Marques-Silva. RC2: an efficient maxsat solver. *J. Satisf. Boolean Model. Comput.*, 11(1):53–64, 2019.
- 22 Stepan Kochemazov. Improving implementation of SAT competitions 2017–2019 winners. In *Proceedings of SAT 2020, LNCS 12178*, pages 139–148, 2020.
- 23 Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. QMaxSAT: A Partial Max-SAT solver. *J. Satisf. Boolean Model. Comput.*, 8(1/2):95–100, 2012.
- 24 Adrian Kuegel. Improved exact solver for the Weighted MAX-SAT problem. In *Proceedings of Workshop Pragmatics of SAT, POS-10, Edinburgh, UK*, pages 15–27, 2010.
- 25 Zhendong Lei and Shaowei Cai. Solving (Weighted) Partial MaxSAT by dynamic local search for SAT. In *Proceedings of IJCAI 2018*, pages 1346–1352, 2018.
- 26 Chu Min Li and Felip Manyà. MaxSAT, hard and soft constraints. In *Handbook of satisfiability, second edition*, pages 903–927. IOS Press, 2021.
- 27 Chu Min Li, Felip Manyà, Noureddine Ould Mohamedou, and Jordi Planes. Resolution-based lower bounds in MaxSAT. *Constraints*, 15(4):456–484, 2010.

- 28 Chu Min Li, Felip Manyà, and Jordi Planes. Exploiting unit propagation to compute lower bounds in branch and bound Max-SAT solvers. In *Proceedings of CP 2005*, pages 403–414. Springer, 2005.
- 29 Chu Min Li, Felip Manyà, and Jordi Planes. Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In *Proceedings of AAAI 2006*, pages 86–91, 2006.
- 30 Chu Min Li, Felip Manyà, and Jordi Planes. New inference rules for Max-SAT. *Journal of Artificial Intelligence Research*, 30:321–359, 2007.
- 31 Chu-Min Li, Fan Xiao, Mao Luo, Felip Manyà, Zhipeng Lü, and Yu Li. Clause vivification by unit propagation in CDCL SAT solvers. *Artificial Intelligence*, 279, 2020.
- 32 Jia Hui Liang, Chanseok Oh, Vijay Ganesh, Krzysztof Czarnecki, and Pascal Poupart. MapleCOMSPS, MapleCOMSPS LRB, MapleCOMSPS CHB. In *Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions*, pages 52–53, 2016.
- 33 Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.
- 34 Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In *Proceedings of IJCAI 2017*, pages 703–711, 2017.
- 35 Vasco M. Manquinho, Joao Marques-Silva, and Jordi Planes. Algorithms for weighted Boolean optimization. In *Proceedings of SAT 2009*, pages 495–508. Springer LNCS 5584, 2009.
- 36 Joao Marques-Silva and Vasco M. Manquinho. Towards more effective unsatisfiability-based maximum satisfiability algorithms. In *Proceedings of SAT 2008*, pages 225–230. Springer LNCS 4996, 2008.
- 37 Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. In *Proceedings of SAT 2014*, volume 8561 of *LNCS*, pages 438–445. Springer, 2014.
- 38 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of DAC 2001*, pages 530–535. ACM, 2001.
- 39 Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided MaxSAT resolution. In *Proceedings of AAAI 2014*, pages 2717–2723, 2014.
- 40 Tobias Paxian and Bernd Becker. Pacose: An iterative SAT-based MaxSAT solver. In *MaxSAT Evaluation 2020: Solver and Benchmark Descriptions*, page 12, 2020.
- 41 Paul Saikko, Jeremias Berg, and Matti Järvisalo. LMHS: A SAT-IP hybrid MaxSAT solver. In *Proceedings of SAT 2016*, volume 9710 of *LNCS*, pages 539–546, 2016.
- 42 Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Proceedings of CP 2005*, pages 827–831. Springer LNCS 3709, 2005.
- 43 Fulya Trösser, Simon De Givry, and George Katsirelos. Relaxation-aware heuristics for exact optimization in graphical models. In *Proceedings of CPAIOR 2020*, pages 475–491. Springer, 2020.
- 44 Aolong Zha. QMaxSAT in MaxSAT Evaluation 2018. In *Proceedings of the MaxSAT Evaluation 2020*, page 16, 2020.

Improving Local Search for Minimum Weighted Connected Dominating Set Problem by Inner-Layer Local Search

Bohan Li ✉ 


State Key Laboratory of Computer Science Institute of Software,
Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology,
University of Chinese Academy of Sciences, Beijing, China

Kai Wang ✉

School of Computer Science and Information Technology,
Northeast Normal University, Changchun, China

Yiyuan Wang ✉ 

School of Computer Science and Information Technology,
Northeast Normal University, Changchun, China
Key Laboratory of Applied Statistics of MOE,
Northeast Normal University, Chnagchun, China

Shaowei Cai¹ ✉ 

State Key Laboratory of Computer Science Institute of Software,
Chinese Academy of Sciences, Beijing, China
School of Computer Science and Technology,
University of Chinese Academy of Sciences, Beijing, China

Abstract

The minimum weighted connected dominating set (MWCDS) problem is an important variant of connected dominating set problems with wide applications, especially in heterogenous networks and gene regulatory networks. In the paper, we develop a nested local search algorithm called NestedLS for solving MWCDS on classic benchmarks and massive graphs. In this local search framework, we propose two novel ideas to make it effective by utilizing previous search information. First, we design the restart based smoothing mechanism as a diversification method to escape from local optimal. Second, we propose a novel inner-layer local search method to enlarge the candidate removal set, which can be modelled as an optimized version of spanning tree problem. Moreover, inner-layer local search method is a general method for maintaining the connectivity constraint when dealing with massive graphs. Experimental results show that NestedLS outperforms state-of-the-art meta-heuristic algorithms on most instances.

2012 ACM Subject Classification Theory of computation → Randomized local search; Applied computing → Operations research

Keywords and phrases Operations Research, NP-hard Problem, Local Search, Weighted Connected Dominating Set Problem

Digital Object Identifier 10.4230/LIPIcs.CP.2021.39

Supplementary Material *Software (Source Code)*: <https://github.com/DouglasLee001/NestedLS>

Funding This work is partially supported by National Key R&D Program of China (2019AAA0105200), Beijing Academy of Artificial Intelligence (BAAI), NSFC Grant 61806050, and Jilin Science and Technology Association QT202005.

¹ corresponding author



© Bohan Li, Kai Wang, Yiyuan Wang, and Shaowei Cai;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 39; pp. 39:1–39:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Given an undirected connected graph $G = (V, E)$, a set $D \subseteq V$ forming a connected subgraph in G is called a connected dominating set (CDS) if each vertex in V either belongs to D or is adjacent to at least one vertex in D . The minimum connected dominating set (MCDS) problem is to find a CDS with the minimum size. MCDS is a well-known combinatorial optimization problem with important applications [1, 14].

MCDS assumes that vertices are equally important. However, this assumption fails to hold in many real world scenarios where each vertex is associated with various types of weights. A specific application is to model heterogeneous networks [22] where each vertex generates different cost (e.g., energy consumption and communication delay). The paradigm of handling such vertex weighted graph refers to an important generalization of MCDS, i.e., minimum weighted connected dominating set (MWCDS) problem, aiming to find a CDS with the minimum total weight. The MWCDS is used to form a low-cost network backbone for communication applications where the cost usually represents the power consumption rate or corresponding security coefficient of backbones [27, 28]. Moreover, MWCDS has other applications in biological networks [16] and generating pictorial storylines [23].

1.1 Related Work

MWCDS is a classic NP-hard problem, meaning that there are no polynomial-time algorithms for the MWCDS problem, unless $NP=P$. Although MCDS is widely studied and many specialized algorithms have already been proposed to solve MCDS on graphs with different sizes, these MCDS algorithms [9, 18, 11, 13] cannot be directly used to deal with the MWCDS problem because they fail to consider the weight information and structure characteristics.

Because of its NP-hardness, much of the research effort in the past decade has focused on obtaining a good MWCDS solution within a reasonable time. In the literature, two types of algorithms are mainly distinguished for MWCDS, i.e., approximation algorithms and meta-heuristic algorithms. The approximation algorithms can find approximate solutions with provable guaranteed approximation ratio, but they usually have poor performance in practice, especially in massive graphs. Representative approximation algorithms for MWCDS mainly used centralized methods [2, 29] or distributed methods [5, 21]. According to the literature, the current best meta-heuristic algorithm for MWCDS is ACO-RVNS [3] based on ant colony optimization and reduced variable neighborhood search.

1.2 Our Contributions

Previous MWCDS algorithms performed well for classic benchmarks, but they had poor performance on massive graphs. In this paper, to further improve the performance of MWCDS on both classic and massive graphs, we propose a nested local search framework called NestedLS, including three phases, i.e., vertices swapping phase, tree reconstruction phase and solution restart phase. Based on the framework, we design two novel ideas by utilizing previous search information.

First, we propose the restart based smoothing mechanism (*ReSmooth*), which can be viewed as a diversification method. In order to escape from a local optima, *ReSmooth* restarts the algorithm by reconstructing a new solution during the solution restart phase. During the reconstruction process, two kinds of previous search information (w.r.t, non-dominated information and best solution information) are inherited to guide the algorithm to the promising search space, resulting in a new inheriting scoring function, denoted as

$score_{inher}$. Moreover, after a few restart operations, the initial solution may converge. To address this, we propose a smoothing mechanism based on the repeating rate of solution to further diversify the search spaces.

The second and more important idea is the inner-layer local search method (*InnerSearch*). Although an efficient tree-based connectivity maintenance method (TBC) proposed by Li [13] used the spanning tree to maintain the candidate removal set, it cannot utilize search information when constructing the spanning tree. In order to enlarge the candidate removal set, the *InnerSearch* is applied to reconstruct the spanning tree by modelling it as a weighted max-leaf spanning tree problem (WMST). Meanwhile, based on three novel intuitions of WMST, the corresponding vertex selection rule is proposed to guide the *InnerSearch* to construct the spanning tree and further improve it by a local search procedure.

These proposed ideas can be generally applied to other heuristic algorithms. Specifically, *InnerSearch* is a general method for maintaining the connectivity constraint when dealing with massive graphs, and *ReSmooth* provides a novel diversification scheme for restart-based heuristic algorithms.

Extensive experiments are carried out to evaluate NestedLS on classic benchmarks and massive graphs. Experimental results indicate that NestedLS outperforms other state-of-the-art MWCDS heuristic algorithms on most instances, and confirm the effectiveness of two novel ideas.

2 Preliminaries

Let $G = (V, E, w)$ be a weighted graph where V is the set of vertices, E is the set of edges and each vertex $v \in V$ is associated with a positive weight $w(v)$. For a vertex v , its neighborhood is $N_G(v) = \{u \in V | \{u, v\} \in E\}$, and its closed neighborhood is $N_G[v] = N_G(v) \cup \{v\}$. The degree of a vertex v , denoted as $d_G(v)$, is defined as $|N_G(v)|$, and Δ_G is the maximum number of $d_G(v)$ for $\forall v \in V$. Given a vertex set $S \subseteq V$, $N_G(S) = \bigcup_{v \in S} N_G(v) \setminus S$ and $N_G[S] = \bigcup_{v \in S} N_G[v]$ stands for the neighborhood and closed neighborhood of S , respectively. $G[S] = (V_S, E_S)$ is a subgraph in G induced by S such that $V_S = S$ and E_S consists of all the edges in E whose endpoints are in S . A weighted graph G is connected when it has at least one vertex and there is a path between every pair of vertices.

► **Definition 1.** Given a weighted connected graph G , a vertex in G is an articulation vertex iff removing it, together with the edges connected to it, disconnects the graph. The articulation vertex set of G is denoted as $art(G)$.

Given a vertex set $D \subseteq V$, a vertex $v \in V$ is *dominated* by D if $v \in N_G[D]$, and is non-dominated otherwise. We use $D \subseteq V$ to denote a candidate solution and the weight of D is $w(D) = \sum_{v \in D} w(v)$. $unDom_G(D) = V \setminus N_G[D]$ denotes a subset of vertices in G non-dominated by D . If $G[D]$ is connected and D dominates all vertices in V , D is a connected dominating set (CDS). The minimum weighted connected dominating set problem (MWCDS) is to find a CDS with the minimum total weight.

2.1 Review of Scoring Function for MWCDS

The frequency based scoring function $score_f$ is recently proposed by Wang et al. [26]. Each vertex $v \in V$ has a property: frequency, denoted as $freq[v]$. It works as follows: 1) at first, $freq[v]=1$ for $\forall v \in V$; 2) at the end of each iteration of local search, $freq[v]=freq[v] + 1$ for each non-dominated vertex. If $u \in D$, $score_f(u) = -\sum_{v \in C_1(u)} freq[v]/w(u)$, and otherwise $score_f(u) = \sum_{v \in C_2(u)} freq[v]/w(u)$, where $C_1(u)$ is the set of dominated vertices that would

become non-dominated by removing u from D and $C_2(u)$ is the set of non-dominated vertices that would become dominated by adding u to D . Moreover, considering that age ² is usually used to break ties for diversification, the selection rule is described as follows.

Selection Rule: Select the added or removed vertex with the greatest $score_f$, breaking ties by preferring the one with the greatest age .

2.2 Review of TBC

For combinatorial optimization problem with connectivity constraint, a key factor to the performance is the connectivity maintenance methods, especially for massive graphs. To tackle it, a tree-based connectivity maintenance called TBC method was proposed [13], inspired by spanning trees. Given a candidate solution D , a spanning tree T of $G[D]$ and its corresponding leaf set $LS(T)$ are maintained during the search process. Each vertex $v \in LS(T)$ is allowed to be removed from D , while all other vertices are forbidden to be removed. Details for TBC can refer to [13].

3 The NestedLS Algorithm

In this section, we propose an algorithm for solving MWCDs called NestedLS.

The pseudo code of NestedLS is presented in Algorithm 1. On a top level, NestedLS works as follows. After the initialization, a loop (lines 3–18) is executed until a given time limit is reached, and the best solution is finally returned (line 19). Each iteration of the loop consists of three phases, namely vertices swapping phase, tree reconstruction phase and solution restart phase. At the first phase (lines 4–12), the candidate solution is updated by swapping vertices. In the second phase (lines 13–14), the spanning tree is periodically updated for diversification. During the third phase (lines 15–18), the candidate solution and corresponding spanning tree are rebuilt if the algorithm falls into the local optima.

Before detailed description, we first introduce some notations and definitions. In NestedLS, $NoImproveStep$ denotes the number of consecutive iterations without improvement. $MaxNoImprove$ and $TreeNoImprove$ denote the parameters for reconstructing the solution and the spanning tree respectively. D , D^* and D_{last} denote the current candidate solution, the best solution and the previous solution after last construction, respectively. During the search process, two candidate selection subsets are maintained as follows.

- (1) The candidate subset for addition is defined as $candAdd(D) = N_G(D) \cap N_G(unDom_G(D))$, where $N_G(D)$ contains vertices maintaining connectivity and $N_G(unDom_G(D))$ is adjacent to the non-dominated vertex set. To avoid visiting previous candidate solutions, we use the CC² strategy [26] to further restrain $candAdd(D)$.
- (2) The candidate subset for removal is denoted as $candRem(D)$. If $|D| < \kappa$, $candRem(D) = D \setminus art(G[D])$ where $art(G[D])$ is calculated by Tarjan's algorithm [10]. Otherwise, TBC is adopted and $candRem(D) = LS(T)$. To overcome the cycling problem, the tabu method [8] is applied to exclude those just added vertices from $candRem(D)$ for the next tt iterations. In our work, $tt = 5 + rand(10)$ and $\kappa = 100$.

Now we describe the NestedLS algorithm in detail.

² The age of a vertex v is the number of steps that have occurred since v last changed its state.

■ **Algorithm 1** The NestedLS algorithm.

Input: A weighted graph $G = (V, E, w)$, the cutoff time
Output: The best obtained solution D^*

```

1  $D := D^* := D_{last} := ReSmooth(\emptyset, V)$ ;
2  $T := InnerSearch(G[D])$  and  $NoImproveStep := 1$ ;
3 while  $timeElapse < cutoff$  do
4   for  $i := 1$  to  $neighborSize$  do
5      $\lfloor$  choose vertex  $u \in candRem(D)$  using selection rule and  $D := D \setminus \{u\}$ ;
6   while  $|unDom_G(D)| \neq 0$  and  $w(D) < w(D^*)$  do
7      $\lfloor$  choose vertex  $v \in candAdd(D)$  using selection rule and  $D := D \cup \{v\}$ ;
8   if  $D$  is a feasible solution then
9      $\lfloor D^* := D$  and  $NoImproveStep := 1$ ;
10  else
11     $\lfloor freq[v] := freq[v] + 1$ , for  $\forall v \in unDom_G(D)$ ;
12     $\lfloor NoImproveStep := NoImproveStep + 1$ ;
13  if  $NoImproveStep \% TreeNoImprove == 0$  then
14     $\lfloor T := InnerSearch(G[D])$ ;
15  if  $NoImproveStep > MaxNoImprove$  then
16     $\lfloor NoImproveStep := 1$ ;
17     $\lfloor D := D_{last} := ReSmooth(D_{last}, D^*)$ ;
18     $\lfloor T := InnerSearch(G[D])$ ;
19 return  $D^*$ ;

```

In the beginning, D , D^* and D_{last} are initialized by the *ReSmooth* procedure (line 1) which will be discussed in Section 4. The corresponding spanning tree T is built by a novel inner-layer local search, which will be introduced in Section 5, and $NoImproveStep$ is set to 1 (line 2).

In the vertices swapping phase, $neighborSize$ vertices are first chosen from $candRem(D)$ using the selection rule. Then, vertices $v \in candAdd(D)$ are added via the selection rule, until there are no non-dominated vertices or $w(D) \geq w(D^*)$. During this process, the total weight of current candidate solution stays below the best value.

Thus, after swapping vertices, if a feasible solution is obtained, indicating that a better solution is found, then D^* and $NoImproveStep$ are updated (line 9). Otherwise, the corresponding $freq$ values and $NoImproveStep$ are increased by one (lines 10–12).

In the tree reconstruction phase, if the condition is satisfied (line 13), then T will be reconstructed accordingly (line 14).

In the solution restart phase, when $NoImproveStep$ exceeds $MaxNoImprove$, meaning that the algorithm falls into the local optima, $NoImproveStep$ is reset and the candidate solution D and D_{last} are reconstructed (lines 16–17). Then, the spanning tree T is rebuilt accordingly (line 18).

4 Restart Based Smoothing Mechanism

In the solution restart phase of NestedLS, an important component is called restart based smoothing mechanism (*ReSmooth*), which restarts the algorithm by constructing a new solution when falling into the local optima.

4.1 Inheriting Scoring Function

In the solution restart phase, starting from an empty candidate set, vertices are iteratively added to the candidate solution by some strategy, until its weight exceeds the best solution or all vertices are dominated. During this phase, if a pure random procedure is applied to generate an initial solution, the initial solution will fail to inherit previous search information. This may make the algorithm deviate from the promising search space and thus degrade the convergence rate of local search.

To hand this issue, two kinds of search information need to be considered.

The first information is the accumulated non-dominated information, represented by $score_f$. The second essential information is “the high-quality solution”, from which the vertices should be selected with higher priority than others. To make full use of the two kinds of search information above, we define a novel scoring function called inheriting scoring function, denoted as $score_{inher}$ as follows.

$$score_{inher}(v) = \begin{cases} score_f(v) \times \beta, & v \notin D^* \cup D_{last_best} \\ score_f(v), & v \in D^* \cup D_{last_best} \end{cases}$$

In the above equation, “the high-quality solution” refers to D^* and D_{last_best} which denotes the solution dominating most vertices since last solution restart phase. If all vertices are dominated by D_{last_best} , then D_{last_best} is equal to D^* . Parameter β denotes the penalty coefficient. Based on this scoring function, we propose the novel selection rule.

Inheriting-Based Selection Rule: Choose the vertex with the greatest $score_{inher}$ value, breaking ties randomly.

4.2 Smoothing Mechanism

We observe that the initial candidate solution may converge after several solution restart phases. The main reason is that $freq$ values of some vertices accumulate to a large amount, leading the algorithm to follow the previous search trajectory and then explore some recently visited search spaces.

To avoid such phenomenon, $freq$ should be smoothed when the initial solutions converge. Thus, NestedLS employs a weight smoothing scheme which resembles SWT [4] in some respect. First, we introduce the Jaccard index [12] to illustrate the repeating rate of solutions.

► **Definition 2.** The repeating rate between the initial solution of last restart D_{last} and D is defined by the Jaccard index: $J(D, D_{last}) = |D \cap D_{last}| / |D \cup D_{last}|$.

When $J(D, D_{last})$ exceeds a threshold $MaxRepeat$, indicating that the initial solutions converge, the $freq$ values of all vertices are smoothed as follows.

$$freq[v] = \rho \cdot freq[v] + (1 - \rho) \cdot \overline{freq}, \quad \forall v \in V$$

where \overline{freq} is the average value of $freq$ and ρ is the smoothing parameter. After smoothing all $freq$ values, score values will be updated accordingly. Experiments on classic benchmark show that the average repeating rate without smoothing is on average 0.69 after calling the *ReSmooth* 100 times, which confirms that without the smoothing method, the initial candidate solution may converge.

■ **Algorithm 2** *ReSmooth*(D_{last}, D^*).

Input: The solution after last construction D_{last}, D^*
Output: A restart candidate solution D

- 1 choose a random node $v \in V$ and $D := \{v\}$;
- 2 **while** $|unDom_G(D)| \neq 0$ and $w(D) < w(D^*)$ **do**
- 3 choose $v \in candAdd(D)$ based on **Inheriting-based Selection Rule** and
 $D := D \cup \{v\}$;
- 4 **if** $J(D, D_{last}) > MaxRepeat$ **then**
- 5 $freq[v] = \rho \cdot freq[v] + (1 - \rho) \cdot \overline{freq}$, for $\forall v \in V$;
- 6 **return** D ;

4.3 The *ReSmooth* Algorithm

The *ReSmooth* is described in Algorithm 2. A random vertex is first added into the empty candidate solution (line 1). Then, vertices are chosen to the candidate solution D based on the inheriting-based selection rule, until all vertices are dominated, or the weight of D exceeds that of the best solution ever (lines 2–3). The *freq* values are smoothed if the repeating rate exceeds the threshold *MaxRepeat* (lines 4–5). Finally, the restart candidate solution D is returned (line 6).

5 Inner-layer local search

In the tree reconstruction and solution restart phases when handling massive graphs, in order to enlarge *candRem*, an important component called inner-layer local search *InnerSearch* is proposed to rebuild a corresponding spanning tree. Also, it can be modelled as a weighted max-leaf spanning tree problem, which is an interesting version of classic spanning tree problem [7].

For current solution D , $G[D] = (V_D, E_D)$ and T denote its subgraph and corresponding spanning tree. $LS(T)$ is the leaf set of T , which serves as *candRem*(D), while $TS(T) = D \setminus LS(T)$ denotes the trunk set where vertices are forbidden to be removed during the vertices swapping phase.

5.1 Motivation for Inner-layer Local Search

Before constructing a new spanning tree, we first formally define the weighted max-leaf spanning tree problem (WMST).

► **Definition 3.** Given a graph $G = (V, E, w)$, the weighted max-leaf spanning tree problem is to find a spanning tree of G with the maximum total weight of leaf set, that is, the minimum total weight of trunk set.

For any spanning tree T of solution D , its trunk set $TS(T)$ is connected and connects to all leaf vertices in $LS(T)$. Thus, WMST can be converted to find a MWCDS of $G[D]$, serving as the trunk set $TS(T)$. We propose an *InnerSearch* method to construct a CDS as $TS(T)$, and then further improve its quality by the local search procedure. To define the scoring function for obtaining $TS(T)$, we propose three intuitions whose importance is displayed in descending order.

- (1) The first intuition is that there should be more candidate removal vertices to enlarge the search space. Moreover, vertices with large weight value should be more likely to be removed to lower $w(D)$. During the vertices swapping phase, $CandRem(D) = LS(T)$ when solving massive graphs. In order to implement the above intuition, there should be more leaf vertices, and vertices with large weight values should be maintained in $LS(T)$. Specifically, we employ a simplified version of $score_f$ as the main scoring function of solving WMST, denoted as $score'_f$ with respect to $TS(T)$. Given a graph $G[D]$, if $u \in TS(T)$, $score'_f(u) = -|C_1(u)|/w(u)$ and otherwise $score'_f(u) = |C_2(u)|/w(u)$, where $C_1(u)$ and $C_2(u)$ have already been defined in Section 2.2.
- (2) Our second intuition is that vertices which intensively degrade the quality of D if deleted, should be forbidden to be removed, and thus they should be excluded from leaf set. To achieve it, the direct way is that the $score_f$ of leaf vertices should be higher, while the $score_f$ of trunk vertices should be lower. This means that vertices with lower $score_f$ are preferred to be left in $TS(T)$.
- (3) The third intuition is that the leaf set should differ from previous ones, so that the algorithm can have more different removing options. To achieve this, vertices with higher exchanging frequency of operations (i.e., to be moved during the vertices swapping), denoted as $score_e$, are preferred to be left in the trunk set. Since those vertices are frequently set as leaf vertices since last construction, leaving them in the trunk set can make the leaf set differ from the previous one.

It is important to notice that during the *InnerSearch* procedure, the $score'_f$ values will be dynamically updated, while the corresponding $score_f$ values keep unchanged because the corresponding $score_f$ is based on D that remains unchanged in this procedure. For $v \in D$, $score_f(v)$ is always no larger than 0. Based on these three intuitions, we propose the novel selection rule for constructing $TS(T)$ as follows.

WMST Selection Rule: Select an added (or removed) vertex with the greatest $score'_f$, breaking ties by picking one with the highest (or lowest) $|score_f|$ value. Further ties are broken by choosing one with the highest (or lowest) $score_e$.

5.2 The *InnerSearch* Algorithm

The pseudo code of *InnerSearch* is shown in Algorithm 3. The algorithm first constructs a CDS of $G[D]$ called D' , serving as the trunk set of $G[D]$, by greedily adding vertices until it becomes a feasible solution (lines 1–3), similar to the *ReSmooth* procedure, and then the spanning tree T' of D' is built by breadth first search (line 5). The loop iterates until it fails to find a better solution within *MaxNoImproveInner* steps (line 6). During each loop, local search is applied by iteratively swapping vertices based on the WMST selection rule to improve D' (lines 7–10). At the end of each loop, the corresponding spanning tree T' needs to be updated (line 11). After the loop, the spanning tree T of D is constructed by adding the remaining vertices in $D \setminus D'$ to T' by using the adding rule of TBC method [13] (line 15). At last, the new spanning tree T is returned (line 16).

Note that to lower the complexity, the best solution during *InnerSearch* is not recorded, and an approximated best solution D' is obtained by setting *MaxNoImproveInner* to a small value. In *InnerSearch*, the complexity of each iteration (lines 6–14) is $O(neighborSize * \Delta_{G[D]})$, while the complexity of remaining parts is $O(|V_D| * \Delta_{G[D]} + |E_D|)$. Since D only accounts for 13.07% of vertices of the original graph on average, *InnerSearch* can be seen as a lightweight local search procedure, compared to Algorithm 1.

Algorithm 3 *InnerSearch*($G[D]$).

Input: a subgraph $G[D]$ induced by candidate solution D
Output: a spanning tree T of $G[D]$

- 1 choose a random vertex $v \in G[D]$ and $D' := \{v\}$;
- 2 **while** $|unDom_{G[D]}(D')| \neq 0$ **do**
- 3 choose vertex $v \in N_{G[D]}(D')$ using **WMST selection rule** and $D' := D' \cup \{v\}$;
- 4 $MinWeight := w(D')$ and $InnerStep := 1$;
- 5 construct a spanning tree T' of D' ;
- 6 **while** $InnerStep < MaxNoImproveInner$ **do**
- 7 **for** $i := 1$ **to** $neighborSize$ **do**
- 8 choose vertex $u \in LS(T')$ using **WMST selection rule** and $D' := D' \setminus \{u\}$;
- 9 **while** $|unDom_{G[D]}(D')| \neq 0$ **do**
- 10 choose vertex $v \in candAdd(D')$ using **WMST selection rule** and
 $D' := D' \cup \{v\}$;
- 11 update the spanning tree T' based on D' ;
- 12 **if** $w(D') < MinWeight$ **then**
- 13 $MinWeight := w(D')$ and $InnerStep := 1$;
- 14 **else** $InnerStep := InnerStep + 1$;
- 15 construct T where $TS(T) = T'$ and $LS(T) = D \setminus D'$;
- 16 **return** T ;

6 Experimental Results

6.1 Experiment Preliminaries

Extensive experiments are carried out to evaluate the performance of NestedLS, compared with four state-of-the-art heuristic algorithms, including HGA [6], PBIG [6], ACO-RVNS [3] and ACO-e, which was modified by the author of ACO-RVNS, specialized for massive graphs. Since the source or binary codes of HGA and PBIG were not available, we reimplemented and then compared to them. The source code of ACO-RVNS and ACO-e were kindly provided by authors. The data structure of all competitors was modified for massive graphs. Specifically, the adjacency list are applied to store the graph information. NestedLS and its competitors were implemented in C++ and compiled by g++ with '-O3'. All experiments were run on a server with Intel Xeon CPU E7-8850 v2 2.30GHz with 2048GB RAM under Ubuntu 16.04.5. All algorithms were executed 10 times with random seeds from 1 to 10 on each instance independently. The cutoff time was set to 1000 seconds for the classic benchmarks, and 5000 seconds for massive graphs. We report the best size (*min*) and average size (*avg*) of the solution found by each algorithm. The bold values indicate the best solution among all the algorithms.

The parameters of NestedLS are tuned by irace [15]. We select 40 graphs randomly from all benchmarks, and irace was applied for 5000 s with a budget of 10000 applications. The chosen values of parameters are presented in Table 1. Moreover, the parameters of all competitors are also tuned by irace, and our re-implementation versions can obtain similar performance as the original papers, which confirms their effectiveness and efficiency.

We evaluate NestedLS on 5 benchmarks, including 2 classic benchmarks in the literature and 3 massive benchmarks.

■ **Table 1** Parameter tuning.

Parameter	Domain	Chosen value
<i>neighborSize</i>	{1,3,5}	3
<i>MaxNoImprove</i>	{10000,50000,100000}	100000
<i>MaxNoImproveInner</i>	{1000,5000}	1000
<i>TreeNoImprove</i>	{5000,10000}	10000
<i>MaxRepeat</i>	{0.1,0.3,0.5}	0.3
ρ	{0.3,0.7}	0.7
β	{0.5,0.7,0.9}	0.7

■ **Table 2** Experiment results on the first classic benchmark. The averaged value of $\min(\overline{min})$ and the number of connected instances with the same size ($\#inst$) are reported for each family.

Instance Family	$\#inst$	NestedLS \overline{min}	PBIG \overline{min}	ACO-e \overline{min}	ACO-RVNS \overline{min}	HGA \overline{min}	Instance Family	$\#inst$	NestedLS \overline{min}	PBIG \overline{min}	ACO-e \overline{min}	ACO-RVNS \overline{min}	HGA \overline{min}
TYPEI							V800E10000	1	2059	2080	2111	2076	2442
V250E750	7	2833	2850.3	2836.4	2833	3068.1	V1000E5000	1	6538	6762	6652	6668	7281
V250E1000	9	2038	2056.8	2039.1	2038	2227.8	V1000E10000	1	2989	3013	3052	3029	3531
V250E2000	10	965.9	974	968.7	965.9	1090.3	V1000E15000	1	2164	2178	2189	2189	2434
V250E3000	10	650.4	653.1	653	650.4	744.3	V1000E20000	1	1612	1639	1645	1616	1800
V250E5000	10	390.2	392.3	391.5	390.9	433.9	TYPEII						
V300E750	2	4272.5	4242.4	4283.5	4272.5	4449.5	V250E750	6	877.5	896.5	877.5	876.8	924.7
V300E1000	9	3067.9	3111	3076.2	3068.2	3315.4	V250E1000	9	953.7	956.2	958	953.9	1014.6
V300E2000	10	1439.4	1457.5	1444.7	1439.4	1639.4	V250E2000	10	1159.9	1161.7	1163.6	1159.9	1272.9
V300E3000	10	936.1	942.2	939.5	936.3	1066.4	V250E5000	10	1469.8	1471.9	1471.8	1469.8	1601.5
V300E5000	10	555.1	561.1	557.6	556.9	634.9	V300E750	1	974	981	979	974	999
V500E2000	1	4179	4239	4183	4182	4579	V300E1000	9	1037.7	1054.6	1040.6	1037.7	1092.6
V500E5000	1	1565	1571	1580	1565	1748	V300E2000	10	1276.3	1287.6	1279.4	1276.4	1395.6
V500E10000	1	852	852	868	852	922	V300E5000	10	1612.9	1618.9	1613	1612.9	1882.5
V800E5000	1	4178	4321	4223	4205	4740							

The first classic benchmark originally from [19] is classified into Type I (96 instances) and Type II (65 instances). There are a few unconnected graphs in the benchmark, and we choose to ignore them. The second classic benchmark (20 instances) is originally generated in [6]. To save space, we do not report the results on graphs with less than 250 vertices where NestedLS always performs best. In total, we selected 181 classic instances.

A total of 118 massive real-world graphs are selected from the Network Data Repository (NDR) [17] and Stanford Large Network Dataset Collection (SNAP)³, as well as large instances from the 10th DIMACS implementation challenge (DIMACS10)⁴. Due to space limitations, we only report results on graphs from the SNAP and DIMACS10 benchmarks with at least 30,000 vertices and graphs from the NDR benchmark with more than 100,000 vertices and more than 1,000,000 edges. Hence, we picked 22, 31 and 65 graphs in SNAP, DIMACS10, and NDR, respectively. To obtain the corresponding weighted instances, we used the same method as in previous works [24, 25]: for the i th vertex v_i , $w(v_i)=(i \bmod 200)+1$.

³ <http://snap.stanford.edu/data>⁴ <https://www.cc.gatech.edu/dimacs10/>

■ **Table 3** Experiment results on the second classic benchmark.

Instance	NestedLS <i>min(avg)</i>	PBIG <i>min</i>	ACO-e <i>min(avg)</i>	ACO-RVNS <i>min(avg)</i>	HGA <i>min</i>
V250E500	4464(4464)	4585	4464(4479.6)	4464(4469.2)	4716.1
V250E1000	2203.5(2203.5)	2228	2227.4(2227.5)	2211.8(2213.1)	2389
V250E1500	1365.7(1365.7)	1384	1365.7(1365.7)	1365.7(1365.7)	1548.1
V250E2000	1020.3(1020.3)	1044.9	1020.3(1026.7)	1020.3(1020.3)	1104.9
V250E2500	822.1(822.1)	822.1	822.1(822.1)	822.1(822.1)	960.3
V500E1000	8636.7(8637.1)	8837.3	8646.69(8679.2)	8637.2(8648)	9444.3
V500E2000	4256(4256)	4352	4296.1(4340)	4277.9(4294.4)	4693.7
V500E3000	2867.2(2867.3)	2915.8	2895.1(2927.8)	2875.7(2875.7)	3256.9
V500E4000	2145.7(2145.7)	2164.3	2157.1(2176.6)	2157.1(2170.2)	2434.5
V500E5000	1531.6(1531.6)	1538	1531.6(1541.8)	1531.6(1531.6)	1766.5
V750E1500	13894.9(13903.7)	14298.5	14042.2(14101.7)	13984.6(14031.4)	15491.2
V750E3000	6106.7(6110.9)	6250.9	6209.7(6252.7)	6154.6(6173.3)	6979.4
V750E4500	4244.4(4244.4)	4383.5	4330.6(4398.8)	4308.7(4328.1)	4674.7
V750E6000	3151.7(3152.9)	3188.9	3167.7(3180.1)	3163.1(3163.1)	3505.6
V750E7500	2401.8(2402.5)	2435	2451.2(2469)	2434.1(2434.7)	2744.8
V1000E2000	17745.5(17768.1)	18235.3	17838.3(17922.3)	17845(17889.3)	19786.5
V1000E4000	8222.8(8222.8)	8453.3	8328.6(8360.6)	8319.7(8335.8)	9532
V1000E6000	5247.9(5250.4)	5341.9	5332(5372.1)	5301.2(5319.2)	5938.7
V1000E8000	3906.2(3910.7)	3983.5	3955.5(4012.7)	3931.1(3956.5)	4465.1
V1000E10000	3106.6(3108.7)	3154.8	3187.2(3201.3)	3119.2(3150.9)	3683.4

6.2 Results on Classic Benchmarks

Results on classic benchmarks are reported in Tables 2 and 3. NestedLS is better than all competitors, except for V250E750, indicating its robustness. The average run time of NestedLS on some instances where it can generate the same solution quality (i.e., same minimal and average values) as PBIG, ACO-e and ACO-RVNS is 16.3 s, 27.3 s and 11.6 s, respectively, while that of competitors is 5.2 s, 87 s and 15 s.

6.3 Results on Massive Graphs

Note that ACO-RVNS and HGA fail to find a solution on most massive instances, mainly due to their high complexity heuristics (i.e., RVNS and Minimize functions). Thus, we mainly report the results of NestedLS, PBIG and ACO-e on Tables 4 and 5. NestedLS significantly outperforms all competitors on most instances, with only 8 exceptions. Moreover, NestedLS can solve all the 118 instances within the time limit, while PBIG, ACO-e, ACO-RVNS and HGA can only solve 103, 47, 19 and 13 instances, respectively. Among all the instances solvable by NestedLS and a corresponding competitor, the best solution obtained by NestedLS is on average 4.18%, 1.37%, 1.08% and 1.39% better than that found by PBIG, ACO-e, ACO-RVNS and HGA, respectively. Since the weight value can amount to 10^8 on some massive graphs, they are significant improvements.

39:12 Improving Local Search for Minimum Weighted Connected Dominating Set Problem

■ **Table 4** Experiment results on SNAP and DIMACS10 benchmarks. If an algorithm fails to find a solution within the cutoff time, it is indicated by “N/A”.

Instance	NestedLS <i>min(avg)</i>	PBIG <i>min(avg)</i>	ACO-e <i>min(avg)</i>
Amazon0302	3607951(3628251.9)	3898308(3903172)	3702466(3702466)
Amazon0312	4201432(4215293.4)	4397464(4402398.5)	N/A
Amazon0505	4383032(4393754.3)	4576088(4579237)	N/A
Amazon0601	3780727(3792534.5)	3987798(3989201.8)	N/A
Cit-HepPh	247185(247337.8)	255517(255731)	253099(253493.8)
Cit-HepTh	256033(256647.4)	263235(263496.3)	260885(261364)
cit-Patents	59497255(59657281.8)	64161977(64167456)	N/A
Email-EuAll	228890(228936.4)	228935(228954)	228951(228975.5)
p2p-Gnutella04	210746(210813.7)	211570(211610.5)	211153(211304.3)
p2p-Gnutella25	451125(451178.5)	451333(451426.5)	451056 (451208.4)
p2p-Gnutella30	711958(712177.5)	712094(712192.5)	711915 (712066.5)
p2p-Gnutella31	1262834(1263059.3)	1262095(1262181.3)	1262676(1262869.1)
Slashdot0811	1460128(1460346.5)	1461470(1461546.8)	1461427(1461427)
Slashdot0902	1580606(1580816.2)	1583119(1583276.5)	1582395(1582395)
soc-Epinions1	1663107(1663222.1)	1663776(1663911.3)	1664085(1664085)
web-BerkStan	2936734(2939268.8)	2987292(2989303.8)	2934995(2934995)
web-Google	7864164(7868993.6)	7985154(7985584)	N/A
web-NotreDame	2495507(2496448.9)	2519655(2520284)	2497288(2497288)
web-Stanford	980121(980582.7)	1007522(1008040.5)	987255(987255)
Wiki-Vote	107222(107227.9)	107222(107223.5)	107234(107249.3)
WikiTalk	3478539(3478544.5)	3478560(3478579)	N/A
333SP	95311299(96023116.9)	104222969(104222969)	N/A
as-22july06	193529(193542.3)	193557(193560.8)	193562(193581.1)
audikw1	544788(546375.8)	645510(646619.3)	561656(561656)
belgium.osm	117292752(117713316.1)	N/A	N/A
cage15	18904266(18939673.7)	22288856(22296289.5)	N/A
caidaRouterLevel	4324957(4327477.8)	4376005(4377893.3)	N/A
citationCiteseer	4434164(4439087.7)	4525350(4527033.5)	4466529(4466529)
cnr-2000	2443499(2444996)	2457027(2457309.8)	2449713(2449713)
coAuthorsCiteseer	3701461(3702751.2)	3717505(3718382.5)	N/A
coAuthorsDBLP	4738187(4739697.2)	4765060(4765668.3)	4749845(4749845)
cond-mat-2005	482173(482484.3)	487804(488072)	486812(487278.3)
coPapersCiteseer	2840116(2848253.2)	2928376(2929221)	N/A
coPapersDBLP	3779813(3790462.5)	3883208(3885354)	N/A
ecology1	37169194(37512291.1)	40995892(41068701.3)	N/A
eu-2005	3186216(3187215.3)	3212190(3212849.5)	3193332(3193332)
G_n_pin_pout	706058(707810.3)	793613(797417.8)	745885(745885)
in-2004	8493855(8495948.8)	8540255(8542346.3)	N/A
kron...logn16	369629(369629)	370490(370553.5)	370386(370495.5)
ldoor	2130615(2131629.9)	2607563(2615331.8)	N/A
luxembourg.osm	9954051(9955957.2)	10123554(10232006.8)	N/A
pref...Attachment	544964(545494.2)	582867(583472.8)	564066(564066)
rgg_n_2_17_s0	1143351(1145682.7)	1411146(1422660.5)	1194638(1194638)
rgg_n_2_19_s0	3619945(3623934.4)	4882585(4902738)	N/A
rgg_n_2_20_s0	6597646(6612833.5)	9305396(9391407.5)	N/A
rgg_n_2_21_s0	12315149(12359474.8)	17639374(17775821.8)	N/A
rgg_n_2_22_s0	27505305(27607164.4)	33784024(34175561.5)	N/A
rgg_n_2_23_s0	50168656(63767170.7)	N/A	N/A
smallworld	1218021(1221583.3)	1311258(1312652.3)	1281937(1281937)
uk-2002	114212809(117625999.3)	113849945(113854708.5)	N/A
wave	975601(978701.5)	1082203(1083760)	999481(999481)

■ **Table 5** Experiment results on NDR benchmark. If an algorithm fails to find a solution within the cutoff time, it is indicated by “N/A”.

Instance	NestedLS <i>min(avg)</i>	PBIG <i>min(avg)</i>	ACO-e <i>min(avg)</i>
bn-human...1-bg	231444(232173.4)	248647(248983.5)	234886(234886)
bn-human...2-bg	193908(194530.6)	206436(206702)	197614(197614)
ca-coauthors-dblp	3780154(3790227.3)	3883208(3885354)	N/A
ca-dblp-2012	4898659(4900170.1)	4931550(4932246.8)	4912016(4912016)
ca-hollywood-2009	4196974(4208205.8)	4484581(4485353.5)	N/A
channel...b050	33862787(33944666)	37854483(37974114)	N/A
dbpedia-link	153458727(153739853.3)	154088350(154089188)	N/A
delaunay_n22	95435272(95559053.8)	104855386(105650155.3)	N/A
delaunay_n23	188365284(188544203.8)	N/A	N/A
delaunay_n24	379521881(408693281.8)	N/A	N/A
friendster	63527982(63557140.7)	64653832(64656091.5)	N/A
hugebubbles-00020	970833598(1202159141.6)	N/A	N/A
hugetrace-00010	554859414(566474478.5)	N/A	N/A
hugetrace-00020	731497084(792040454.8)	N/A	N/A
inf-europe_osm	5092357075(5094787915.4)	N/A	N/A
inf-germany_osm	941456751(942923953.3)	N/A	N/A
inf-road-usa	2375849346(2377787146.5)	N/A	N/A
inf-roadNet-CA	92151724(92854875.3)	98142433(98369828.5)	N/A
inf-roadNet-PA	50457051(50693249.2)	54112058(54182813.5)	N/A
rec-dating	1137467(1137484.5)	1138910(1139023.8)	1138531(1138531)
rec-epinions	826618(826642.9)	831768(832227)	829707(829707)
rec-libimseti-dir	1209219(1209288.3)	1213842(1214225)	1212387(1212387)
rgg_n_2_23_s0	50329249(50441454.7)	N/A	N/A
rgg_n_2_24_s0	518533632(713025008.5)	N/A	N/A
rt-retweet-crawl	8119952(8120894.8)	8112459(8112605.3)	N/A
sc-lldoor	2148767(2153395.2)	2608260(2628863.8)	N/A
sc-msdoor	853236(854334.5)	1006625(1011167)	N/A
sc-pwtk	441580(442377.9)	602802(605173.5)	451326(451326)
sc-rel9	12466895(12494110.1)	13371415(13373856.3)	N/A
sc-shipsec1	587596(589711.3)	673591(675410.8)	607640(607640)
sc-shipsec5	737132(741136.8)	840811(849266)	762199(762199)
soc-buzznet	8275(8275)	8373(8381.5)	8337(8386.8)
soc-delicious	5684064(5685039.5)	5689449(5689994.5)	5683212(5683212)
soc-digg	6884347(6888681.2)	6906274(6906978.3)	N/A
soc-dogster	2343228(2344555.2)	2373262(2373356)	2352360(2352360)
soc-flickr-und	29310795(29333534)	29701624(29702432)	N/A
soc-flxster	9190111(9190239.9)	9189919(9190039.3)	N/A
soc-FourSquare	6055451(6058625.3)	6063201(6064206)	6062299(6062299)
soc-lastfm	6747994(6748217.7)	6748666(6748975.3)	N/A
soc-livejournal	80381637(80396298.8)	83450152(83455918.8)	N/A
soc-...-user-groups	109130362(109143584)	109708034(109708334)	N/A
soc-LiveMocha	106551(106560.2)	108182(108286.5)	107712(107846.8)
soc-ljournal-2008	103641550(103684264.4)	105872796(105877923.8)	N/A
soc-orkut	8377576(8436848.5)	9246155(9249442.5)	N/A
soc-orkut-dir	7371792(7388939.2)	8257778(8260096.8)	N/A
soc-pokec	18650680(18678287.5)	19938844(19941250)	N/A
soc-sinaweibo	5894908130(5894908130)	N/A	N/A
soc-twitter-higgs	1160854(1161304)	1184020(1185051.3)	1170510(1170510)
soc-youtube	9898687(9900778.7)	9936591(9937572.8)	N/A
soc-youtube-snap	23382235(23384447.6)	23408462(23585903.3)	N/A
socfb-A-anon	19919414(19952815.8)	20350881(20351694.5)	N/A
socfb-B-anon	18669945(18697816.9)	18997889(18999053)	N/A
socfb-uci-uni	5866001161(5866001161)	N/A	N/A
tech-as-skitter	17726432(17747980.9)	18668301(18669829)	N/A
tech-ip	2283(2283.5)	2986(3010)	2484(2484)
twitter_mpi	56327895(56337886.8)	56435803(56436632)	N/A
web-arabic-2005	2017151(2017601.7)	2021106(2022620)	2021129(2021129)
web-baidu-baike	25951517(25969911.1)	26457056(26457712.3)	N/A
web-it-2004	3464760(3464814.9)	3465855(3465914)	N/A
web-uk-2005	170958(170958)	170958(170958.8)	170958(170958)
web-wikipedia_link	17428644(17452302.6)	17888836(17889610.3)	N/A
web-wikipedia-growth	10192627(10212490.8)	10592826(10594605.3)	N/A
web-wikipedia2009	37603865(37659492.6)	38742158(38746820.3)	N/A
wikipedia_link_en	21240536(21242706.8)	21362465(21363129.3)	N/A

6.4 Effectiveness of Proposed Strategies

To confirm the effectiveness of *ReSmooth*, we compare NestedLS with its modified versions where NoSmooth does not use this strategy and adopt previous weight smoothing mechanisms SWT and PAWS from [4] and [20], respectively.

The excellent results of NestedLS on massive graphs are mainly due to the inner-layer local search. To confirm its effectiveness, five modified versions are proposed for comparison as follows.

- To confirm the overall effectiveness of inner-layer local search, Alg₁ replaces inner-layer local search with breadth first search to construct the spanning tree for the current candidate solution, as the traditional construction method in [13].
- To confirm the effectiveness of the scoring function in inner-layer local search, Alg₂ and Alg₃ modifies inner-layer local search by not applying the second and third scoring criterion respectively.
- To confirm the effectiveness of WMST selection rule, Alg₄ adopts the same selection rule mentioned as in Section 2.
- To confirm that local search can improve the quality of the spanning tree, Alg₅ constructs the spanning tree without improving it by local search.

The results are shown in Tables 6 and 7. We report the number of instances where NestedLS finds better (worse) solutions than its modified versions, denoted as #better (#worse). The results shown in Table 6 confirm that *ReSmooth* is effective on both classic and massive graphs, and the results shown in Table 7 validate the effectiveness of inner-layer local search on massive graphs.

■ **Table 6** Effectiveness of *ReSmooth*.

		Classic	SNAP	DIMACS	NDR
vs. NoSmooth	#better	59	16	20	37
	#worse	0	3	5	21
vs. SWT	#better	50	17	19	43
	#worse	0	0	5	14
vs. PAWS	#better	14	15	25	52
	#worse	3	6	5	10

■ **Table 7** Effectiveness of *InnerSearch*.

		vs. Alg ₁	vs. Alg ₂	vs. Alg ₃	vs. Alg ₄	vs. Alg ₅
SNAP	#better	17	16	15	19	15
	#worse	4	5	0	2	6
DIMACS	#better	27	22	18	26	23
	#worse	2	8	11	3	6
NDR	#better	41	41	50	56	48
	#worse	17	20	11	5	12

7 Conclusion

We proposed a local search algorithm NestedLS for MWCDS based on two main ideas, including the restart based smoothing mechanism and the inner-layer local search method. Experiments on classic benchmarks and massive graphs showed its superiority over previous algorithms for MWCDS.

Two proposed ideas can be generally applied to other heuristic algorithms. Specifically, the inner-layer local search method is a general method for maintaining the connectivity constraint when dealing with massive graphs. It contributes to constraint programming by providing not only a better strategy of maintaining the connectivity constraint when dealing with massive instances, but also insights for future study on the connectivity constraints. In addition, the restart based smoothing mechanism provides a novel diversification scheme for restart-based heuristic algorithms.

References

- 1 Jamal N Al-Karaki and Ahmed E Kamal. Efficient virtual-backbone routing in mobile ad hoc networks. *Computer Networks*, 52(2):327–350, 2008.
- 2 Christoph Ambühl, Thomas Erlebach, Matúš Mihalák, and Marc Nunkesser. Constant-factor approximation for minimum-weight (connected) dominating sets in unit disk graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 3–14. Springer, 2006.
- 3 Salim Bouamama, Christian Blum, and Jean-Guillaume Fages. An algorithm based on ant colony optimization for the minimum connected dominating set problem. *Applied Soft Computing*, 80:672–686, 2019.
- 4 Shaowei Cai and Kaile Su. Local search for boolean satisfiability with configuration checking and subscore. *Artif. Intell.*, 204:75–98, 2013.
- 5 Orhan Dagdeviren, Kayhan Erciyes, and Savio Tse. Semi-asynchronous and distributed weighted connected dominating set algorithms for wireless sensor networks. *Computer Standards & Interfaces*, 42:143–156, 2015.
- 6 Zuleyha Akusta Dagdeviren, Dogan Aydin, and Muhammed Cinsdikici. Two population-based optimization algorithms for minimum weight connected dominating set problem. *Appl. Soft Comput.*, 59:644–658, 2017.
- 7 Tetsuya Fujie. An exact algorithm for the maximum leaf spanning tree problem. *Computers & Operations Research*, 30(13):1931–1944, 2003.
- 8 Fred Glover, Manuel Laguna, et al. Handbook of combinatorial optimization. *Springer*, pages 2093–2229, 1998.
- 9 Sudipto Guha and Samir Khuller. Approximation algorithms for connected dominating sets. *Algorithmica*, 20(4):374–387, 1998.
- 10 John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.
- 11 Raka Jovanovic and Milan Tuba. Ant colony optimization algorithm with pheromone correction strategy for the minimum connected dominating set problem. *Computer Science and Information Systems*, 10(1):133–149, 2013.
- 12 Michael Levandowsky and David Winter. Distance between sets. *Nature*, 234(5323):34–35, 1971.
- 13 Bohan Li, Xindi Zhang, Shaowei Cai, Jinkun Lin, Yiyuan Wang, and Christian Blum. Nucds: An efficient local search algorithm for minimum connected dominating set. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pages 1503–1510, 2020.

- 14 Jiawei Li, Xiangxi Wen, Minggong Wu, Fei Liu, and Shuangfeng Li. Identification of key nodes and vital edges in aviation network based on minimum connected dominating set. *Physica A: Statistical Mechanics and its Applications*, 541:123340, 2020.
- 15 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.
- 16 Tijana Milenković, Vesna Memišević, Anthony Bonato, and Nataša Pržulj. Dominating biological networks. *PloS one*, 6(8):e23016, 2011.
- 17 Ryan Rossi and Nesreen Ahmed. The network data repository with interactive graph analytics and visualization. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 4292–4293, 2015.
- 18 Lu Ruan, Hongwei Du, Xiaohua Jia, Weili Wu, Yingshu Li, and Ker-I Ko. A greedy approximation for minimum connected dominating sets. *Theoretical Computer Science*, 329(1-3):325–330, 2004.
- 19 Shyong Jian Shyu, Peng-Yeng Yin, and Bertrand MT Lin. An ant colony optimization algorithm for the minimum weight vertex cover problem. *Annals of Operations Research*, 131(1-4):283–304, 2004.
- 20 John Thornton, Duc Nghia Pham, Stuart Bain, and Valnir Ferreira Jr. Additive versus multiplicative clause weighting for sat. In *AAAI*, volume 4, pages 191–196, 2004.
- 21 Mustafa Tosun and Elif Haytaoglu. A new distributed weighted connected dominating set algorithm for wsns. In *2018 IEEE 8th International Conference on Consumer Electronics-Berlin (ICCE-Berlin)*, pages 1–6. IEEE, 2018.
- 22 Sattar Vakili and Qing Zhao. Distributed node-weighted connected dominating set problems. In *2013 Asilomar Conference on Signals, Systems and Computers*, pages 238–241, 2013.
- 23 Dingding Wang, Tao Li, and Mitsunori Ogihara. Generating pictorial storylines via minimum-weight connected dominating set approximation in multi-view graphs. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, pages 683–689, 2012.
- 24 Yiyuan Wang, Shaowei Cai, Jiejiang Chen, and Minghao Yin. A fast local search algorithm for minimum weight dominating set problem on massive graphs. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 1514–1522, 2018.
- 25 Yiyuan Wang, Shaowei Cai, Jiejiang Chen, and Minghao Yin. Scwalk: An efficient local search algorithm and its improvements for maximum weight clique problem. *Artificial Intelligence*, 280:103230, 2020.
- 26 Yiyuan Wang, Shaowei Cai, and Minghao Yin. Local search for minimum weight dominating set with two-level configuration checking and frequency based scoring function. *Journal of Artificial Intelligence Research*, 58:267–295, 2017.
- 27 Yu Wang, Weizhao Wang, and Xiang-Yang Li. Efficient distributed low-cost backbone formation for wireless networks. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):681–693, 2006.
- 28 Yu Wang, Weizhao Wang, and Xiang-Yang Li. *Weighted Connected Dominating Set*, pages 1020–1023. Springer US, 2008.
- 29 Feng Zou, Yuexuan Wang, XiaoHua Xu, Xianyu Li, Hongwei Du, Peng-Jun Wan, and Weili Wu. New approximations for minimum-weighted dominating sets and minimum-weighted connected dominating sets on unit disk graphs. *Theor. Comput. Sci.*, 412(3):198–208, 2011.

Automatic Generation of Declarative Models For Differential Cryptanalysis

Luc Libralesso

LIMOS, CNRS UMR 6158, University Clermont Auvergne, Aubière, France

François Delobel

LIMOS, CNRS UMR 6158, University Clermont Auvergne, Aubière, France

Pascal Lafourcade

LIMOS, CNRS UMR 6158, University Clermont Auvergne, Aubière, France

Christine Solnon

INSA Lyon, CITI, INRIA CHROMA, F-69621 Villeurbanne, France

Abstract

When designing a new symmetric block cipher, it is necessary to evaluate its robustness against differential attacks. This is done by computing Truncated Differential Characteristics (TDCs) that provide bounds on the complexity of these attacks. TDCs are often computed by using declarative approaches such as CP (Constraint Programming), SAT, or ILP (Integer Linear Programming). However, designing accurate and efficient models for these solvers is a difficult, error-prone and time-consuming task, and it requires advanced skills on both symmetric cryptography and solvers.

In this paper, we describe a tool for automatically generating these models, called TAGADA (Tool for Automatic Generation of Abstraction-based Differential Attacks). The input of TAGADA is an operational description of the cipher by means of black-box operators and bipartite Directed Acyclic Graphs (DAGs). Given this description, we show how to automatically generate constraints that model operator semantics, and how to generate MiniZinc models. We experimentally evaluate our approach on two different kinds of differential attacks (e.g., single-key and related-key) and four different symmetric block ciphers (e.g., the AES (Advanced Encryption Standard), Craft, Midori, and Skinny). We show that our automatically generated models are competitive with state-of-the-art approaches. These automatically generated models constitute a new benchmark composed of eight optimization problems and eight enumeration problems, with instances of increasing size in each problem. We experimentally compare CP, SAT, and ILP solvers on this new benchmark.

2012 ACM Subject Classification Security and privacy → Cryptanalysis and other attacks

Keywords and phrases Constraint Programming, SAT, ILP, Differential Cryptanalysis

Digital Object Identifier 10.4230/LIPIcs.CP.2021.40

Supplementary Material

Software (Source Code): https://gitlab.limos.fr/iaa_lulibral/tagada/

archived at `swb:1:dir:43b1382c69c9612241160a8bfb9e019e90927539`

Dataset (Models and Results): https://gitlab.limos.fr/iaa_lulibral/experiment-results

archived at `swb:1:dir:f691fc943675263dab923d092f5a6508bae79ff6`

Funding This work has been partially supported by the French government research program “Investissements d’Avenir” through the IDEX-ISITE initiative 16-IDEX-0001 (CAP 20-25) and the IMobS3 Laboratory of Excellence (ANR-10-LABX-16-01) and the French ANR PRC grant MobiS5 (ANR-18-CE39-0019), DECRYPT (ANR-18-CE39-0007), SEVERITAS (ANR-20-CE39-0005).

1 Introduction

Symmetric cryptography provides algorithms for ciphering a text given a secret key. Differential cryptanalysis is a well-known attack technique that aims at checking if the key can be guessed by introducing differences and studying their propagation during the ciphering process [6]. To evaluate the robustness of a new ciphering algorithm towards differential



© Luc Libralesso, François Delobel, Pascal Lafourcade, and Christine Solnon;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 40; pp. 40:1–40:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

attacks, we compute *Truncated Differential Characteristics (TDCs)* as initially proposed by Knudsen in [20], where sequences of bits are abstracted by Boolean values in order to locate differences (without computing their exact values). We first solve an optimization problem (called Step1-opt) that aims at finding a TDC that has a minimal number of differences that pass through non-linear operators. This provides bounds on the complexity of differential attacks, and in some cases these bounds are large enough to ensure security. When bounds are not large enough, we have to solve an enumeration problem (called Step1-enum) that aims at finding all TDCs that have a given number of differences that pass through non-linear operators. Finally, for each enumerated TDC, we have to compute a *Maximum Differential Characteristic (MDC)*, i.e., find difference values that have the largest probability given their positions identified in the TDC. MDCs are then used to design attacks. Computing an MDC given a TDC is a problem that is efficiently tackled by CP solvers (thanks to table constraints) [16]. Step1-opt and Step1-enum are much more challenging problems. They may be solved by using declarative approaches such as CP (Constraint Programming), SAT, or ILP (Integer Linear Programming) [11]. However, designing accurate and efficient models for these solvers is a difficult, error-prone and time-consuming task, and it requires advanced skills in both symmetric cryptography and combinatorial optimization.

Contributions and Overview of the Paper

In this paper, we describe a tool (called TAGADA) that automatically generates MiniZinc models for solving Step1-opt and Step1-enum problems given a cipher description. In Section 2, we introduce a unifying framework for describing symmetric block ciphers by means of elementary operators and bipartite Directed Acyclic Graphs (DAGs) that specify how these operators are combined. In Section 3, we formally define Step1-opt and Step1-enum problems, and we describe existing approaches for solving these problems.

In Section 4, we describe the input format of TAGADA which is based on the framework introduced in Section 2. Operator semantics are specified by functions which may be black boxes extracted from an existing implementation of the cipher. The DAG is specified in a JSON file. As the creation of this file may be tedious, TAGADA includes a set of functions for easing its generation. TAGADA also includes a function for automatically transforming the input description into an operational cipher. Hence, the correctness of the description is tested by comparing the outputs of the automatically generated cipher with the outputs of the original implementation of the cipher.

In Section 5, we describe how TAGADA automatically generates MiniZinc [21] models for computing TDCs. One key point is to define constraints associated with operators. In existing models, these constraints have been crafted by researchers, and some of these constraints require to have advanced knowledge on both symmetric cryptography and mathematical modelling. We show how to automatically generate these constraints from the functions that describe operator semantics. We also automatically improve models by both enriching and shaving the DAG.

In Section 6, we experimentally evaluate these models for two kinds of differential attacks, i.e., single-key and related-key, and four ciphering algorithms, i.e., the AES, Craft, Midori and Skinny. We report results obtained with ILP, SAT and CP solvers. We also compare the automatically generated models with state-of-the-art hand-crafted models, and we show that TAGADA models are competitive with them.

Notations

We denote $[n, m]$ the set of all integer values ranging from n to m . Sequences of bits are denoted by x, y, z, \dots (possibly sub-scripted). The length of a sequence x is denoted $\#x$. The bitwise XOR operator is denoted \oplus . Tuples are denoted t (possibly sub-scripted), and the arity of a tuple t is denoted $\#t$. We denote $[0, 1]^{k \times p}$ the set of all possible tuples of k -bit sequences of arity p . Given two tuples of bit sequences $t = (y_1, \dots, y_n)$ and $t' = (y'_1, \dots, y'_n)$, we denote $t \oplus t'$ the tuple corresponding to $(y_1 \oplus y'_1, \dots, y_n \oplus y'_n)$.

2 Unifying Description of Symmetric Block Ciphers

The best-known symmetric block cipher is the AES (Advanced Encryption Standard), which is the standard for block ciphers since 2001 [12]. There exist many other symmetric block ciphers, that have been designed for previous competitions or the ongoing lightweight cryptography standardization competition organized by the NIST (*National Institute of Standards and Technology*). Some ciphers are designed for devices with limited computational resources, for example: Craft [5], Deoxys [19], Gift [2], Midori [1], Present [8], Skinny [4], Simon and Speck [3].

As our goal is to design a generic tool that automatically generates a model for computing TDCs from the description of a cipher, we describe these ciphers in a unified way, by means of DAGs. This unifying description is our first step towards automatic differential cryptanalysis.

2.1 Ciphering Operators

The encryption of a plaintext is achieved by applying elementary ciphering operators. Each operator o has a tuple of input parameters denoted $t_{in}(o)$ and a tuple of output parameters denoted $t_{out}(o)$ such that each parameter is a bit sequence, i.e., $t_{in}(o) = (x_1, \dots, x_{\#t_{in}(o)})$ and $t_{out}(o) = (y_1, \dots, y_{\#t_{out}(o)}) = o(x_1, \dots, x_{\#t_{in}(o)})$. Without loss of generality, we assume that all bit sequences have the same length k (if this is not the case, we may split sequences so that they all have the same length). Typically, $k = 8$ (resp. $k = 4$) and k -bit sequences correspond to bytes (resp. nibbles).

► **Example 1.** The AES uses four elementary operators that operate on bytes (i.e., $k = 8$):

- XOR, such that $t_{in}(xor) = (x_1, x_2)$, $t_{out}(xor) = (y_1)$, and $y_1 = x_1 \oplus x_2$;
- ShiftRows, denoted SR_s with $s \in [0, 3]$, such that $t_{in}(SR_s) = (x_1, x_2, x_3, x_4)$, $t_{out}(SR_s) = (y_1, y_2, y_3, y_4)$, and $\forall i \in [1, 4], y_i = x_{1+(i+s)\%4}$ where $\%$ is the modulo operation (in other words, SR_s simply shifts the positions of the four input bytes);
- MixColumns, denoted MC , such that $t_{in}(MC) = (x_1, x_2, x_3, x_4)$, $t_{out}(MC) = (y_1, y_2, y_3, y_4)$, and $\forall i \in [1, 4], y_i = (M_{i,1} \otimes x_1) \oplus (M_{i,2} \otimes x_2) \oplus (M_{i,3} \otimes x_3) \oplus (M_{i,4} \otimes x_4)$ where $M_{i,j}$ are constant coefficients, and \otimes is a finite field multiplication;
- SubBytes, denoted S , such that $t_{in}(S) = (x_1)$, $t_{out}(S) = (y_1)$, and y_1 is obtained from x_1 by using a substitution that is represented by a look-up table, called S-Box.

More generally, there are two main categories of operators that ensure two main concepts identified by Shannon in [24]: Non-linear operators that ensure confusion, and linear operators that ensure diffusion. Non-linear operators are either S-Boxes (like the AES SubBytes) or non-linear arithmetic operations (like in ARX¹ structures). The most common linear

¹ ARX schemes use only modular Addition, Rotation and XOR.

operations used in symmetric ciphers are: multiplication by a MDS (Maximum Distance Separable) matrix (like the AES MixColumns), bit permutations, XOR and rotation (like the AES ShiftRows). Every linear operator o satisfies the following property: $\forall t, t' \in [0, 1]^{k \cdot \#t_{in}(o)}, o(t) \oplus o(t') = o(t \oplus t')$.

2.2 Description of a Cipher with a DAG

Given a plaintext and a key, a cipher returns a ciphertext. The plaintext and the key are bit-sequences, and we assume that they have been split into k -bit sequences. The ciphertext is computed by applying operators, and this process may be described by a DAG that contains two different kinds of vertices denoted P and O , respectively: each vertex in P corresponds to a parameter and is a k -bit sequence, whereas each vertex in O corresponds to an operator. Arcs connect operators to their input and output parameters: the predecessors (resp. successors) of an operator o are denoted $pred(o)$ (resp. $succ(o)$) and they correspond to input (resp. output) parameters. As parameters are ordered, $pred(o)$ and $succ(o)$ are tuples (instead of sets) and the order is represented by arc labels: an incoming arc (x, o) (resp. outgoing arc (o, x)) is labelled with $i \in [1, \#t_{in}(o)]$ (resp. $i \in [1, \#t_{out}(o)]$), meaning that x is the i^{th} input (resp. output) parameter in $pred(o)$ (resp. $succ(o)$).

Some input parameters have no predecessor in the DAG. These input parameters either correspond to k -bit sequences that are resulting from the plaintext or the key, or to constant values. The set of input parameters that are constant values is denoted C .

Most ciphers are iterative processes composed of r rounds. This round decomposition does not appear in the DAG as it is not necessary for automatically generating models.

► **Example 2.** We display in Fig. 1 the DAG that describes the first AES round.

3 Optimization and Enumeration of TDCs

We first define MDCs in Section 3.1; then we define TDCs in Section 3.2; and finally, we define the two problems addressed in this paper, Step1-opt and Step1-enum, in Section 3.3.

3.1 Maximum Differential Characteristics

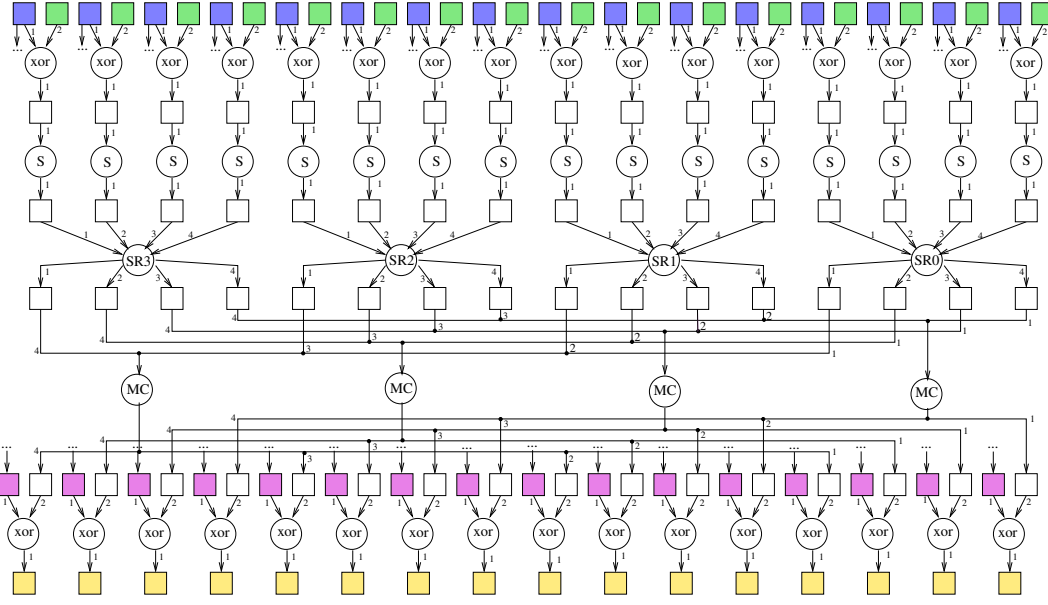
To design differential attacks, we study the propagation of differences during the ciphering process. To introduce differences in a k -bit sequence x , we XOR it with another k -bit sequence x' , and we denote δx the resulting differential sequence, i.e., $\delta x = x \oplus x'$. When $\delta x = 0$, there is no difference (i.e., $x = x'$) whereas when $\delta x \neq 0$ there are differences (i.e., $x \neq x'$). Similarly, we denote δt the differential tuple obtained by XORing the elements of the two tuples t and t' , i.e., $\delta t = t \oplus t'$. By abuse of language, we say that a tuple δt is equal to 0 whenever all its elements are equal to 0, i.e., δt does not contain differences.

Given an operator o , some input/output differences are more likely to occur than others, and this is quantified by means of differential probabilities.

► **Definition 3** (Differential probability of an operator). *The probability that an operator o transforms an input difference δt_{in} into an output difference δt_{out} is*

$$p_o(\delta t_{out} | \delta t_{in}) = \frac{\#\{(t, t') \in [0, 1]^{k \cdot \#t_{in}(o)} \times [0, 1]^{k \cdot \#t_{out}(o)} : \delta t_{in} = t \oplus t' \wedge \delta t_{out} = o(t) \oplus o(t')\}}{2^{k \cdot \#t_{in}(o)}}$$

This probability is equal to 0 or 1 for linear operators. More precisely, for any linear operator o , $p_o(\delta t_{out} | \delta t_{in}) = 1$ if $o(\delta t_{in}) = \delta t_{out}$ and $p_o(\delta t_{out} | \delta t_{in}) = 0$ otherwise. This comes from the fact that for any linear operator o and any input parameters t and t' , $o(t) \oplus o(t') = o(t \oplus t')$.



■ **Figure 1** DAG of the first round of the AES for 128-bit keys. Bytes are represented with squares, and operators with circles. The input key and plaintext have 128 bits and are split into 16 bytes colored in blue and green, respectively. Yellow squares correspond to the text state after one encryption round. Pink squares correspond to the first round sub-key and are obtained from the blue squares by applying operations which are not displayed to avoid overloading the figure (these operations are: 16 XORs, 4 SubBytes, and 1 XOR with a constant).

When an operator o is not linear, p_o may be different from 0 and 1 and the only case where $p_o(\delta t_{out}|\delta t_{in})=1$ is when $\delta t_{in}=\delta t_{out}=0$. In all other cases, it is strictly smaller than 1.

► **Example 4.** For the AES, all operators but SubBytes are linear. For SubBytes, the probability $p_S(\delta t_{out}|\delta t_{in})$ belongs to $\{0, 2^{-6}, 2^{-7}, 1\}$.

Let us now formally define what is an MDC.

► **Definition 5 (MDC).** *Given a DAG that describes a cipher, a differential characteristic is a function $\delta : P \setminus C \rightarrow [0, 1]^k$ that associates a differential sequence δx_i with every non-constant parameter $x_i \in P \setminus C$. The probability of a differential characteristic is obtained by multiplying, for each operator $o \in O$, the probability $p_o(\delta succ(o)|\delta pred(o))$ where δt denotes the tuple obtained by replacing every parameter x_i that occurs in t by δx_i if $x_i \in P \setminus C$, and by 0 if $x_i \in C$.*

An MDC is a differential characteristic with maximum probability.

3.2 Truncated Differential Characteristics

MDCs are usually computed in two steps, as initially proposed by Knudsen in [20]: First, we search for TDCs, and then we compute MDCs associated with TDCs.

A TDC is a solution to an abstract problem. More precisely, the abstraction of a k -bit differential sequence δx is a Boolean value denoted ΔX such that $\Delta X = 1$ iff δx contains a difference, i.e., $\delta x \neq 0$. Similarly, the abstraction of a differential tuple $\delta t = (\delta x_1, \dots, \delta x_i)$ is the Boolean tuple $\Delta t = (\Delta x_1, \dots, \Delta x_i)$ such that Δx_j is the abstraction of δx_j for each $j \in [1, i]$.

► **Definition 6 (TDC).** Given a bipartite DAG that describes a cipher, a TDC is a function $\Delta : P \setminus C \rightarrow \{0, 1\}$ that associates a Boolean value Δx_i with every non-constant parameter $x_i \in P \setminus C$.

A concretization of a TDC Δ is a differential characteristic δ such that, for each non-constant parameter $x \in P \setminus C$, $\Delta x = 0 \Leftrightarrow \delta x = 0$. Δ is concretizable if it has at least one concretization, the probability of which is different from 0.

Finding a concretization of a TDC that has a maximal probability (or proving that the TDC cannot be concretized) is efficiently tackled by CP solvers thanks to table constraints (see, e.g., [16]). However, there exists an exponential number of candidate TDCs with respect to the number of non-constant parameters in $P \setminus C$. Hence, the key point for an efficient solution process is to reduce as much as possible the number of candidate TDCs. This is done by adding constraints that prevent the generation of non concretizable TDCs as much as possible, without removing any concretizable TDC.

► **Example 7 (XOR).** If $\delta y_1 = \delta x_1 \oplus \delta x_2$, then it is not possible to have only one sequence in $\{\delta x_1, \delta x_2, \delta y_1\}$ which contains a difference. Therefore, we can add the constraint $\Delta x_1 + \Delta x_2 + \Delta y_1 \neq 1$ for each XOR operator.

► **Example 8 (MC).** There is no straightforward constraint that may be associated with MC as knowing which input parameters contain differences is not enough to know which output parameters contain differences: To answer this question, we must know the exact values of the input differences. However, MC usually satisfies the MDS property [25] that relates the number of input differences with the number of output differences. The exact definition of this relation depends on the constant coefficients $M_{i,j}$. For the AES, this relation is: among the four input differences $\delta x_1, \dots, \delta x_4$ and the four output differences $\delta y_1, \dots, \delta y_4$, either all differences are equal to 0, or at least five of them are different from 0. Hence, we can add the constraint $\sum_{i=1}^4 \Delta X_i + \Delta Y_i \in \{0, 5, 6, 7, 8\}$ for each MC operator.

► **Example 9 (SR_s).** SR_s simply moves bytes. Therefore, we can add an equality constraint between the corresponding Boolean variables, i.e., $\forall i \in [1, 4], \Delta y_i = \Delta x_{1+(i+s)\%4}$.

► **Example 10 (S).** S is not linear, and we cannot deterministically compute the output difference δy_1 given the input difference δx_1 . However, as the look-up table is a bijection, we know that $\delta x_1 = 0 \Leftrightarrow \delta y_1 = 0$. Therefore, we can add the constraint $\Delta x_1 = \Delta y_1$ for each S operator.

3.3 Definition of Step1-opt and Step1-enum Problems

As the probability $p_o(\delta t_{out} | \delta t_{in})$ associated with a non-linear operator o is equal to 1 whenever $\delta t_{out} = \delta t_{in} = 0$ whereas it is very small otherwise (e.g., smaller than or equal to 2^{-6} for the AES Sbox), we can compute an upper bound on an MDC by computing a lower bound on the number of *active* non-linear operators in a TDC, where an operator is said to be active whenever its input/output differential tuples are different from 0. More precisely, let $s(\Delta)$ be the number of active non-linear operators in a TDC Δ (i.e., $s(\Delta) = \#\{o \in O : o \text{ is not linear} \wedge \delta pred(o) \neq 0\}$), and let s^* be the minimal value of $s(\Delta)$ for all possible TDCs Δ . If the maximal probability of an active non-linear operator is equal to p , then the probability of an MDC is upper bounded by p^{s^*} . For example, for the AES this upper bound is $2^{-6 \cdot s^*}$. In some cases, this upper bound is small enough to ensure the security of the cipher with respect to differential attacks, and it is not necessary to actually compute MDCs. Most papers that introduce new ciphering algorithms demonstrate the security of

their cipher with respect to differential attacks only by computing this upper bound (e.g., [5]). When the upper bound p^{s^*} is large enough to allow mounting differential attacks, we have to enumerate all possible TDCs that have a given number of active non-linear operators, and we have to search for an MDC for each of these TDCs.

Step1-opt is the problem that aims at computing s^* whereas *Step1-enum* is the problem that aims at enumerating all TDCs that have a given number of active non-linear operators.

There exist different kinds of differential attacks, depending on where differences can be injected. In this paper, we consider *Single-key* attacks, where differences are only injected in the clear text (i.e., for each k -bit sequence x_i coming from the input key, we have $\Delta x_i = 0$), and *Related-key* attacks, where differences can be injected in both the plaintext and the key.

3.4 Existing Approaches for Solving Step1-opt and Step1-enum

Two dedicated approaches have been proposed to solve these problems: An approach based on dynamic programming (e.g., for AES [13] and Skinny [11]), and an approach based on Branch & Bound (e.g., for AES [7]). The dynamic programming approach is rather efficient, but it runs out of memory for large instances (e.g., when the key has more than 128 bits for the AES); the Branch & Bound approach has no memory issue but needs weeks to solve middle size instances and cannot be used to solve all instances within a reasonable amount of time.

Also, ILP, CP, or SAT are commonly used to solve Step1-opt and Step1-enum: on Skinny [11], Craft [18], Deoxys [26, 10], AES [23, 16], and Midori [15], for example.

While ILP/CP/SAT approaches require less programming work than dedicated ones, they still require designing mathematical models. In particular, it is necessary to find constraints that limit the number of non concretizable TDCs as much as possible, and this can be time-consuming. In this paper, we present an automatic way to generate models for Step1-opt and Step1-enum.

4 Description of a Symmetric Block Cipher with Tagada

The DAG associated with a cipher (see Section 2) must be described in a JSON file. This file first specifies a list of parameters such that each parameter has one attribute, i.e., its name (which must be unique). Then, it specifies a list of operators such that each operator has three attributes, i.e., its list of input parameters, its list of output parameters, and its UID (a unique identifier) that must correspond to an executable function.

► **Example 11** (JSON representation of a XOR followed by a SubBytes).

```
{ "parameters": [ {"name": "X00"}, {"name": "K00"}, {"name": "ARK00"}, {"name": "S00"} ],
  "operators": [ {"uid": "xor_2_1", "in": ["X00", "K00"], "out": ["ARK00"]},
                 {"uid": "s_1_1", "in": ["ARK00"], "out": ["S00"]} ] }
```

The UIDs `xor_2_1` and `s_1_1` correspond to computable functions: `xor_2_1` reads two k -bit sequences and outputs their XOR, and `s_1_1` reads one k -bit sequence and returns the substitution associated with it according to the S-Box.

Some patterns may be repeated in the DAG. For example, let us consider the DAG describing the first round of the AES displayed in Fig. 1. At the top level of this DAG, there are 16 XORs which correspond to the *AddRoundKey* (ARK) step, where each byte of the text (in blue) is XORed with the corresponding byte of the key (in green). As it is tedious to write 16 times the JSON representation of one XOR operation, TAGADA provides functions corresponding to meta-operators, where a meta-operator is a classical combination of operators.

► **Example 12** (ARK meta-operator). The ARK meta-operator has 3 groups of parameters: the first group corresponds to the 16 input text bytes; the second to the 16 input key bytes; and the third to the 16 output parameters. This meta-operator generates the JSON description of 16 XORs such that each XOR has two input parameters coming from the first and the second group, and one output parameter from the third group.

These meta-operators strongly simplify the definition of the JSON file. For example, the JSON file corresponding to 4 rounds of the AES contains 364 parameters and 288 operators. This file is generated by approximately 100 lines of code when using meta-operators.

To test the JSON file, TAGADA provides a function that has three input parameters, i.e., a JSON file F describing a cipher, a plaintext X and a key K , and that returns the ciphertext obtained when ciphering X with K according to F (this computation is done by performing a topological sort to order DAG operators, and applying operators in this order). This function allows us to test the correctness of the JSON file with the *initialization vectors*, i.e., a set of (key, plaintext, ciphertext) triples that are usually provided by cipher authors to validate that implementations are correct. Moreover, these vectors are mandatory for the authors of all candidates to NIST's competitions.

5 Automatic Generation of Models with Tagada

We show how TAGADA automatically generates state-of-the-art MiniZinc models for solving Step1-opt and Step1-enum problems given JSON files that describe ciphers. This is done in four steps: (i) generation of constraints from the black boxes associated with operators (Section 5.1); (ii) simplification of the DAG (Section 5.2); (iii) extension of the DAG (Section 5.3); and (iv) generation of the model from the DAG and the constraints (Section 5.4).

5.1 Automatic Generation of Constraints

As pointed out in Section 3.2, the key point for an efficient process is to tighten the abstraction to prevent as much as possible the generation of non concretizable TDCs. For non-linear operators, we add a constraint to ensure that $\Delta x_1 = \Delta y_1$ where x_1 is the input parameter and y_1 is the output parameter because $\delta x_1 = 0 \Leftrightarrow \delta y_1 = 0$ for all non-linear operators.

For linear operators, we have to add constraints and, in all existing works, these constraints have been manually derived from a careful analysis of operators, as illustrated in Ex. 7 to 9. While this has lead to efficient models, this was also time-consuming and error-prone. Hence, we propose to automatically generate table constraints for which domain consistency can be efficiently achieved. Tables are generated by using the functions that provide operational definitions of these operators. More precisely, the constraint associated with an operator o is the relation \mathcal{R}_o of arity $\#t_{in}(o) + \#t_{out}(o)$ which contains every boolean tuple corresponding to possible difference positions for the input/output parameters of o . As $o(t) \oplus o(t') = o(t \oplus t')$ for any $t, t' \in [0, 1]^{k \cdot \#t_{in}(o)}$, we can build \mathcal{R}_o from the black-box definition of o as follows.

► **Definition 13** (Relation \mathcal{R}_o associated with an operator o).

$\mathcal{R}_o = \{(\Delta(x_1), \dots, \Delta(x_{\#t_{in}(o)}), \Delta(y_1), \dots, \Delta(y_{\#t_{out}(o)})) : \exists(x_1, \dots, x_{\#t_{in}(o)}) \in [0, 1]^{k \cdot \#t_{in}(o)}, (y_1, \dots, y_{\#t_{out}(o)}) = o(x_1, \dots, x_{\#t_{in}(o)})\} \text{ where } \forall x \in [0, 1]^k, \Delta(x) \text{ denotes the Boolean abstraction of } x, \text{ i.e., } \Delta(x) = 0 \Leftrightarrow x = 0.$

To compute this relation, we must (i) enumerate every possible k -bit sequence for every input parameter of o ; (ii) for each enumerated combination of input parameters, call o to compute output parameter values; and (iii) compute the abstract Boolean values $\Delta(x_i)$ and $\Delta(y_j)$ from their corresponding concrete values x_i and y_j . Hence, the time

complexity for building \mathcal{R}_o is $\mathcal{O}(t \cdot 2^{k \cdot \#t_{in}(o)})$ where t is the time complexity of o . Moreover, k is either equal to 4 or 8, and the number of input parameters, $\#t_{in}(o)$, is usually very small: $\#t_{in}(o)$ is always smaller than or equal to four for all ciphers we are aware of. Hence, the relation is rather quickly computed. In the worst case, the relation contains all possible binary tuples of arity $\#t_{in}(o) + \#t_{out}(o)$. Hence, the space complexity of \mathcal{R}_o is $\mathcal{O}((\#t_{in}(o) + \#t_{out}(o)) \cdot 2^{\#t_{in}(o) + \#t_{out}(o)})$.

Note that the relation is computed only once for each black box (identified by its UID), even if the operator is used more than once in the DAG. Also, some operators are shared by multiple ciphers (such as XOR which is used by all ciphers). In this case, we only need to compute the relation once, and we can memorize it for future usage.

► **Example 14** (\mathcal{R}_{xor}). The relation associated with XOR contains all triples $(\Delta(x_1), \Delta(x_2), \Delta(x_1 \oplus x_2))$ such that $x_1, x_2 \in [0, 1]^k$. We obtain the following relation: $\mathcal{R}_{xor} = \{0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 0), (1, 1, 1)\}$. Note that the constraint $(\Delta x_1, \Delta x_2, \Delta y_1) \in \mathcal{R}_{xor}$ has exactly the same semantics as the constraint $\Delta x_1 + \Delta x_2 + \Delta y_1 \neq 1$ which is usually added to model XORs in Step1-opt and Step1-enum models.

► **Example 15** (\mathcal{R}_{MC}). The relation associated with MC contains all tuples $(\Delta(x_1), \Delta(x_2), \Delta(x_3), \Delta(x_4), \Delta(y_1), \Delta(y_2), \Delta(y_3), \Delta(y_4))$ such that $\forall i \in [1, 4], y_i = (M_{i,1} \otimes x_1) \oplus (M_{i,2} \otimes x_2) \oplus (M_{i,3} \otimes x_3) \oplus (M_{i,4} \otimes x_4)$. This relation, for the AES MixColumns, contains 102 tuples and has exactly the same semantics as the constraint associated with the famous MDS property, i.e., it contains only tuples such that the number of 1s belongs to $\{0, 5, 6, 7, 8\}$.

5.2 Simplification of the DAG

Before generating a MiniZinc model from the DAG, we simplify it by applying shaving rules that are described in this section. Each rule removes one or more vertices (and their incident edges), and rules are iteratively applied until reaching a fixed point.

Rule 1: Merging Equal Parameters

When building a relation \mathcal{R}_o from the black box that defines o , we can search for every couple of input/output parameters (x_i, y_j) with $i \in [1, \#t_{in}(o)]$ and $j \in [1, \#t_{out}(o)]$ such that x_i is always equal to y_j : before starting the construction of the relation, we initialize a Boolean variable eq_{x_i, y_j} to true; then, for each generated tuple of input parameters, if $x_i \neq y_j$ we set eq_{x_i, y_j} to false. This does not change the time complexity for building the relation.

We use a list L_{eq} to store all couples of parameter vertices that are related by an equality relation. Before starting the shaving process, L_{eq} is initialized by traversing the DAG: for each operator vertex o and each couple of parameter vertices $(x_i, y_j) \in pred(o) \times succ(o)$, if $eq_{x_i, y_j} = true$, we add (x_i, y_j) to L_{eq} . Rule 1 is triggered whenever L_{eq} is not empty, and it is defined as follows.

► **Definition 16** (Rule 1). *If $L_{eq} \neq \emptyset$, then (i) compute equivalence classes corresponding to all binary equality relations contained in L_{eq} (using a union-find data structure) and reinitialize L_{eq} to the empty set, (ii) merge all vertices of the DAG that belong to a same equivalence class, and (iii) remove every operator vertex that is no longer connected to a parameter vertex.*

► **Example 17** (SR_s). When building the relation \mathcal{R}_{SR_s} , we infer that eq_{x_i, y_j} is true whenever $j = 1 + (i + s) \% 4$. When considering the DAG displayed in Fig. 1, this allows us to merge each of the four predecessors of SR_s vertices with its corresponding successor and, finally, to remove each SR_s vertex.

Rule 2: Suppressing Constant Parameters

When an operator vertex o has an input parameter x_i that has a constant value c , then this parameter is replaced with 0 in the differential characteristic because $c \oplus c = 0$ (see Def. 3) and, therefore, it can be removed from the DAG. Moreover, if all input parameters of o are constants, its outputs are also constants and o can be removed from the DAG.

We use a list L_C to store all parameter vertices that have constant values. Before starting the shaving process, L_C is initialized with the set C of constant parameters. Rule 2 is triggered whenever L_C is not empty, and it is defined as follows.

- **Definition 18** (Rule 2). *If $L_C \neq \emptyset$, then repeat the three following steps until $L_C = \emptyset$:*
- (i) *choose one operator vertex o such that $\text{pred}(o) \cap L_C \neq \emptyset$;*
 - (ii) *remove from the DAG and from L_C every parameter vertex $x_i \in L_C \cap \text{pred}(o)$;*
 - (iii) *if $\text{pred}(o) = \emptyset$, then remove o from the DAG and add every parameter vertex in $\text{succ}(o)$ to L_C , else update the relation \mathcal{R}_o and update L_{eq} if new equality relations can be inferred;*
- **Example 19** (XOR with a constant value). Let us consider a XOR operator with one output parameter y_1 and two input parameters x_1 and x_2 such that x_1 is a constant (i.e., $x_1 \in C$). This operator is used in the key schedule of the AES, for example. In this case, x_1 is removed from the DAG, the relation associated with this operator becomes $\{(0, 0), (1, 1)\}$, and we add the couple (x_2, y_1) to the list L_{eq} .

Rule 3: Suppressing Free Parameters

When an output parameter vertex x has no successor and its predecessor o is a linear operator, then we can remove both o and x from the DAG because we can deterministically compute the output difference δx of o given the differences of all input parameters of o .

Similarly, when an input parameter vertex x has no predecessor, and it has only one successor which is a linear operator, we can also remove both o and x from the DAG because we can deterministically compute the input difference δx of o given the differences of all other input parameters of o and the difference of its output parameter.

More formally, Rule 3 is defined as follows.

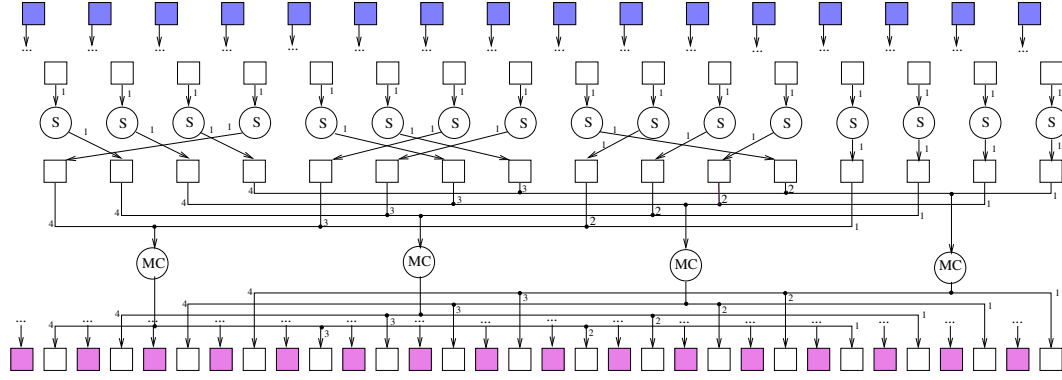
- **Definition 20** (Rule 3). *If there exists a parameter vertex x such that the out-degree of x is equal to 0 and the predecessor of x is a linear operator, then remove x and the predecessor of x from the DAG.*

If there exists a parameter vertex x such that the in-degree of x is equal to 0, the out-degree of x is equal to 1, and the successor of x is a linear operator, then remove x and the successor of x from the DAG.

- **Example 21.** Let us consider the DAG displayed in Fig. 1. Every yellow vertex has no successor and its predecessor is a linear operator (i.e., a XOR). Hence, we can remove all yellow vertices, and all XOR operators that are predecessors of yellow vertices.

Also, every green vertex (corresponding to one byte of the plaintext) has no predecessor and one successor which is a linear operator (i.e., a XOR). Hence, we can remove all green vertices, and all XOR operators that are successors of green vertices.

Note that we cannot remove vertices that precede S operators, though they have no more predecessors once we have removed XOR operators that succeeded green vertices, because S is not linear. The shaved DAG obtained from the DAG of Fig. 1 after applying Rules 1, 2, and 3 is displayed in Fig. 2. We do not apply the shaving rules on vertices associated with the key vertices (in blue and pink) as we have not displayed the operator vertices that are used to compute pink vertices from blue ones in Fig. 1.



■ **Figure 2** Shaved DAG obtained from the DAG of Fig. 1 after applying Rules 1, 2, and 3.

5.3 Extension of the DAG

A basic CP model may be generated from the shaved DAG (this will be explained in Section 5.4). However, the resulting model is often not tight enough, i.e., the bound provided by Step1-opt is smaller than the actual value and/or many solutions of Step1-enum cannot be concretized into differential characteristics with strictly positive probabilities. In this section, we show how to tighten this model by extending the DAG.

5.3.1 Generation of New Vertices and Edges from Existing Operators

In [17, 16, 23], Step1-opt and Step1-enum models are tightened by exploiting the fact that, if $t_1 = MC(t_2)$ and $t_3 = MC(t_4)$ (where t_1, t_2, t_3 , and t_4 are tuples of arity 4), then $t_1 \oplus t_3 = MC(t_2 \oplus t_4)$. As a consequence, the MDS property also holds on $t_1 \oplus t_3$ and $t_2 \oplus t_4$, i.e., the number of k -bit sequences in $t_1 \oplus t_3$ and $t_2 \oplus t_4$ that are different from 0 is either equal to 0 or strictly greater than 4. Hence, a new variable (called *diff* variable in [16]) is added for each parameter of each couple of *MC* operators. These *diff* variables are related with initial parameters by adding XOR constraints. Finally, constraints that ensure the MDS property are added for these new *diff* variables.

In TAGADA, we generalize this idea to all linear operators. Indeed, for any kind of linear operator identified by its UID u , we have $u(t_1) \oplus u(t_2) = u(t_1 \oplus t_2)$. Therefore, for each pair of operator vertices $o_1, o_2 \in O$ such that the UID of o_1 and o_2 is u , we can add a new operator vertex whose UID is u and whose input and output parameter vertices are obtained by XORing input and output parameter vertices of o_1 and o_2 . More precisely, let $pred(o_1) = (x_{1,1}, \dots, x_{1,\#t_{in}(u)})$, $succ(o_1) = (y_{1,1}, \dots, y_{1,\#t_{out}(u)})$, $pred(o_2) = (x_{2,1}, \dots, x_{2,\#t_{in}(u)})$, and $succ(o_2) = (y_{2,1}, \dots, y_{2,\#t_{out}(u)})$. We extend the DAG as follows:

- For each $i \in [1, \#t_{in}(u)]$, we add a new parameter vertex $x_{3,i}$ corresponding to the result of XORing $x_{1,i}$ and $x_{2,i}$, i.e., we add a new XOR vertex whose predecessors are $x_{1,i}$ and $x_{2,i}$ and whose successor is $x_{3,i}$;
- For each $j \in [1, \#t_{out}(u)]$, we add a new parameter vertex $y_{3,j}$ corresponding to the result of XORing $y_{1,j}$ and $y_{2,j}$, i.e., we add a new XOR vertex whose predecessors are $y_{1,j}$ and $y_{2,j}$ and whose successor is $y_{3,j}$;
- We add a new operator vertex o_3 such that the UID of o_3 is u , the predecessors of o_3 are $x_{3,1}, \dots, x_{3,\#t_{in}(u)}$, and the successors of o_3 are $y_{3,1}, \dots, y_{3,\#t_{out}(u)}$.

This may be done for each kind of linear operator except XOR (as this is useless in this case).

As this step may drastically increase the size of the DAG, it is optional, and the user can choose the kind of linear operator that should be considered for this step.

5.3.2 Generation of New XORs

XOR equations may be combined to generate new equations. For example, consider two XOR equations: $a \oplus b \oplus c = 0$, and $b \oplus c \oplus d = 0$. By XORing these two equations, we obtain a new equation $a \oplus d = 0$. This new equation is redundant when computing MDCs, but it tightens the abstraction when computing TDCs. Indeed, let Δi be the boolean abstraction of each k -bit sequence $i \in \{a, b, c, d\}$. If we only post the two constraints $(\Delta a, \Delta b, \Delta c) \in \mathcal{R}_{xor}$ and $(\Delta b, \Delta c, \Delta d) \in \mathcal{R}_{xor}$ (where \mathcal{R}_{xor} is the relation defined in Ex. 14), then it is possible to assign Δa , Δb , and Δc to 1, and Δd to 0 because $(1, 1, 1) \in \mathcal{R}_{xor}$ and $(1, 1, 0) \in \mathcal{R}_{xor}$. However, if we add the constraint $(\Delta a, \Delta d) \in \{(0, 0), (1, 1)\}$, then this assignment is no longer consistent.

This trick was introduced in [16] for the AES, but it has been limited to XORs that occur in the key schedule. In TAGADA, we generalize it to all XORs. Let $adj(o) = pred(o) \cup succ(o)$ be the set of input and output parameters of an operator vertex o . For each couple of operator vertices (o_1, o_2) such that both o_1 and o_2 are XORs that share at least one common parameter (i.e., $adj(o_1) \cap adj(o_2) \neq \emptyset$), we compute the set $S = (adj(o_1) \cup adj(o_2)) \setminus (adj(o_1) \cap adj(o_2))$ (corresponding to parameters that are adjacent to o_1 or o_2 but not to both o_1 and o_2). If S does not contain more than n_{max} parameters, then we add a new operator vertex o_3 to the DAG, and we add an edge between each parameter vertex in S and o . This process is recursively applied, until no more vertex can be added.

n_{max} is a given integer value that is used to control the growth of the DAG: when $n_{max} = 0$, no new XOR operator is added to the DAG; the larger n_{max} , the more XOR operators are added.

For all possible values of $\#S \in [0, n_{max}]$, we have to generate the relation associated with a XOR of $\#S$ parameters, as described in Section 5.1. Also, we infer equality relations and apply Rule 1 (as described in Section 5.2) to merge vertices of the DAG that belong to a same equivalence class.

5.4 Generation of the MiniZinc Model from the DAG

Given a DAG, we generate a MiniZinc model as follows:

- We declare a Boolean variable Δx for each parameter vertex x of the DAG;
- We add a constraint $\Delta(pred(o), succ(o)) \in \mathcal{R}_o$ for each operator vertex o (where $\Delta(pred(o), succ(o))$ is the tuple of Boolean variables associated with parameters in $pred(o)$ and $succ(o)$);
- We declare an integer variable s which corresponds to the number of active non-linear operators in the TDC, and we add a constraint $s = \sum_{x \in NL} \Delta x$ where NL contains the set of parameter vertices that are predecessors of a non-linear operator vertex.

For Step1-opt, the goal is to minimize s , and we add the constraint $s \geq 1$ because TDCs must contain at least one active non-linear operator. For Step1-enum, s is assigned to the number of active non-linear operators, and the goal is to enumerate all solutions.

■ **Table 1** Model performance summary of Picat-SAT on the 35 Midori instances, 25 AES instances, 56 SKINNY instances and 38 CRAFT instances, for different values of n_{max} ranging from 0 to 5. The 6 first (resp. last) rows give results without (resp. with) selecting MC. $\#d$ corresponds to the number of instances where the model is not tight enough. When $\#d=0$, we report the number of instances that are solved within 1 hour for Step1-opt ($\#o$) and Step1-enum ($\#e$), and we highlight the best values. We report – when models have not been generated because DAGs are too large.

model	Midori (35)			AES (25)			SKINNY (56)			CRAFT (38)		
	$\#d$	$\#o$	$\#e$	$\#d$	$\#o$	$\#e$	$\#d$	$\#o$	$\#e$	$\#d$	$\#o$	$\#e$
$n_{max}=0$	18			12			0	24	22	0	38	38
$n_{max}=1$	18			12			0	25	22	0	38	38
$n_{max}=2$	18			12			0	25	22	0	38	38
$n_{max}=3$	18			12			0	25	22	0	38	38
$n_{max}=4$	18			12			0	24	22	0	38	38
$n_{max}=5$	–	–	–	12			–	–	–	–	–	–
$n_{max}=0$ MC	18			12			–	–	–	0	38	38
$n_{max}=1$ MC	18			12			–	–	–	0	38	38
$n_{max}=2$ MC	18			12			–	–	–	0	38	38
$n_{max}=3$ MC	18			12			–	–	–	0	38	38
$n_{max}=4$ MC	0	35	34	0	23	21	–	–	–	0	37	37
$n_{max}=5$ MC	–	–	–	0	24	21	–	–	–	–	–	–

6 Experimental Results

We performed all experiments on a PC with a Xeon Gold 5118 (2.30 GHz) with 24 cores and 32 GB of RAM. Each experiment used only one thread, and we ran 20 of them in parallel to speed up the computations. All the source-code and results are available online ^{2 3}.

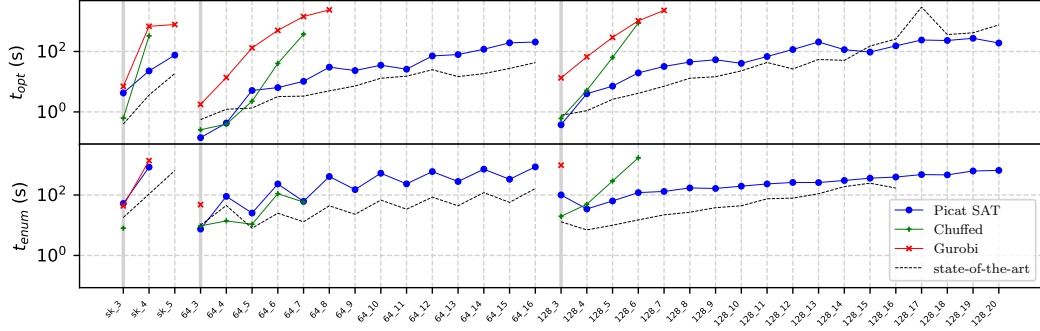
We consider four symmetric block ciphers for which there exist recent differential cryptanalysis results, i.e., the AES [16], Midori [14], Skinny [11], and Craft [18]. For each cipher, there are different instances that are obtained by considering either single-key or related-key attacks, by changing the size of the key for related-key attacks of ciphers that have different key lengths (i.e., 64 and 128 for Midori, 128, 192, and 256 for the AES), and by changing the number r of rounds of the ciphering process, starting from $r = 3$ up to the largest value for r considered in the literature. We obtain 35 (resp. 25, 56, and 38) instances for Midori (resp. the AES, Skinny, and Craft). Finally, for each instance, we solve two different problems, i.e., Step1-opt and Step1-enum. Hence, our benchmark contains 308 instances.

TAGADA has a parameter n_{max} that is used to control the maximum size of new generated XOR equations (see Section 5.3.2). It is also possible to select the linear operators for which we infer new vertices and edges as explained in Section 5.3.1. In the four considered ciphers, the only linear operator that can be selected is MC as SR is removed during the DAG shaving step. Increasing n_{max} and/or selecting MC tightens the abstraction, but it also increases the number of variables and constraints in the generated model.

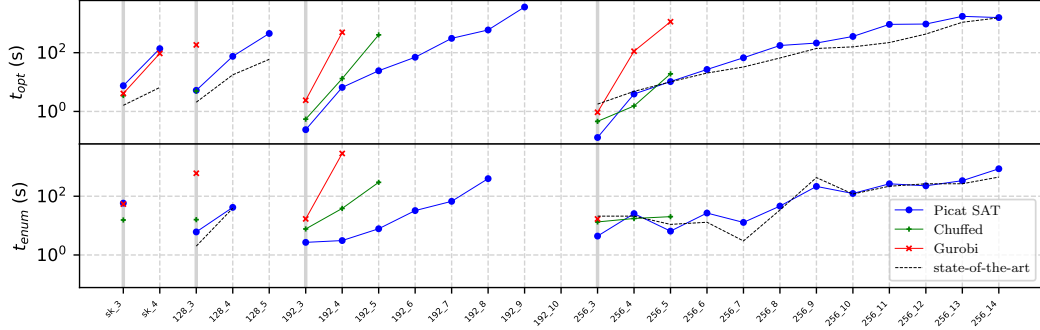
In Table 1, we report the number of instances for which the generated model is not tight enough (i.e., the bound computed by Step1-opt is smaller than the best known bound) for different values of n_{max} and with or without selecting MC . This shows us that the best

² Tagada: https://gitlab.limos.fr/iia_lulibral/tagada/

³ models and results: https://gitlab.limos.fr/iia_lulibral/experiment-results



■ **Figure 3** CPU time of Picat-SAT, Chuffed and Gurobi on the model generated by TAGADA for Midori instances when $n_{max} = 4$ and MC is selected (top plot for Step1-opt and bottom plot for Step1-enum). State-of-the art is the handcrafted model of [14] run with Picat-SAT.



■ **Figure 4** CPU time of Picat-SAT, Chuffed and Gurobi on the model generated by TAGADA for AES instances when $n_{max} = 5$ and MC is selected (top plot for Step1-opt and bottom plot for Step1-enum). State-of-the art is the handcrafted model of [16] run with Picat-SAT.

parameter setting depends on the cipher: For Midori and the AES, it is necessary to select MC and to set n_{max} to a value larger than or equal to 4 to generate a model that is tight enough for all instances; For Skinny and Craft, the generated model is tight enough even when $n_{max} = 0$ and MC is not selected.

In Table 1, we also report the number of instances that are solved within one hour of CPU time by Picat-SAT [27] whenever the model is tight enough (it is meaningless to report these results when models are not tight enough, as they do not solve the same problem). When increasing n_{max} , the model has more constraints, and the number of new constraints grows exponentially with n_{max} . In [16] and [14], this parameter has been fixed to 4 for the handcrafted models, and this seems to be a rather good setting. However, for the AES, one more instance is solved when increasing n_{max} to 5, and for Skinny one more instance is solved when decreasing n_{max} to 3. For Midori, Skinny and Craft, when $n_{max} = 5$ the number of new constraints is so large that we have not run the resulting models. As models are automatically generated by TAGADA, the user can easily fiddle with parameters to find the settings that generate the tightest and most efficient models for a cipher.

In Fig. 3 to 6, we display results, on a per-instance basis, and for three different kinds of solvers, i.e. Picat-SAT [27] (that generates a SAT instance from the MiniZinc model and uses Lingeling to solve it), Gurobi [22] (which is an ILP solver), and Chuffed [9] (which is

a CP solver with lazy clause generation). For these figures, we report results for the best parameter setting for each cipher, i.e., $n_{max} = 4$ and MC is selected for Midori, $n_{max} = 5$ and MC is selected for the AES, $n_{max} = 0$ and MC is not selected for Skinny and Craft.

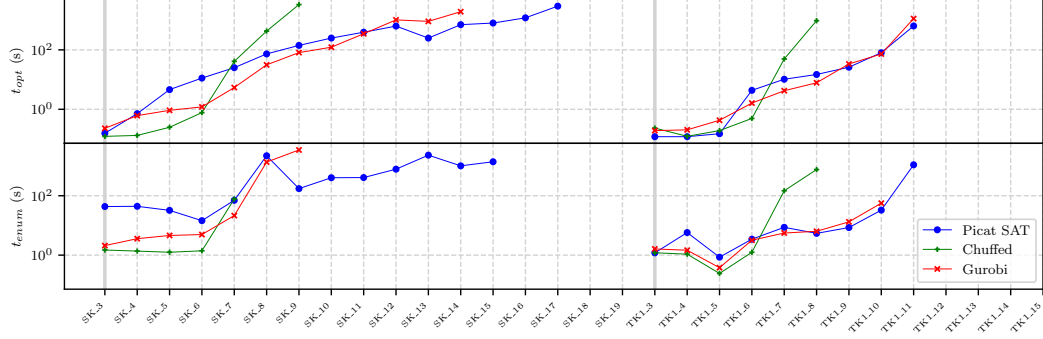


Figure 5 CPU time of Picat-SAT, Chuffed and Gurobi on the model generated by TAGADA for Skinny when $n_{max} = 0$ and MC is not selected (top plot for Step1-opt and bottom plot for Step1-enum).

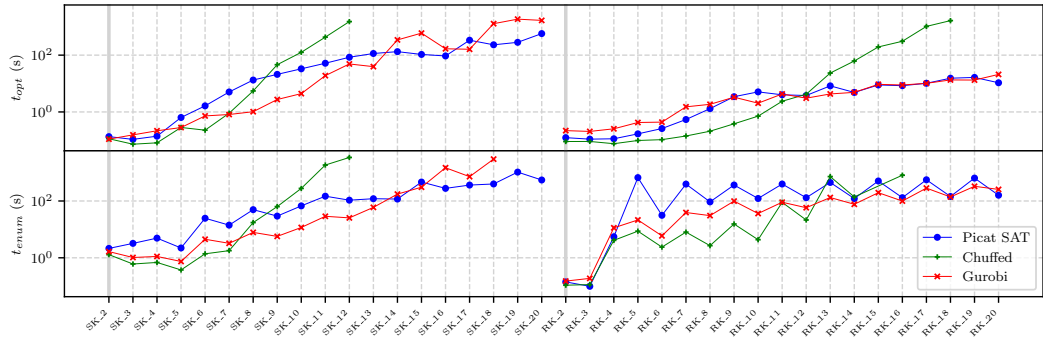


Figure 6 CPU time of Picat-SAT, Chuffed and Gurobi on the model generated by TAGADA for Craft when $n_{max} = 0$ and MC is not selected (top plot for Step1-opt and bottom plot for Step1-enum).

Picat-SAT is usually more efficient than Chuffed and Gurobi. However, Chuffed is often faster on small instances, and Gurobi is the best performing solver on many Craft instances.

The MiniZinc models for the AES and Midori described in [16] and [14] are publicly available, and we compare our automatically generated models with these handcrafted models (we only report results with Picat-SAT in this case as this is the best performing solver). However, for instances of AES-192 we do not report results obtained with the model of [16] because it does not solve the same problem: for these instances, the model of [16] does not integrate in the objective function the S-boxes of the last round, which is an error of this model for this particular case. For both Midori and the AES, models automatically generated with TAGADA are competitive with state-of-the-art handcrafted models. The largest Midori instances (when the key has 128 bits and the number of rounds is greater than 17) cannot be solved within one hour by the model of [14] whereas the TAGADA model solves them. This is remarkable because it takes weeks/months for a researcher to design these handcrafted models. Moreover, with TAGADA we can check that the description of the cipher is correct (as explained in Section 4), and the model is automatically generated from this description without any human action (except parameter selection).

For Skinny, the most efficient approach is a dedicated dynamic program [11]. However, this approach consumes huge amounts of memory (more than 700 GB of RAM). In [11], a MiniZinc model is also described, and results obtained with Picat-SAT are reported. The number of instances solved by this approach within one hour on a server composed of $2 \times$ AMD EPYC7742 64-Core is the same as with our TAGADA model when using Picat-SAT, i.e., 22.

Finally, for Craft, [5] only reports optimal solutions of Step1-opt and does not report CPU times. Our TAGADA model has found the same solutions as those of [5].

7 Conclusion

In this article, we present TAGADA, a tool for automatically generating MiniZinc models for solving differential cryptanalysis problems given the description of a symmetric block cipher. The description is based on a unifying framework, i.e., a DAG that describes how operators are combined and black-boxes that give an operational definition of operators.

This description allows us to perform a correctness verification using initialization vectors and comparing the behavior of our implementation with reference implementations found in the literature, limiting the possible bugs.

Then, for each black box operator, we perform an exhaustive search of its input and output values to infer a relation that represents a provably optimal abstraction for this operator. The DAG is further modified by removing some parts that are not useful for differential attacks, and by adding new operators that tighten the model. Finally, the MiniZinc model is generated from the relations and the DAG.

We experimentally compare automatically generated models with state-of-the-art approaches on four ciphers (Midori, AES, Skinny, Craft) and on two types of attacks (Single-Key and Related-Key). For all scenarios, our models find the same solutions as hand-crafted models, and they have similar running times.

While the models generated by TAGADA have the same tightness and performance as state-of-the-art hand-crafted models, MIP/CP/SAT solvers still struggle to solve the largest instances. Recently, some ad-hoc dynamic programming algorithms have been proposed (for instance, on Skinny [11]), and show better scale-up properties though they have high space complexities. Hence, we plan to study the possibility of integrating dynamic programming approaches within TAGADA.

Also, we plan to integrate other differential attacks than single-key and related-key (i.e., related-tweak, related-tweakey and boomerang attacks), and to extend TAGADA so that it also generates models for computing MDCs given TDCs. Of course, we will use TAGADA to analyze the recent ten finalists of NIST's competition, as there is a need to provide quickly differential attacks (or prove the robustness of the cipher against these attacks).

References

- 1 S. Banik, A. Bogdanov, T. Isobe, K. Shibutani, H. Hiwatari, T. Akishita, and F. Regazzoni. Midori: A block cipher for low energy. In *ASIACRYPT*, volume 9453 of *LNCS*, pages 411–436. Springer, 2015.
- 2 Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. Gift: a small present. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 321–345. Springer, 2017.
- 3 Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith, and Louis Wingers. The simon and speck lightweight block ciphers. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

- 4 Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In *CRYPTO 2016 – 36th Annual International Cryptology Conference*, volume 9815 of *Lecture Notes in Computer Science*, pages 123–153. Springer, 2016.
- 5 Christof Beierle, Gregor Leander, Amir Moradi, and Shahram Rasoolzadeh. Craft: Lightweight tweakable block cipher with efficient protection against dfa attacks. *IACR Transactions on Symmetric Cryptology*, 2019(1):5–45, 2019.
- 6 E. Biham and A. Shamir. Differential cryptanalysis of feal and n-hash. In *EUROCRYPT*, volume 547 of *LNCS*, pages 1–16. Springer, 1991.
- 7 A. Biryukov and I. Nikolic. Automatic search for related-key differential characteristics in byte-oriented block ciphers: Application to AES, camellia, khazad and others. In *Advances in Cryptology, LNCS 6110*, pages 322–344. Springer, 2010.
- 8 Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelse. Present: An ultra-lightweight block cipher. In *International workshop on cryptographic hardware and embedded systems*, pages 450–466. Springer, 2007.
- 9 Geoffrey Chu and Peter J. Stuckey. Chuffed solver description, 2014. Available at http://www.minizinc.org/challenge2014/description_chuffed.txt.
- 10 Carlos Cid, Tao Huang, Thomas Peyrin, Yu Sasaki, and Ling Song. A security analysis of deoxys and its internal tweakable block ciphers. *IACR Trans. Symmetric Cryptol.*, 2017(3):73–107, 2017.
- 11 Stéphanie Delaune, Patrick Derbez, Paul Huynh, Marine Minier, Victor Mollimard, and Charles Prud’homme. SKINNY with scalpel – comparing tools for differential analysis. *IACR Cryptol. ePrint Arch.*, 2020:1402, 2020.
- 12 FIPS 197. Advanced Encryption Standard. Federal Information Processing Standards Publication 197, 2001. U.S. Department of Commerce/N.I.S.T.
- 13 P. Fouque, J. Jean, and T. Peyrin. Structural evaluation of AES and chosen-key distinguisher of 9-round AES-128. In *Advances in Cryptology – CRYPTO 2013 – Part I*, volume 8042 of *LNCS*, pages 183–203. Springer, 2013.
- 14 D. Gérard. *Security Analysis of Contactless Communication Protocols*. PhD thesis, Université Clermont Auvergne, 2018.
- 15 D. Gérard and P. Lafourcade. Related-key cryptanalysis of midori. In *INDOCRYPT*, volume 10095 of *LNCS*, pages 287–304, 2016.
- 16 D. Gérard, P. Lafourcade, M. Minier, and C. Solnon. Computing AES related-key differential characteristics with constraint programming. *Artif. Intell.*, 278, 2020.
- 17 D. Gérard, M. Minier, and C. Solnon. Constraint programming models for chosen key differential cryptanalysis. In *CP*, volume 9892 of *LNCS*, pages 584–601. Springer, 2016.
- 18 Hosein Hadipour, Sadegh Sadeghi, Majid M Niknam, Ling Song, and Nasour Bagheri. Comprehensive security analysis of craft. *IACR Transactions on Symmetric Cryptology*, pages 290–317, 2019.
- 19 Jérémy Jean, Ivica Nikolic, Thomas Peyrin, and Yannick Seurin. Deoxys v1. 41. *Submitted to CAESAR*, 2016.
- 20 L. Knudsen. Truncated and higher order differentials. In *Fast Software Encryption*, pages 196–211. Springer, 1995.
- 21 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*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.
- 22 Gurobi Optimization. Gurobi optimizer reference manual, 2018. URL: <http://www.gurobi.com>.
- 23 L. Rouquette and C. Solnon. abstractXOR: A global constraint dedicated to differential cryptanalysis. In *26th International Conference on Principles and Practice of Constraint Programming*, volume 12333 of *LNCS*, pages 566–584, Louvain-la-Neuve, Belgium, 2020. Springer.

- 24 CE Shannon. Communication theory of secrecy systems, bell systems tech. *Bell System Technical Journal*, 28:656–715, 1949.
- 25 R. Singleton. Maximum distance q-nary codes. *IEEE Trans. Inf. Theor.*, 10(2):116–118, 2006. doi:10.1109/TIT.1964.1053661.
- 26 Boxin Zhao, Xiaoyang Dong, and Keting Jia. New related-tweakey boomerang and rectangle attacks on deoxys-bc including bdt effect. *IACR Transactions on Symmetric Cryptology*, pages 121–151, 2019.
- 27 N.-F. Zhou, H. Kjellerstrand, and J. Fruhman. *Constraint Solving and Planning with Picat*. Springer, 2015.

A Bound-Independent Pruning Technique to Speeding up Tree-Based Complete Search Algorithms for Distributed Constraint Optimization Problems

Xiangshuang Liu ✉

College of Computer Science, Chongqing University, China

Ziyu Chen¹ ✉

College of Computer Science, Chongqing University, China

Dingding Chen ✉

College of Computer Science, Chongqing University, China

Junsong Gao ✉

College of Computer Science, Chongqing University, China

Abstract

Complete search algorithms are important methods for solving Distributed Constraint Optimization Problems (DCOPs), which generally utilize bounds to prune the search space. However, obtaining high-quality lower bounds is quite expensive since it requires each agent to collect more information aside from its local knowledge, which would cause tremendous traffic overheads. Instead of bothering for bounds, we propose a Bound-Independent Pruning (BIP) technique for existing tree-based complete search algorithms, which can independently reduce the search space only by exploiting local knowledge. Specifically, BIP enables each agent to determine a subspace containing the optimal solution only from its local constraints along with running contexts, which can be further exploited by any search strategies. Furthermore, we present an acceptability testing mechanism to tailor existing tree-based complete search algorithms to search the remaining space returned by BIP when they hold inconsistent contexts. Finally, we prove the correctness of our technique and the experimental results show that BIP can significantly speed up state-of-the-art tree-based complete search algorithms on various standard benchmarks.

2012 ACM Subject Classification Computing methodologies → Cooperation and coordination

Keywords and phrases DCOP, complete algorithms, search

Digital Object Identifier 10.4230/LIPIcs.CP.2021.41

Supplementary Material *Software (Source Code)*: <https://github.com/czy920/BIP>

Acknowledgements We would like to thank the reviewers for their valuable comments, and Dr. Ferdinando Fioretto and Dr. Yanchen Deng for their helpful suggestions.

1 Introduction

Distributed Constraint Optimization Problems (DCOPs) [14, 10] are a fundamental framework for coordinated and cooperative multi-agent systems. They have been widely deployed in many real applications such as sensor network [8], task scheduling [18, 26], smart grid [11] and many others.

¹ Corresponding author.



Incomplete algorithms for DCOPs [18, 28, 17, 9, 22, 21] aim to rapidly find a good solution at an acceptable overhead, while complete algorithms guarantee to find the optimal one by employing either inference or search to systematically explore the entire solution space. DPOP [23] and Action_GDL [27] are typical inference-based complete algorithms which perform dynamic-programming to solve a DCOP. However, they require a linear number of messages of exponential size. MB-DPOP and its variant [25, 5] proposed to trade the number of messages for smaller size of message. DPOP with function filtering [3] exploits utility bounds to reduce the size of message, where agents need to collect projected utilities to establish utility bounds.

On the other hand, search-based complete algorithms perform distributed backtrack searches and have a linear size of messages but an exponential number of messages. Tree-based complete search algorithms are the most popular ones among them and normally utilize bounds to prune the search space. Some work has been done to tailor centralized pruning techniques such as soft arc consistency [6] to tighten lower bounds in a distributed setting. BnB-ADOPT⁺-AC/FDAC [12] proposed to get strong lower bounds via arc consistency (AC) and full directional arc consistency (FDAC) levels of soft arc consistency. However, stronger consistency levels require agents to know more information about other agents to plan sequences of soft arc consistency operations, which would compromise privacy and cause tremendous communication overheads. Besides, PT-FB [16] builds tight lower bounds via a forward bounding procedure which requires cost estimates from neighbors. ADOPT-BDP [1], DJAO [15] and PT-ISABB [7] came out to perform an approximation inference to acquire tighter lower bounds in the preprocessing phase. Recently, HS-CAI [4] was proposed to tighten lower bounds by executing the context-based inference iteratively. Like soft arc consistency, these methods also need to collect more information aside from local knowledge and thus lead to traffic overheads inevitably.

In a nutshell, the existing pruning techniques for DCOPs are bound-dependent and require collecting more information to obtain tight bounds. Different from them, we present a novel pruning technique independent of bounds and dispensing with information collection to accelerate existing tree-based complete search algorithms. More specifically, our contributions are listed as follows.

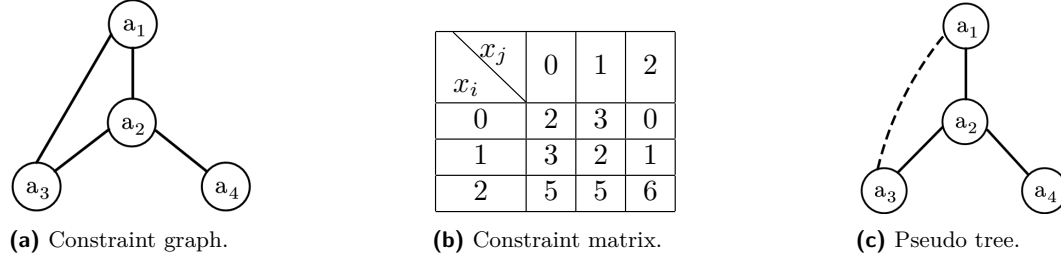
- We present a Bound-Independent Pruning (BIP) technique for existing tree-based complete search algorithms, which utilizes local constraints and running contexts to cut down the search space independently.
- We further introduce an acceptability testing mechanism (ATM) to filter out unacceptable search results produced by existing tree-based complete search algorithms when enforcing BIP under the inconsistent contexts.
- We theoretically show the correctness of BIP and ATM, and the experimental results demonstrate that BIP substantially improves state-of-the-art tree-based complete search algorithms on all the metrics in most cases.

2 Background

In this section, we introduce the preliminaries including DCOPs, pseudo tree and tree-based complete search algorithms.

2.1 Distributed Constraint Optimization Problems

A distributed constraint optimization problem [19] can be formalized by a tuple $\langle A, X, D, F \rangle$ where



■ **Figure 1** An example of a DCOP and its pseudo tree.

- $A = \{a_1, a_2, \dots, a_n\}$ is a set of agents.
- $X = \{x_1, x_2, \dots, x_m\}$ is a set of variables.
- $D = \{D_1, D_2, \dots, D_m\}$ is a set of finite and discrete domains, where each variable x_i takes a value assignment in D_i . Here, we denote the maximal domain size as $d_{max} = \max_{a_i \in A} |D_i|$.
- $F = \{f_1, f_2, \dots, f_q\}$ is a set of constraint functions, where each constraint $f_i : D_{i_1} \times \dots \times D_{i_k} \rightarrow \mathbb{R}_{\geq 0}$ specifies the non-negative cost for each combination of x_{i_1}, \dots, x_{i_k} .

For the sake of simplicity, we assume that each agent controls exactly one variable (i.e., $n = m$) and all constraint functions are binary (i.e., $f_{ij} : D_i \times D_j \rightarrow \mathbb{R}_{\geq 0}$). Thus, the term *agent* and *variable* can be used interchangeably. An optimal solution to a DCOP is an assignment to all the variables such that the total cost is minimized. That is,

$$X^* = \arg \min_{d_i \in D_i, d_j \in D_j} \sum_{f_{ij} \in F} f_{ij}(x_i = d_i, x_j = d_j)$$

A DCOP can be visualized by a constraint graph where the nodes denote agents and the edges denote constraints. Figure 1 (a) gives a DCOP with four variables and four constraints. For simplicity, the domain size of each variable is three and all the constraints are identical as shown in Fig. 1 (b) where $i < j$.

2.2 Pseudo Tree

A pseudo tree is a partial ordered arrangement to a constraint graph and can be generated by depth-first search traversal, where different branches are independent from each other and its constraints are categorized into tree edges and pseudo edges (i.e., non-tree edges). According to the relative positions in a pseudo tree, the neighbors of an agent a_i connected by tree edges are categorized into its parent $P(a_i)$ and children $C(a_i)$, while the ones connected by pseudo edges are denoted as its pseudo parents $PP(a_i)$ and pseudo children $PC(a_i)$. For succinctness, we also denote all its (pseudo) parents as $AP(a_i) = PP(a_i) \cup \{P(a_i)\}$, all its (pseudo) children as $CD(a_i) = C(a_i) \cup PC(a_i)$, its ancestors as $Anc(a_i)$ and its descendants as $Desc(a_i)$. Besides, we denote the set of ancestors who share constraints with a_i and its descendants as $Sep(a_i)$ [24]. Figure 1 (c) gives a possible pseudo tree deriving from Fig. 1 (a) where tree edges and pseudo edges are denoted by solid and dashed lines, respectively.

2.3 Tree-based Complete Search Algorithms

Tree-based complete search algorithms perform a systematic search on a pseudo tree. Specifically, each agent a_i traverses the subtree rooted at itself under the running context $Context_i$ (i.e., the assignment to $Sep(a_i)$) and avoids expanding suboptimal branches by exploiting the bounds including the lower and upper bounds of its subproblem (i.e., LB_i and UB_i), the ones of its subproblem given its value $d_i \in D_i$ (i.e., $LB_i(d_i)$ and $UB_i(d_i)$) and the ones of its subproblem given its value $d_i \in D_i$ for its child $a_c \in C(a_i)$ (i.e., $lb_i^c(d_i)$ and $ub_i^c(d_i)$).

According to the way that agents update their assignments, tree-based complete search algorithms can be classified as synchronous or asynchronous. Synchronous algorithms constrain the agents' decisions to follow a particular order. As a result, only when a_i and its descendants have thoroughly explored its subproblem given $Context_i$, it reports its search results including LB_i and UB_i (here, $LB_i = UB_i = opt_i$ if $Context_i$ is feasible and $LB_i = UB_i = \infty$ otherwise, and opt_i is the optimal solution cost of the subproblem) if it is a non-root agent; otherwise, it finds the optimal solution. In contrast, asynchronous ones allow agents to update their assignments solely based on their local view of their subproblems and report their search results including LB_i and UB_i (here, $LB_i \leq opt_i \leq UB_i$) continually. Once the root agent a_i has $LB_i = UB_i$, the optimal solution is found.

Next, we will take PT-FB and BnB-ADOPT for example to describe the concrete implementation of synchronous and asynchronous tree-based complete search algorithms, respectively. In PT-FB, each agent a_i explores its subproblem by sequentially expanding $Context_i$ via sending CPA messages² to its children, and reports its search results via a UB message³ once exhausting its domain. When receiving a UB message including LB_c and UB_c from a_c , a_i updates $lb_i^c(d_i)$ and $ub_i^c(d_i)$ with LB_c and UB_c , respectively. As for BnB-ADOPT, each agent a_i explores its subproblem by constantly informing its current value to its constrained descendants via VALUE messages⁴ and reporting its search results including LB_i and UB_i to its parent via a COST message². Similarly to PT-FB, once receiving a COST message including LB_c and UB_c from a_c , a_i updates $lb_i^c(d_i)$ and $ub_i^c(d_i)$ with LB_c and UB_c , respectively. Afterwards, it updates $LB_i(d_i)$ and $UB_i(d_i)$, and then updates LB_i and UB_i according to the following equations.

$$LB_i = \min_{d_i \in D_i} \{LB_i(d_i)\} \quad (1)$$

$$UB_i = \min_{d_i \in D_i} \{UB_i(d_i)\} \quad (2)$$

3 Proposed Method

In this section, we propose a Bound-Independent Pruning (BIP) technique for existing tree-based complete search algorithms, where each agent solely exploits its local constraints and running contexts to confirm a subspace containing the optimal solution and thereby obtains pruned domains for itself and its children under the current context. We further introduce an acceptability testing mechanism to tailor existing tree-based complete search algorithms to match BIP when they hold inconsistent contexts. Before elaborating on BIP, we first introduce some terms, definitions and their related properties used in this paper.

► **Definition 1 (dims).** Let U be a cost table, $dims(U)$ is the set of variables involved in U .

$D^U = \times_{x_i \in dims(U)} D_i$ is the set of all value combinations of $dims(U)$. Given $x_i \in dims(U)$, $D_{-i}^U = \times_{x_j \in dims(U) \setminus \{x_i\}} D_j$ is the set of all value combinations of $dims(U)$ except x_i . Particularly, we specify $D_{-i}^U = \{\emptyset\}$ when $dims(U) \setminus \{x_i\} = \emptyset$.

Take f_{12} in Fig. 1(b) for example. We have $dims(f_{12}) = \{x_1, x_2\}$ and $D_{-2}^{f_{12}} = \{((x_1, 0)), ((x_1, 1)), ((x_1, 2))\}$.

² A CPA message contains the Current Partial Assignment of the sending agent and all its ancestors.

³ A UB (COST) message contains the results of a solution found to the sending agent's sub-problem.

⁴ A VALUE message contains the value assignment of the sending agent.

► **Definition 2** (POS). Given $\mathbf{d}_{-i} \in D_{-i}^U$, $d_i \in D_i$ is a *Primary Optimal Support (POS)* for \mathbf{d}_{-i} on U , denoted by $\text{pos}_U(\mathbf{d}_{-i})$, iff $U(d_i, \mathbf{d}_{-i}) = \min_{d'_i \in D_i} U(d'_i, \mathbf{d}_{-i})$ (ties are broken alphabetically).

Consider f_{12} . When $\mathbf{d}_{-2} = ((x_1, 2))$, we have $\text{pos}_{f_{12}}(\mathbf{d}_{-2})=0$ since $f_{12}(\mathbf{d}_{-2}, x_2 = 0) = f_{12}(\mathbf{d}_{-2}, x_2 = 1) < f_{12}(\mathbf{d}_{-2}, x_2 = 2)$ and 0 precedes 1.

According to Definition 2, we can obtain a subset of D^U , denoted by S_i^U , by Eq. (3).

$$S_i^U = \{(d_i, \mathbf{d}_{-i}) \mid d_i = \text{pos}_U(\mathbf{d}_{-i}), \forall \mathbf{d}_{-i} \in D_{-i}^U\} \quad (3)$$

Considering f_{12} again, we have $S_2^{f_{12}} = \{((x_2, 2), (x_1, 0)), ((x_2, 2), (x_1, 1)), ((x_2, 0), (x_1, 2))\}$.

► **Definition 3** (Join). Let U, U' be two cost tables, the *join* of U and U' , denoted by $U \otimes U'$, is a relation defined over $D^{U \otimes U'} = \times_{x_i \in \text{dims}(U) \cup \text{dims}(U')} D_i$ such that

$$(U \otimes U')(\mathbf{d}) = U(\mathbf{d}_{[\text{dims}(U)]}) + U'(\mathbf{d}_{[\text{dims}(U')]}), \forall \mathbf{d} \in D^{U \otimes U'}$$

where $\mathbf{d}_{[\text{dims}(U)]}$ and $\mathbf{d}_{[\text{dims}(U')]}$ are slices of \mathbf{d} along $\text{dims}(U)$ and $\text{dims}(U')$, respectively.

Next, we give the following properties of S_i^U , based on which BIP is presented.

► **Property 4.** There exists at least one element in S_i^U leading to the optimal cost in U . That is, $\exists \mathbf{d} \in S_i^U$, s.t. $U(\mathbf{d}) = \min_{\mathbf{d}' \in D^U} U(\mathbf{d}')$.

Proof. Assume that $\forall \mathbf{d} \in S_i^U$, $U(\mathbf{d}) > U(\mathbf{d}^*)$ where $\mathbf{d}^* = (d_i^*, \mathbf{d}_{-i}^*)$ is the optimal solution.

According to Definition 2, we have $\exists d'_i \in D_i$, s.t. $d'_i = \text{pos}_U(\mathbf{d}_{-i}^*)$ and thus $U(d'_i, \mathbf{d}_{-i}^*) = U(\mathbf{d}^*)$. Furthermore, we have $(d'_i, \mathbf{d}_{-i}^*) \in S_i^U$ by Eq. (3). That is, $(d'_i, \mathbf{d}_{-i}^*) \in S_i^U$ and $U(d'_i, \mathbf{d}_{-i}^*) = U(\mathbf{d}^*)$ which contradicts the assumption. Thus, Property 4 is proved. ◀

► **Property 5.** Given two cost tables U and U' , $S_i^U = S_i^{U'}$ if

$$U = U' \otimes U'' \quad (4)$$

where $\text{dims}(U'') \subseteq \text{dims}(U') \setminus \{x_i\}$ and $x_i \in \text{dims}(U')$.

Proof. Firstly, we prove that $U(d_i, \mathbf{d}_{-i}) - U(d'_i, \mathbf{d}_{-i}) = U'(d_i, \mathbf{d}_{-i}) - U'(d'_i, \mathbf{d}_{-i})$, $\forall d_i, d'_i \in D_i, \mathbf{d}_{-i} \in D_{-i}^U$.

According to Eq. (4), we have $\text{dims}(U') = \text{dims}(U)$ and $x_i \in \text{dims}(U)$. Thus, for all $d_i, d'_i \in D_i$ and $\mathbf{d}_{-i} \in D_{-i}^U$, we have

$$\begin{aligned} & U(d_i, \mathbf{d}_{-i}) - U(d'_i, \mathbf{d}_{-i}) \\ &= (U'(d_i, \mathbf{d}_{-i}) + U''(\mathbf{d}_{-i[\text{dims}(U'')]})) - (U'(d'_i, \mathbf{d}_{-i}) + U''(\mathbf{d}_{-i[\text{dims}(U'')]})) \\ &= U'(d_i, \mathbf{d}_{-i}) - U'(d'_i, \mathbf{d}_{-i}) \end{aligned} \quad (5)$$

Next, we prove that given $\mathbf{d}_{-i} \in D_{-i}^U$ and $d_i \in D_i$, $U(d_i, \mathbf{d}_{-i}) = \min_{d'_i \in D_i} U(d'_i, \mathbf{d}_{-i})$ if $U'(d_i, \mathbf{d}_{-i}) = \min_{d'_i \in D_i} U'(d'_i, \mathbf{d}_{-i})$.

Assume that $\exists d'_i \in D_i$, s.t. $U(d_i, \mathbf{d}_{-i}) > U(d'_i, \mathbf{d}_{-i})$. Since $U'(d_i, \mathbf{d}_{-i}) = \min_{d'_i \in D_i} U'(d'_i, \mathbf{d}_{-i})$, we have $U'(d_i, \mathbf{d}_{-i}) \leq U'(d'_i, \mathbf{d}_{-i})$, $\forall d'_i \in D_i$. Further, we have $U(d_i, \mathbf{d}_{-i}) \leq U(d'_i, \mathbf{d}_{-i})$, $\forall d'_i \in D_i$ by Eq. (5), which is contradictory to the assumption. Thus, the conclusion holds.

Based on the above conclusion and Definition 2, we have

$$\text{pos}_U(\mathbf{d}_{-i}) = \text{pos}_{U'}(\mathbf{d}_{-i}) \quad (6)$$

Therefore, we can conclude $S_i^U = S_i^{U'}$ based on Eqs. (3) and (6), and thereby Property 5 is proved. ◀

According to Property 5, we can readily obtain S_i^U from U' if U and U' satisfy Eq. (4). Namely, we can derive the desired subspace of a table from its subtable which includes its partial information if they satisfy Eq. (4). Look into a DCOP. Given its pseudo tree and a running context, if U' and U are respectively instantiated to the combination cost table of local constraints at an agent a_i and the combination cost table of all constraints in the subproblem rooted at itself after eliminating $Desc(a_i) \setminus CD(a_i)$, we find that they still satisfy Eq. (4) (see Lemma 7 for detail). Accordingly, each agent a_i can confirm the subspace containing the optimal solution like S_i^U from its local constraints under the running context. Hereby, we propose BIP for tree-based complete search algorithms.

3.1 Bound-Independent Pruning(BIP) Technique

BIP aims to enable each agent a_i to exclude some elements in $D^{U_i} \setminus S_i^{U_i}$ to prune the search space as possible under the current context in light of Property 4 and 5. Here, U_i is the combination of all local constraints at a_i under $Context_i$. Formally,

$$U_i = cd_cost_i \otimes ap_cost_i \quad (7)$$

$$cd_cost_i = \otimes_{a_j \in CD(a_i)} f_{ij} \quad (8)$$

$$ap_cost_i = \sum_{a_j \in AP(a_i)} f_{ij} (Context_i(x_j)) \quad (9)$$

where cd_cost_i is the combination of all constraints between a_i and $CD(a_i)$ and ap_cost_i is the sum of all constraints between a_i and $AP(a_i)$ under the current context.

Theoretically, all the elements in $D^{U_i} \setminus S_i^{U_i}$ should be pruned. However, it is hard to do since pruning some elements requires the joint implementation by $CD(a_i)$ at different branches which search their subspace independently in existing tree-based complete search algorithms. Therefore, we choose to prune some elements from $D^{U_i} \setminus S_i^{U_i}$, which only involves a_i or the joint implementation by a_i and its child. Specifically, a_i removes DV_i computed by Eq. (10) from its domain (i.e., D_i). For each value $d_i \in D_i \setminus DV_i$, a_i suggests its child $a_c \in C(a_i)$ to remove $DV_i^c(d_i)$ computed by Eq. (11) from the domain of a_c (i.e., D_c).

$$DV_i = \{d_i \in D_i \mid (d_i, \mathbf{d}_{-i}) \in D^{U_i} \setminus S_i^{U_i}, \forall \mathbf{d}_{-i} \in D_{-i}^{U_i}\} \quad (10)$$

$$DV_i^c(d_i) = \{d_c \in D_c \mid (d_i, d_c, \mathbf{d}_{-(i,c)}) \in D^{U_i} \setminus S_i^{U_i}, \forall \mathbf{d}_{-(i,c)} \in D_{-(i,c)}^{U_i}\}, a_c \in C(a_i) \quad (11)$$

Here, $D_{-(i,c)}^{U_i} = \times_{x_j \in \text{dims}(U_i) \setminus \{x_i, x_c\}} D_j$.

Take a_2 in Fig. 1(c) for example. Given $Context_2 = \{(x_1, 0)\}$, we have $U_2 = f_{12}(x_1 = 0) \otimes f_{23} \otimes f_{24}$ as shown in Fig. 2 where all the elements in $S_2^{U_2}$ are highlighted in bold. We have $DV_2 = \{2\}$ according to Eq. (10), and $DV_2^3(1) = \{0, 2\}$ according to Eq. (11). Similarly, we have $DV_2^3(0) = \emptyset$, $DV_2^4(0) = \emptyset$ and $DV_2^4(1) = \{0, 2\}$. Accordingly, a_2 obtains all the removed elements shown in gray.

Since $DV_{P(a_i)}^i(Context_i(P(a_i)))$ can be piggybacked by a CPA or VALUE message from $P(a_i)$ and DV_i is computed by itself, a_i can obtain the pruned domain Dom_i by:

$$Dom_i = D_i \setminus (DV_i \cup DV_{P(a_i)}^i(Context_i(P(a_i)))) \quad (12)$$

Algorithm 1 gives the sketch of calculating Dom_i for both synchronous and asynchronous tree-based complete search algorithms when enforcing BIP. Each agent a_i computes cd_cost_i firstly (line 1). Afterwards, it calculates DV_i and $DV_i^c(d_i)$ by calling $Compute_DVs()$ and

x_2	x_3	x_4	U
0	0	0	6
0	0	1	7
0	0	2	4
0	1	0	7
0	1	1	8
0	1	2	5
0	2	0	4
0	2	1	5
0	2	2	2

(a) $x_2 = 0$.

x_2	x_3	x_4	U
1	0	0	9
1	0	1	8
1	0	2	7
1	1	0	8
1	1	1	7
1	1	2	6
1	2	0	7
1	2	1	6
1	2	2	5

(b) $x_2 = 1$.

x_2	x_3	x_4	U
2	0	0	10
2	0	1	10
2	0	2	11
2	1	0	10
2	1	1	10
2	1	2	11
2	2	0	11
2	2	1	11
2	2	2	12

(c) $x_2 = 2$.

■ **Figure 2** U_2 under $Context_2 = \{(x_1, 0)\}$.

■ **Algorithm 1** Calculating Dom_i for a_i .

When Initialization:

```

1 | compute  $cd\_cost_i$  according to Eq. (7)
2 | if  $a_i$  is the root then
3 | |   Compute_DVs()
4 | |   compute  $Dom_i$  according to Eq. (12)

```

When received a CPA or VALUE from $P(a_i)$:

```

5 |   Compute_DVs()
6 |   compute  $Dom_i$  according to Eq. (12)

```

When sending a CPA or VALUE to $a_c \in C(a_i)$:

```

7 |   attach  $DV_i^c(d_i)$  to the CPA or VALUE message

```

Function Compute_DVs():

```

8 |    $DV_i \leftarrow D_i$ 
9 |    $DV_i^c(d_i) \leftarrow D_c, \forall d_i \in D_i, a_c \in C(a_i)$ 
10 |  foreach  $d_{-i} \in D_{-i}^{U_i}$  do
11 |     $d_i = pos_{U_i}(d_{-i})$ 
12 |     $DV_i \leftarrow DV_i \setminus \{d_i\}$ 
13 |     $DV_i^c(d_i) \leftarrow DV_i^c(d_i) \setminus d_{-i[x_c]}, \forall a_c \in C(a_i)$ 

```

Dom_i according to Eq. (12) if it is the root (lines 2–4, 8–13) or receiving a CPA or VALUE message from $P(a_i)$ (line 5–6), and attaches $DV_i^c(d_i)$ to the message when forwarding a CPA or VALUE message to a_c (line 7). *Compute_DVs()* performs the following steps to obtain DV_i and $DV_i^c(d_i)$. Firstly, each agent a_i initializes DV_i to D_i and $DV_i^c(d_i)$ to D_c for each $d_i \in D_i$ and $a_c \in D_c$ (lines 8–9). Then, a_i traverses U_i to filter out elements from DV_i and $DV_i^c(d_i)$ if they do not satisfy Eq. (10) and (11) (lines 10–13), respectively.

3.2 An Example for BIP

We take Fig. 1 as an example to trace BIP running on a tree-based synchronous search algorithm. Table 1 shows the variable update for BIP in the first three cycles. For the sake of simplicity, we omit the variable update for the search algorithm.

Cycle 1: After constructing a pseudo tree shown in Fig. 1(c), the root agent a_1 computes $U_1 = f_{12} \otimes f_{13}$, and then calls *Compute_DVs()* to get DV_1 and $DV_1^2(d_1)$ for itself and its child a_2 , respectively. Afterwards, it computes $Dom_1 = D_1 \setminus DV_1 = \{0, 1\}$ and sends $\{(x_1, 0)\}$ and $DV_1^2(0) = \emptyset$ to its child a_2 via a CPA message. (Assume that a_1 takes its feasible assignment $(x_1, 0)$.)

$$a_1 \rightarrow a_2 : \text{CPA}(\{(x_1, 0)\}, DV_1^2(0))$$

■ **Table 1** The trace of assignments to the variables of BIP.

$Cycle(a_i)$	$Context_i$	DV_i	$DV_i^c(d_i)$	Dom_i	x_i
$1(a_1)$	\emptyset	$\{2\}$	$DV_1^2(0) = \emptyset,$ $DV_1^2(1) = \{0, 2\},$ $DV_1^2(2) = \{0, 1, 2\}$	$\{0, 1\}$	0
$2(a_2)$	$\{(x_1, 0)\}$	$\{2\}$	$DV_2^3(0) = DV_2^4(0) = \emptyset,$ $DV_2^3(1) = DV_2^4(1) = \{0, 2\},$ $DV_2^3(2) = DV_2^4(2) = \{0, 1, 2\}$	$\{0, 1\}$	1
$3(a_3)$	$\{(x_1, 0),$ $(x_2, 1)\}$	$\{0, 1\}$	—	\emptyset	—
$3(a_4)$	$\{(x_1, 0),$ $(x_2, 1)\}$	$\{0, 1\}$	—	\emptyset	—

Cycle 2: When a_2 receives the CPA message from a_1 , it computes $U_2 = f_{12}(x_1 = 0) \otimes f_{23} \otimes f_{24}$, calls $Compute_DVs()$ to get DV_2 , $DV_2^3(d_2)$ and $DV_2^4(d_2)$ for itself, a_3 and a_4 , respectively, and then computes $Dom_2 = D_2 \setminus (DV_2 \cup DV_1^2(0)) = \{0, 1\}$. Next, a_2 takes its feasible assignment $(x_2, 1)$ and sends a CPA message containing $\{(x_1, 0), (x_2, 1)\}$ and $DV_2^3(1)$ to a_3 and a CPA message containing $\{(x_1, 0), (x_2, 1)\}$ and $DV_2^4(1)$ to a_4 .

$$a_2 \rightarrow a_3 : CPA(\{(x_1, 0), (x_2, 1)\}, DV_2^3(1))$$

$$a_2 \rightarrow a_4 : CPA(\{(x_1, 0), (x_2, 1)\}, DV_2^4(1))$$

Cycle 3: Upon receipt of the CPA message from a_2 , a_3 computes DV_3 by calling $Compute_DVs()$ ⁵, and gets $Dom_3 = D_3 \setminus (DV_3 \cup DV_2^3(1)) = \emptyset$ which means $Context_3$ can not lead to the optimal solution and should be changed. Therefore, it backtracks to its parent a_2 with an infinity cost. a_4 performs the same as a_3 .

3.3 Acceptability Testing Mechanism(ATM)

For existing tree-based complete search algorithms, each agent a_i explores its subproblem conditioned on $Context_i$. When enforcing BIP, a_i needs to exploit Dom_i which is computed under the assignments to $AP(a_i)$ and $AP(P(a_i))$ according to Eqs. (10) - (12). However, the assignment to $AP(P(a_i)) \setminus Sep(a_i)$ is not contained in $Context_i$. Consequently, a_i is searching under its running context while $AP(P(a_i)) \setminus Sep(a_i)$ may change their values, which is very common in asynchronous algorithms. In the case, a_i 's search results might be unacceptable. There exist two naïve solutions to the issue. The one is to expand the running context of a_i from the assignment of $Sep(a_i)$ to the ones of $Sep(a_i) \cup AP(P(a_i))$. The other is to remove $DV_{P(a_i)}^i(Context_i(P(a_i)))$ from Eq. (12). Unfortunately, the former could result in more frequent changes of running contexts and thus severely degrade the original algorithms while the latter could lead to missing opportunities to prune the search space.

Instead of changing the running context or BIP, we propose ATM to filter out unacceptable search results to ensure the completeness of the original algorithms when the inconsistent contexts happen. Specifically, for a_i and its child a_c , when $AP(a_i) \setminus Sep(a_c)$ change their values, a_c has to exploit new Dom_c , which puts a_c 's search results under its old Dom_c at risk of unacceptability. For clarity, we denote the old Dom_c and the new Dom_c as $Dom_c(d_i)$ and $Dom'_c(d_i)$ for a given d_i , respectively. We introduce the following rules to determine if a_c 's search results under $Dom_c(d_i)$ (i.e., $lb_i^c(d_i)$ and $ub_i^c(d_i)$) are acceptable or not.

⁵ Note that $Compute_DVs()$ is also applied to leaves since we specify $DV_{-i}^U = \{\emptyset\}$ when $dims(U) \setminus \{x_i\} = \emptyset$.

- **Rule 1:** $lb_i^c(d_i)$ is acceptable under $Dom'_c(d_i)$ if $Dom'_c(d_i) \subseteq Dom_c(d_i)$; otherwise, discarded.
- **Rule 2:** $ub_i^c(d_i)$ is acceptable under $Dom'_c(d_i)$ if $Dom'_c(d_i) \supseteq Dom_c(d_i)$; otherwise, discarded.

Next, we will prove the correction of the two rules.

► **Proposition 6.** *Given d_i , $lb_i^c(d_i)$ produced under $Dom_c(d_i)$ is acceptable under $Dom'_i(d_i)$ if $Dom'_c(d_i) \subseteq Dom_c(d_i)$, and $ub_i^c(d_i)$ produced under $Dom_c(d_i)$ is acceptable under $Dom'_i(d_i)$ if $Dom'_c(d_i) \supseteq Dom_c(d_i)$.*

Proof. Let LB'_c and UB'_c be the search results of a_c under $Dom'_c(d_i)$, and opt'_c be the optimal cost of a_c 's subproblem under $Dom'_c(d_i)$. To prove the proposition, we only need to prove that $lb_i^c(d_i) \leq LB'_c$ if $Dom'_c(d_i) \subseteq Dom_c(d_i)$ and $ub_i^c(d_i) \geq UB'_c$ if $Dom'_c(d_i) \supseteq Dom_c(d_i)$ since $LB'_c \leq opt'_c \leq UB'_c$. Next, we will firstly prove $lb_i^c(d_i) \leq LB'_c$ if $Dom'_c(d_i) \subseteq Dom_c(d_i)$.

As LB'_c is the search result under $Dom'_c(d_i)$, according to Eq. (1), we have

$$LB'_c = \min_{d_c \in Dom'_c(d_i)} \{LB_c(d_c)\}$$

Since $lb_i^c(d_i) = LB_c$ (LB_c is actually obtained under $Dom_c(d_i)$) and $Dom'_c(d_i) \subseteq Dom_c(d_i)$, we have

$$\begin{aligned} lb_i^c(d_i) &= \min_{d_c \in Dom_c(d_i)} \{LB_c(d_c)\} \\ &= \min_{d_c \in (Dom_c(d_i) \setminus Dom'_c(d_i)) \cup Dom'_c(d_i)} \{LB_c(d_c)\} \\ &= \min\left(\min_{d_c \in Dom'_c(d_i)} LB_c(d_c), \min_{d_c \in Dom_c(d_i) \setminus Dom'_c(d_i)} LB_c(d_c)\right) \\ &= \min(LB'_c, \min_{d_c \in Dom_c(d_i) \setminus Dom'_c(d_i)} LB_c(d_c)) \leq LB'_c \end{aligned}$$

Similarly, we can conclude $ub_i^c(d_i) \geq UB'_c$ if $Dom_c(d_i) \subseteq Dom'_c(d_i)$. Thus, Proposition 6 is proved. ◀

To execute Rule 1 and 2, a_i needs to obtain $Dom_c(d_i)$ and $Dom'_c(d_i)$. Here, $Dom_c(d_i)$ can be piggybacked by a COST message from a_c and $Dom'_c(d_i)$ can be obtained by:

$$Dom'_c(d_i) = D_c \setminus (DV_c \cup DV_i^c(d_i))$$

where $DV_i^c(d_i)$ is computed by a_i based on Eq. (11) and DV_c can also be piggybacked by a COST message from a_c . When the search results are unacceptable, a_i attaches a Boolean variable $ReqCost_i^c(d_i)$ to the VALUE message to request a COST message from a_c .

$$\begin{aligned} &\text{VALUE}(a_i, d_i, ID, TH, \mathbf{DV_i^c(d_i)}, \mathbf{ReqCost_i^c(d_i)}) \\ &\text{COST}(a_i, context_i, LB_i, UB_i, ThReq, \mathbf{Dom_i}, \mathbf{DV_i}) \end{aligned}$$

■ **Figure 3** Messages of BnB-ADOPT⁺ when enforcing BIP.

Algorithm 2 presents the sketch of acceptability testing mechanism for BnB-ADOPT⁺ [13] (i.e., a version of BnB-ADOPT which removes most of the redundant messages) when enforcing BIP. Here, we attach Dom_i and DV_i to a COST message, and $DV_i^c(d_i)$ and $ReqCost_i^c(d_i)$ to a VALUE message. Figure 3 shows the modified messages where the attached items are bold. Accordingly, we make the following adjustment in processing VALUE and COST messages. Upon receipt of a COST message from its child or a VALUE message from its parent, a_i needs to check if the search results LB_c and UB_c are acceptable by Rule 1 and 2 (lines 14–24,

■ **Algorithm 2** Acceptability testing mechanism for a_i .

```

When received a COST from  $a_c \in C(a_i)$ :
14 | if  $Context_i$  is compatible with  $Context_c$  then
15 |    $d_i = Context_c(a_i)$ 
16 |   if meet Rule 1 for  $a_c$  then
17 |      $lb_i^c(d_i) \leftarrow \max\{lb_i^c(d_i), LB_c\}$ 
18 |   if meet Rule 2 for  $a_c$  then
19 |     if  $Dom_c = \emptyset$  then
20 |        $ub_i^c(d_i) \leftarrow \infty$ 
21 |     else
22 |        $ub_i^c(d_i) \leftarrow \min\{ub_i^c(d_i), UB_c\}$ 
23 |   if not meet Rule 1 or Rule 2 for  $a_c$  then
24 |      $ReqCost_i^c(d_i) \leftarrow true$ 

When received a VALUE from  $P(a_i)$ :
25 | if  $Context_i$  is compatible with  $Context_c$  then
26 |   foreach  $a_c \in C(a_i), d_i \in D_i$  do
27 |     if not meet Rule 1 for  $a_c$  then
28 |        $lb_i^c(d_i) \leftarrow 0,$ 
29 |     if not meet Rule 2 for  $a_c$  then
30 |        $ub_i^c(d_i) \leftarrow \infty,$ 
31 |     if not meet Rule 1 or Rule 2 for  $a_c$  then
32 |        $ReqCost_i^c(d_i) \leftarrow true$ 

When sending a VALUE to  $a_c \in C(a_i)$ :
33 | if  $ReqCost_i^c(d_i) = true$  then
34 |   attach  $ReqCost_i^c(d_i)$  to the VALUE to request a COST from  $a_c$ 
35 |    $ReqCost_i^c(d_i) \leftarrow false$ 

When sending a COST to  $P(a_i)$ :
36 | if  $Dom_i = \emptyset$  then
37 |    $LB_i \leftarrow \infty$ 
38 |    $UB_i \leftarrow \infty$ 

```

25–32). If LB_c and UB_c are unacceptable, a_i sets $ReqCost_i^c(d_i)$ to true to request the latest search results of a_c by a VALUE message to a_c (lines 33–35). Besides, a_i sets its lower and upper bounds to infinity and sends them to its parent by a COST message if $Dom_i = \emptyset$ (lines 36–38).

3.4 Tradeoff

When deploying BIP into existing tree-based complete search algorithms, each agent a_i needs to store its cost table cd_cost_i which requires the memory consumption of $d_{max}^{|CD(a_i)|+1}$. Thus, we introduce a parameter k to specify the maximum memory budget (i.e., d_{max}^k) for each agent and only the agents with $|CD(a_i)| + 1 < k$ can perform BIP. Hereby, we trade pruning efficiency for memory consumption. In addition, we allocate the remaining memory (i.e., $d_{max}^{k-|CD(a_i)|-1}$) to store DV_i and $DV_i^c(d_i)$ to avoid repeated computation, where least recently used (LRU) policy is used to replace the old entry with the lasted one.

3.5 Complexity

When applied to existing tree-based complete search algorithms, BIP does not introduce any new messages, and only adds some extra attachments which only require linear memory to forwarding messages. Specifically, we attach $DV_i^c(d_i)$ to a CPA message for synchronous tree-based algorithms, $DV_i^c(d_i)$ and $ReqCost_i^c(d_i)$ to a VALUE message and Dom_i and DV_i to a COST message for asynchronous tree-based algorithms.

As for the memory consumption of each agent a_i , it is $O(|D_i|)$ if $|CD(a_i)| + 1 > k$ since a_i does not need to perform BIP and thus only stores Dom_i . Otherwise, it is $O(d_{max}^k)$ for synchronous tree-based algorithms and $O(d_{max}^k + |C(a_i)|d_{max}^2)$ for asynchronous tree-based algorithms. Here, d_{max}^k is the memory consumption as mentioned in Subsection 3.4 and the memory of $|C(a_i)|d_{max}^2$ is used for storing $Dom'_c(d_i)$ and $DV_c(d_i)$ when performing ATM.

For each agent a_i , it needs to traverse U_i whose size is $O(d_{max}^{|CD(a_i)|+1})$ to compute $S_i^{U_i}$. Then, it can obtain DV_i and $DV_i^c(d_i)$ for each $a_c \in C(a_i)$ by enumerating each element in $S_i^{U_i}$ whose size is $d_{max}^{|CD(a_i)|}$. Thus, the overall computational complexity of a_i is $O(d_{max}^{|CD(a_i)|+1} + (1 + |C(a_i)|)d_{max}^{|CD(a_i)|})$

4 Theoretical Results

In the section, we prove the correction of BIP. Firstly, we will prove Eq. (4) can be suitable for DCOPs.

Given a DCOP and its pseudo tree, let us consider the following three cost tables regarding the subproblem rooted at a_i under the current context $Context_i$. U_i is the combination of all local constraints with a_i and computed by Eq. (7). U_i^{irr} is the combination cost table of all constraints without a_i in the subproblem. That is,

$$U_i^{irr} = \otimes_{a_j \in Desc(a_i)} (\otimes_{a_k \in AP(a_j) \setminus (Sep(a_i) \cup \{a_i\})} f_{jk} \otimes (\otimes_{a_k \in AP(a_j) \cap Sep(a_i)} f_{jk}(Context_i(x_k)))) \quad (13)$$

U_i^{sub} is the combination cost table of all constraints in a_i 's subproblem after eliminating $Desc(a_i) \setminus CD(a_i)$. That is,

$$U_i^{sub} = \min_{Desc(a_i) \setminus CD(a_i)} (U_i \otimes U_i^{irr}) \quad (14)$$

► **Lemma 7.** $U'' = \min_{Desc(a_i) \setminus CD(a_i)} U_i^{irr}$ is a cost table such that $U_i^{sub} = U_i \otimes U''$ and $dims(U'') \subseteq dims(U_i) \setminus \{x_i\}$.

Proof. According to Eq. (14), we have

$$\begin{aligned} U_i^{sub} &= \min_{Desc(a_i) \setminus CD(a_i)} (U_i \otimes U_i^{irr}) \\ &= U_i \otimes \min_{Desc(a_i) \setminus CD(a_i)} U_i^{irr} \end{aligned}$$

The equation from the first step to the second step holds since $dims(U_i) = CD(x_i) \cup \{x_i\}$ according to Eq. (7) and $(CD(x_i) \cup \{x_i\}) \cap (Desc(a_i) \setminus CD(a_i)) = \emptyset$. Further, according to Eq. (13), we have $dims(U_i^{irr}) = Desc(a_i)$ and thus $dims(U'') \subseteq dims(U_i) \setminus \{x_i\}$. Therefore, Lemma 7 is proved. ◀

► **Lemma 8.** There exists at least one element in $S_i^{U_i}$ that can be extended to the optimal solution of a_i 's subproblem.

Proof. According to Property 4, we have $\exists \mathbf{d} \in S_i^{U^{sub}}$, s.t. $U_i^{sub}(\mathbf{d}) = \min_{\mathbf{d}' \in D^{U^{sub}}} U_i^{sub}(\mathbf{d}')$.

Further, according to Lemma 7 and Property 5, we have $S_i^{U_i} = S_i^{U^{sub}}$ and $\exists \mathbf{d} \in S_i^{U_i}$, s.t. $U_i^{sub}(\mathbf{d}) = \min_{\mathbf{d}' \in D^{U^{sub}}} U_i^{sub}(\mathbf{d}')$. Therefore, we can conclude that the optimal solution of the subproblem rooted at a_i is the join of \mathbf{d} and the optimal assignment to $Desc(a_i) \setminus CD(a_i)$ from Eq. (14). Thus, Lemma 8 is proved. \blacktriangleleft

► **Theorem 9.** *There exists an optimal solution in the remaining space returned by BIP.*

Proof. For any given context $Context_i$, there exists one element in $S_i^{U_i}$ that can be extended to the optimal solution of the subproblem rooted at a_i according to Lemma 8, and BIP does not prune any elements in $S_i^{U_i}$ according to Eqs. (10) and (11). Thus, Theorem 9 is proved. \blacktriangleleft

5 Empirical Evaluation

5.1 Experimental Configuration

In order to demonstrate its effect on distributed search, BIP is applied to BnB-ADOPT⁺-FDAC, PT-FB and HS-CAI, named BnB-ADOPT⁺-FDAC+BIP, PT-FB+BIP and HS-CAI+BIP, respectively. In our experiments, we will compare these BIP-based algorithms with their originals and RMB-DPOP [5] on two types of problems, i.e., random DCOPs and scale-free networks. RMB-DPOP is the latest best-performing algorithm in the DPOP family. We consider four configurations, and the first two are sparse and dense configurations for random DCOPs. In more detail, we set the graph density to 0.2, the domain size to 3 and the number of agents varying from 22 to 32 for 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 for the dense configuration. The third configuration is the random DCOPs with 22 agents, the graph density of 0.2 and the domain size varying from 3 to 8. In the fourth configuration, we consider the scale-free networks whose degree distribution follows a power law. We generate the instances by BA model [2], where we set the number of agents to 26, the domain size to 3 and m_0 to 10, and vary m_1 from 2 to 8.

In our experiments, we use the number of messages (Msgs) and network load (NL, i.e., the size of total information exchanged) to measure the traffic overheads, and the NCLOs [20] to measure the hardware-independent runtime where the logical operations in the inference and the search are accesses to utilities and constraint checks, respectively. In order to capture the computation overhead introduced by BIP, the accesses to U_i , DV_i and $DV_i^c(d_i)$ are also counted into the NCLOs for the BIP-based algorithms. For each experiment, we generate 50 instances randomly with the integer constraint costs in the range of 0 to 100, and report the average over all instances. Moreover, we choose $k = 4$ and $k = 8$ as the low and high memory budget for HS-CAI, RMB-DPOP and BIP, respectively. For fairness, we set the memory for BIP to be the same as the one for HS-CAI (i.e., $O(|C(a_i)|d_{max}^k)$). 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

Figure 4 presents the experimental results under different numbers of agents on the sparse configuration, and the corresponding improvement over the originals is displayed in the first two rows of Table 2 where the numbers greater than zero are shown in bold. It can be seen

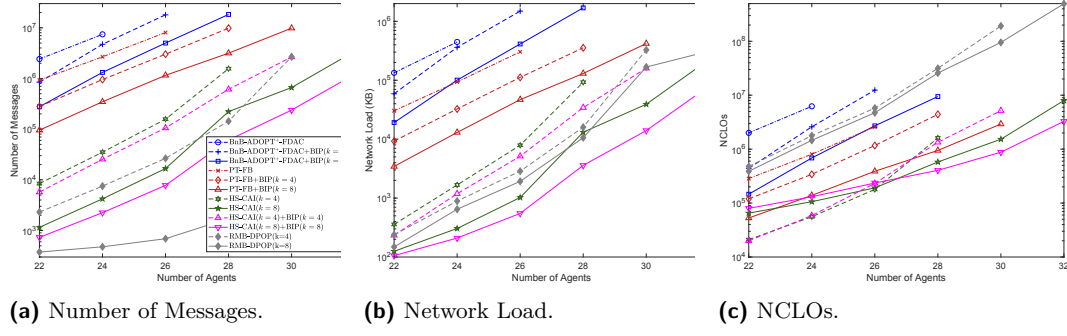


Figure 4 Performance comparison under different numbers of agents on the sparse configuration.

Table 2 The improvement of the BIP-based algorithms over their respective originals.

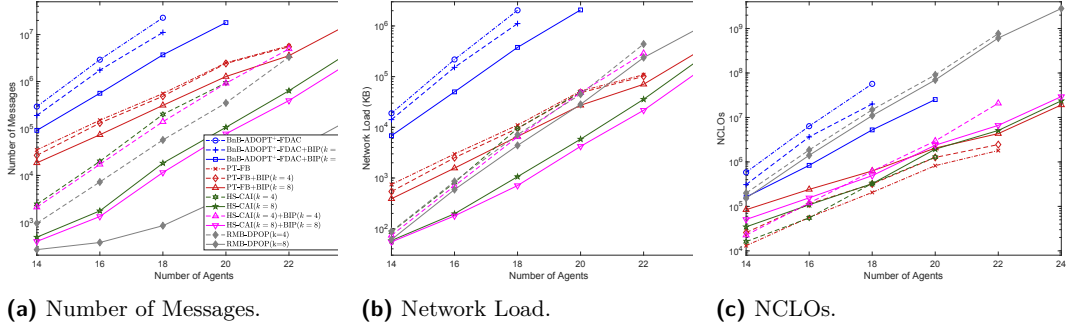
Configuration	k	BnB-ADOPT ⁺ -FDAC+BIP			PT-FB+BIP			HS-CAI+BIP		
		Msgs(%)	NL(%)	NCLOs(%)	Msgs(%)	NL(%)	NCLOs(%)	Msgs(%)	NL(%)	NCLOs(%) ^{6,7}
Sparse	4	36 ~ 65	29 ~ 55	58 ~ 77	62 ~ 69	63 ~ 69	54 ~ 58	28 ~ 61	30 ~ 63	-29 ~ 17
	8	82 ~ 88	77 ~ 85	89 ~ 92	85 ~ 90	84 ~ 88	81 ~ 85	37 ~ 73	16 ~ 73	-23 ~ 59
Dense	4	35 ~ 51	25 ~ 45	43 ~ 65	3 ~ 23	7 ~ 23	-101 ~ -36	2 ~ 30	3 ~ 32	-131 ~ -40
	8	69 ~ 84	64 ~ 81	72 ~ 91	38 ~ 51	35 ~ 47	-542 ~ -140	38 ~ 40	7 ~ 41	-51 ~ -22
Varying	4	57 ~ 65	46 ~ 55	72 ~ 80	49 ~ 70	50 ~ 70	30 ~ 66	19 ~ 32	17 ~ 35	-86 ~ 5
Domain Size	8	85 ~ 89	82 ~ 86	90 ~ 93	73 ~ 90	89 ~ 72	47 ~ 81	19 ~ 27	0 ~ 16	-25 ~ -11
Scale-Free	4	52 ~ 61	21 ~ 43	56 ~ 72	35 ~ 87	37 ~ 86	-2 ~ 86	28 ~ 35	32 ~ 36	-39 ~ -19
Networks	8	68 ~ 84	62 ~ 67	53 ~ 90	71 ~ 94	71 ~ 93	57 ~ 87	30 ~ 67	28 ~ 59	12 ~ 28

that the BIP-based algorithms exhibit a great advantage on all the metrics in most cases and the advantage expands as k increases. This is because more agents performing BIP can lead to better pruning efficiency under larger k . Moreover, the BIP-based algorithms can scale up to larger problems when $k = 4$ and their scalability is further enhanced when $k = 8$. In more detail, BnB-ADOPT⁺-FDAC can only solve problems with the number of agents no greater than 24, and ADOPT⁺-FDAC+BIP can scale up to 26 and 28 when $k = 4$ and $k = 8$, respectively. The similar phenomenon can be found from PT-FB+BIP and HS-CAI($k = 4$)+BIP($k = 4$). In addition, RMB-DPOP has a great advantage over the search algorithms on the number of messages, but performs worse than all the search algorithms except BnB-ADOPT⁺-FDAC and BnB-ADOPT⁺-FDAC+BIP($k = 4$) in terms of the NCLOs. Besides, when $k = 8$, HS-CAI is superior to RMB-DPOP in terms of the network load in most cases and HS-CAI+BIP greatly expands the superiority of HS-CAI over RMB-DPOP.

Figure 5 presents the experimental results under different numbers of agents on the dense configuration, and the third and fourth rows of Table 2 show the corresponding improvement over the originals. It can be seen that the BIP-based algorithms also perform better than their originals in terms of both the number of messages and network load. However, the

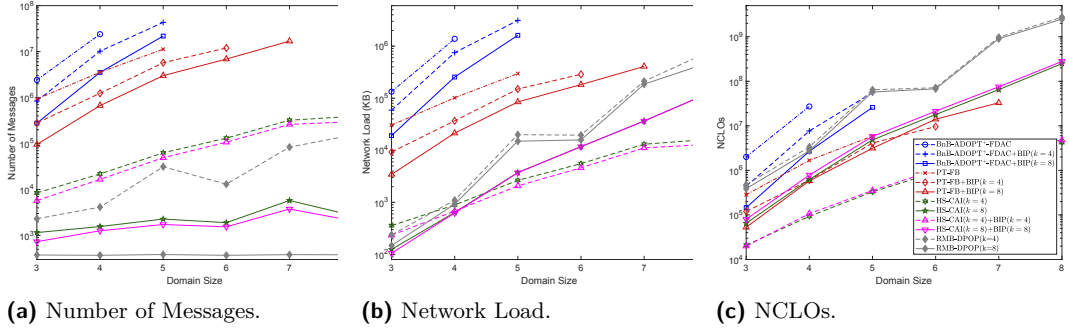
⁶ In the sparse configuration, HS-CAI+BIP is superior to HS-CAI on the NCLOs and the superiority expands as the number of agents increases when the number of agents is greater than 26.

⁷ In the configuration of varying domain size, we set the number of agents to 22 and HS-CAI+BIP is inferior to HS-CAI on the NCLOs, while HS-CAI+BIP will be superior to HS-CAI on this metric when the number of agents is greater than 26, which can be seen from Figure 4(c).



■ **Figure 5** Performance comparison under different numbers of agents on the dense configuration.

gaps narrow compared to the ones on the sparse configuration. This is because less agents at the high positions of the pseudo-tree perform BIP on the dense configuration, which impairs pruning efficiency. In addition, BIP does not always perform well in terms of the NCLOs on the dense configuration. That is because the computational consumption of BIP is exponential to the number of (pseudo) children of an agent and there are more agents with a large number of (pseudo) children in the dense configuration. Compared to the search algorithms, the performance of RMB-DPOP is similar to the one on the sparse configuration.



■ **Figure 6** Performance comparison under different domain sizes.

Figure 6 presents the experimental results under different domain sizes, and the corresponding improvement over the originals can be found in the fifth and sixth rows of Table 2. We can see that BIP still works well when facing the problems with larger domain size. However, the improvement gaps narrow as the domain size increases. This is because the proportion of values pruned out by BIP reduces at large domain size. In addition, the BIP-based algorithms can solve the problems with larger domain size than their originals. In more detail, PT-FB can not solve the problems with the domain size greater than 5, while PT-FB+BIP can scale up to the ones with the domain size of 6 when $k = 4$, and further to the ones with the domain size of 7 when $k = 8$. When facing the problems with larger domain size, the performance of RMB-DPOP is similar to the one in the first two configurations. It is worth noting that the number of messages of RMB-DPOP($k = 8$) holds steady as the domain size increases. That is because under this configuration, it performs just like DPOP where the number of messages is linear to the number of agents.

Figure 7 presents the experimental results on scale-free networks and the seventh and eighth rows of Table 2 show the corresponding improvement over the originals. It can be seen that the BIP-based algorithms exhibit a great advantage over their originals on all the metrics in most cases and the advantage expands as k increases, which is similar to

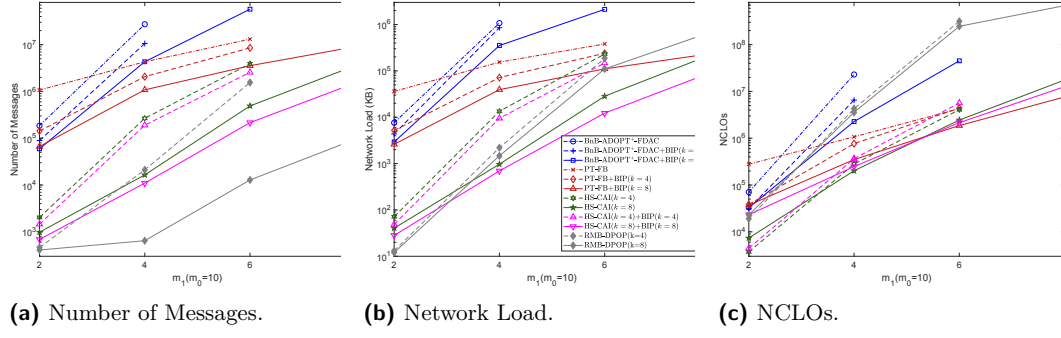


Figure 7 Performance comparison on scale-free networks.

the results on random DCOPs. In addition, when $k = 8$, both BnB-ADOPT⁺-FDAC+BIP and PT-FB+BIP can scale up to the problems with larger m_1 than their originals. The performance of RMB-DPOP is similar to the one on random DCOPs.

From Table 2, we can see that BIP can improve the tree-based complete search algorithms in terms of both the number of messages and network load in all the experimental configurations. This is because BIP can significantly reduce the search space without introducing any new messages, and only adds some extra attachments which only require linear memory to forwarding messages. Thus, BIP is well suited for some real-world applications that are equipped with devices with the limited memory and desire for lower communication overheads. In addition, BIP can greatly improve the search-based algorithms under the sparse configuration on all the metrics. Thus, BIP is also well suited for solving the real-world applications with low graph density.

6 Conclusion

Complete search algorithms for DCOPs depend solely on bounds to prune the search space. However, obtaining strong lower bounds come at a high price. The paper presents a novel pruning technique named BIP which can independently reduce the search space only by means of local knowledge and running contexts. To the best of our knowledge, BIP is the first pruning technique independent of bounds for tree-based complete search algorithms to solve a DCOP. Moreover, our proposed BIP can be easily applied to any existing tree-based complete search algorithms for DCOPs with minor modifications. We theoretically prove the correctness of our technique and our empirical evaluation confirms its great superiority. It is worth noting that our proposed BIP is not specific to tree-based complete search algorithms for DCOPs and can be easily adapted to other backtracking search algorithms for distributed and centralized optimization problems.

References

- 1 James Atlas, Matt Warner, and Keith Decker. A memory bounded hybrid approach to distributed constraint optimization. In *Proceedings 10th International Workshop on DCR*, pages 37–51, 2008.
- 2 Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999. doi:10.1126/science.286.5439.509.
- 3 Ismel Brito and Pedro Meseguer. Improving DPOP with function filtering. In *Proceedings of the 9th AAMAS*, pages 141–148, 2010.

- 4 Dingding Chen, Yanchen Deng, Ziyu Chen, Wenxin Zhang, and Zhongshi He. HS-CAI: A hybrid dcop algorithm via combining search with context-based inference. In *Proceedings of the 34th AAAI*, pages 7087–7094, 2020.
- 5 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 AAMAS*, pages 249–257, 2020.
- 6 Martin C Cooper, Simon De Givry, Martí Sánchez, Thomas Schiex, Matthias Zytnicki, and Tomas Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8):449–478, 2010. doi:10.1016/j.artint.2010.02.001.
- 7 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 AAMAS*, pages 1506–1514, 2019.
- 8 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. doi:10.1007/s10458-013-9225-1.
- 9 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 AAMAS*, volume 2, pages 639–646, 2008.
- 10 Ferdinando Fioretto, Enrico Pontelli, and William Yeoh. Distributed constraint optimization problems and applications: A survey. *Journal of Artificial Intelligence Research*, 61:623–698, 2018. doi:10.1613/jair.5565.
- 11 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 AAMAS*, pages 999–1007, 2017.
- 12 Patricia Gutierrez and Pedro Meseguer. BnB-ADOPT+ with several soft arc consistency levels. In *Proceedings of the 19th ECAI*, pages 67–72, 2010.
- 13 Patricia Gutierrez and Pedro Meseguer. Saving redundant messages in BnB-ADOPT. In *Proceedings of the 24th AAAI*, pages 1259–1260, 2010.
- 14 Katsutoshi Hirayama and Makoto Yokoo. Distributed partial constraint satisfaction problem. In *International Conference on Principles and Practice of Constraint Programming*, pages 222–236, 1997. doi:10.1007/BFb0017442.
- 15 Yoonheui Kim and Victor Lesser. DJAO: A communication-constrained DCOP algorithm that combines features of ADOPT and Action-GDL. In *Proceedings of the 28th AAAI*, pages 2680–2687, 2014.
- 16 Omer Litov and Amnon Meisels. Forward bounding on pseudo-trees for DCOPs and ADCOPs. *Artificial Intelligence*, 252:83–99, 2017. doi:10.1016/j.artint.2017.07.003.
- 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. doi:10.1007/0-387-27972-5_6.
- 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 3rd AAMAS*, volume 1, pages 310–317, 2004.
- 19 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. doi:10.1016/j.artint.2004.09.003.
- 20 Arnon Netzer, Alon Grubshtein, and Amnon Meisels. Concurrent forward bounding for distributed constraint optimization problems. *Artificial Intelligence*, 193:186–216, 2012. doi:10.1016/j.artint.2012.09.002.
- 21 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. doi:10.1613/jair.1.11400.

- 22 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*, 8(5):69, 2017. doi:10.1145/3066156.
- 23 Adrian Petcu and Boi Faltings. Approximations in distributed optimization. In *International Conference on Principles and Practice of Constraint Programming*, pages 802–806, 2005. doi:10.1007/11564751_68.
- 24 Adrian Petcu and Boi Faltings. ODPOP: An algorithm for open/distributed constraint optimization. In *Proceedings of the 21th AAAI*, pages 703–708, 2006.
- 25 Adrian Petcu and Boi Faltings. MB-DPOP: A new memory-bounded algorithm for distributed optimization. In *Proceedings of the 20th IJCAI*, pages 1452–1457, 2007.
- 26 Evan A Sultanik, Pragnesh Jay Modi, and William C Regli. On modeling multiagent task scheduling as a distributed constraint optimization problem. In *Proceedings of the 20th IJCAI*, pages 1531–1536, 2007.
- 27 Meritxell Vinyals, Juan A Rodriguez-Aguilar, and Jesús Cerquides. Generalizing DPOP: DPOP, a new complete algorithm for DCOPs. In *Proceedings of the 8th AAMAS*, pages 1239–1240, 2009.
- 28 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. doi:10.1016/j.artint.2004.10.004.

Data Driven VRP: A Neural Network Model to Learn Hidden Preferences for VRP

Jayanta Mandi ✉ 🏠 

Data Analytics Laboratory, Vrije Universiteit Brussel, Belgium

Rocsildes Canoy ✉ 

Data Analytics Laboratory, Vrije Universiteit Brussel, Belgium

Víctor Bucarey ✉ 🏠 

Institute of Engineering Sciences, Universidad de O'Higgins, Rancagua, Chile

Tias Guns ✉ 🏠

Data Analytics Laboratory, Vrije Universiteit Brussel, Belgium

Department of Computer Science, KU Leuven, Belgium

Abstract

The traditional Capacitated Vehicle Routing Problem (CVRP) minimizes the total distance of the routes under the capacity constraints of the vehicles. But more often, the objective involves multiple criteria including not only the total distance of the tour but also other factors such as travel costs, travel time, and fuel consumption. Moreover, in reality, there are numerous implicit preferences ingrained in the minds of the route planners and the drivers. Drivers, for instance, have familiarity with certain neighborhoods and knowledge of the state of roads, and often consider the best places for rest and lunch breaks. This knowledge is difficult to formulate and balance when operational routing decisions have to be made.

This motivates us to learn the implicit preferences from past solutions and to incorporate these learned preferences in the optimization process. These preferences are in the form of arc probabilities, i.e., the more preferred a route is, the higher is the joint probability. The novelty of this work is the use of a neural network model to estimate the arc probabilities, which allows for additional features and automatic parameter estimation. This first requires identifying suitable features, neural architectures and loss functions, taking into account that there is typically few data available. We investigate the difference with a prior weighted Markov counting approach, and study the applicability of neural networks in this setting.

2012 ACM Subject Classification Computing methodologies → Planning and scheduling

Keywords and phrases Vehicle routing, Neural network, Preference learning

Digital Object Identifier 10.4230/LIPIcs.CP.2021.42

Supplementary Material *Software (Source Code and Anonymized Data)*: <https://github.com/JayMan91/CP2021-Data-Driven-VRP>

Funding Research supported by the FWO Flanders project Data-driven logistics (FWO-S007318N).

1 Introduction

Although the Vehicle Routing Problem (VRP) and its many variants have been extensively studied in the literature, the “theoretical optimal” solution often does not meet the expectations of the route planners and the drivers. This is because in real-life operations, the acceptability of a route is dependent not only on distance, travel time or fuel consumption, which have been studied in the literature, but also on multiple factors which are difficult to put in the objective function. A study by [2] has revealed that local drivers prefer routes that are not optimal in terms of travel time or cost. The drivers take into account several factors which are not in the objective function such as traffic congestion and availability of



© Jayanta Mandi, Rocsildes Canoy, Víctor Bucarey, and Tias Guns;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 42; pp. 42:1–42:17

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

parking and fuel stations. This highlights the necessity of *preference-based* routing where the objective is to minimize the travel cost as perceived by the drivers and the planners. To put it another way, we can see it as maximizing the utility of the drivers and the planners.

In this work, we propose VRP solutions which are acceptable to the route planners and the drivers. We start from the setting studied in [1], which proposes the maximum likelihood routing. Maximum likelihood routing considers the transition probabilities between the stops as *revealed preferences* of the drivers and the planners and finds the maximum utility route by maximizing the joint transition probabilities. To estimate the transition probabilities, [1] uses a Markov counting approach which enumerates the past solutions for each realized route. Their approach uses only the past solutions for probabilities estimation but cannot make use of contextual features such as day of week. We extend their framework to use a neural network model to estimate the transition probabilities before finding the route by applying maximum likelihood routing. The motivation behind using the neural network model is to generate a better estimation of the cost vector by using historical as well as contextual information in the neural network model.

We start with a neural network model which is trained using both contextual information and past solutions. We also include the Markov prediction as a feature in the neural network and observe improvement in the solution quality. Finally, we choose a parsimonious architecture in order to avoid overfitting and with this we are able to outperform [1].

Contributions.

- We formulate the challenge of neural network-based learning of hidden preferences from moderately sized data, in a way that is compatible with existing VRP solvers.
- We investigate different features and architectures for such a neural network, more specifically arc-based linear models combined into per-node probabilistic estimates.
- We investigate how we can combine the Markov model and neural network, e.g., by considering the Markov predictions as an input to the neural network model.
- We propose two loss functions that allow for gradient-descent learning: one based on standard multi-class losses and another based on decision-focused learning that incorporate the VRP solving into the loss function.

2 Related Work

The VRP [3] has been studied with its many different variations. Traditional VRP minimizes a tangible objective such as operational costs [10], travel time [13], fuel consumption or carbon emission [26, 19]. Although multiple aspects of the assignment schedules of the drivers such as route balancing [14] have been studied, learning and optimizing drivers' preferences has recently received increasing attention.

The preferences of the drivers can be considered by including them in the objective function. This can be treated in a multi-objective VRP [11] setting, such as forming an objective function as a weighted sum or finding the set of Pareto optimal solutions based on standard multi-objective evolutionary algorithms [21]. However, the preferences of the drivers are implicit [22] and in most cases, explicit formalization of these preferences is not possible in practice.

Authors in [1] tackled the problem from a different perspective: they introduce a weighted Markov model to learn the preferences. This approach avoids the explicit specification of the preference constraints and the implicit sub-objectives. The Markovian model is built using preferences learned from past solutions, which the planners have constructed by modifying solutions given by off-the-self solvers. Contrary to their work, we use a neural network model to learn the drivers' preferences, allowing a more flexible and general framework.

Learning preferences for drivers have been focused mainly in the setting of one single origin and destination. TRIP [15] leverages past GPS data to learn drivers' preferences by comparing the ratios of the drivers' travel time to the average travel time, and with that, it generates routes that mimic the ones chosen by the drivers. The approach in [6] also deduces driving preferences from GPS traces and models them into the weights of a linear programming formulation, which is then optimized to generate new route suggestions. The authors in [8] are able to enhance the quality of the solutions by considering different routing preferences that vary depending on the contexts. While we do not provide an explicit representation of the preferences, we assume that the preferences can be expressed in terms of probabilities (or utilities) of the arcs in the graph.

Decision-focused learning [25, 5], which combines gradient-based neural network and optimization into a single framework, has recently received much attention in operations research. In this setup, the outputs of the neural network are fed into the optimization module as one of the inputs. The novelty of this approach is that it trains the neural network model while considering the objective value in the optimization problem. Decision-focused learning of submodular optimization problems, zero-sum games and SAT problems have been studied in [25, 17, 24] respectively. The approach proposed by [20] also combines a neural network model with any given optimization oracle via "implicit interpolation". Ours is the first work which uses this framework for learning preferences in the context of vehicle routing.

3 Preliminaries

3.1 Problem Description

In this work, we are interested in the route planning process of an actual small transportation company. The route planners in the company are responsible for organizing tours for a fleet of vehicles in order to deliver goods to the customers. Although they use a commercial route optimization software to produce routes that are optimal in terms of route length and travel time, they are hardly satisfied with the solutions. Solutions have to be modified to come up with a tour which is acceptable to the planners, drivers, and other stakeholders. In this way, the planners are implicitly optimizing the utilities of all those involved.

One way to approach this problem is to explicitly define the set of objectives. However, it is nearly impossible to model such personal preferences. We observe that the planners start from past realized routes because they require minimal modifications compared to the "theoretical optimal" routes. Therefore, in a way, the past solutions capture the preferences (or the utilities) of the planners and the drivers. Our objective in this work is to learn the latent preferences of the drivers and the route planners for vehicle routing using a neural network model and propose tours which are acceptable to the planners. More specifically, we will focus on learning preferences at the arc level. We consider the transition probabilities as *revealed preferences*. We use neural network to output the transition probabilities between every pair of nodes. The advantage of this formulation is that we can use the negative log probability in place of a traditional travel cost in any existing VRP solver.

Challenges. A machine learning model learns from training instances. In our case with the company data, each instance is realized in a day. However, due to functional and operational reasons, tours are not organized each day. Putting that into perspective, it would take more than 6 months to collect only 180 training instances. Consequently, we are not in a state

to use a neural network model trained with thousands of training instances. Hence, we have to be particularly careful about using a neural network model with a large number of parameters, as such network is prone to overfitting with small data.

Another challenge in this case is that not all customers raise a demand request with equal frequency. In fact, some customers have daily requests, whereas others raise requests only once or twice in a month. Depending on which set of customers raise a request, there can be considerable changes in the tour. We also observe a weekly pattern in the tours, i.e., tours of one weekday are different from the other days but very similar to those from the same weekday of the previous weeks. Therefore, learning the weekly patterns from a limited number of weeks poses another challenge.

3.2 Formalization

We begin by formalizing the objective and the data structures. Formally, on a given day t , S^t is the set of stops to be served by a number m^t of homogeneous vehicles. We represent $S^t \doteq \{0, 1, \dots, n\}$, where 0 represents the depot, and the other nodes represent the customers. Let A^t define the set of all arcs in S^t .

We call \mathbf{x}^t a **routing** with respect to S^t with m^t homogeneous vehicles, if \mathbf{x}^t contains a set of at most m^t tours in S^t with each tour starting from and ending at the depot 0 and each node in S^t is visited exactly once to satisfy its demand request. Additionally, a feasible routing should ensure that the total demand allocated to each vehicle does not exceed its capacity Q . Let $\mathcal{X}_{S^t}^{m^t}$ denote all feasible routings of m^t vehicles over S^t . The objective in standard CVRP is to minimize the total travel costs of the routing. We remark that the depot is fixed and always present but the set of stops S^t changes from one day to another as not all customers raise a demand request each day.

For learning the preferences from past data, we are given a dataset $\mathcal{H} = \{(S^t, z^t, \mathbf{X}^t)\}_{t=1}^T$. Each instance in the dataset is a tuple where t is a timestamp, S^t is the set of stops served at t , \mathbf{X}^t denotes the actual preferred routing created by the planners, and z^t are feature variables such as the demand of each stop, the number of vehicles used, the day of the week, or some other known parameters. Hereafter, we will use the symbols without the suffix t to avoid notational complexity.

3.3 Transition Probabilities

Explicit specification of the preferences of the drivers and the planners would result in a complex model with a large number of parameters to tune. Instead, in this paper, we use the framework of [1], which captures the preferences of the route planners and the drivers using transition probabilities. In more formal terms, we learn a model which assigns probabilities to all the arcs within the network. Our hope is that these transition probabilities subsume the hidden preferences of the route planners and the drivers. Formally, we learn $\Pr(r|s)$ which denotes the probability of the next stop being r , conditional on the current stop s . We remark that the transition probability would be a function of some temporal and contextual attributes including but not limited to the traditional cost measures.

3.4 Maximum Likelihood Routing

Once the probabilities are learned, we follow the methodology of [1] to find the most *likely* routing from the set of all feasible routings. We call the routing with the highest probability the maximum likelihood routing (MLE routing). Formally,

$$\max_{x \in \mathcal{X}_S^m} \prod_{(s \rightarrow r) \in x} \Pr(r|s). \quad (1)$$

In order to identify the MLE routing, we solve an optimization problem whose feasible region is defined by the following standard CVRP constraints [23].

$$\sum_{r \in V, r \neq s} x_{sr} = 1 \quad s \in S \quad (2)$$

$$\sum_{s \in V, s \neq r} x_{sr} = 1 \quad r \in S \quad (3)$$

$$\sum_{r=1}^n x_{0r} = m \quad (4)$$

$$\text{if } x_{sr} = 1 \Rightarrow u_s + q_r = u_r \quad (s, r) \in A : t \neq 0, s \neq 0 \quad (5)$$

$$q_s \leq u_s \leq Q \quad s \in S \setminus \{0\} \quad (6)$$

$$x_{sr} \in \{0, 1\} \quad (s, r) \in A. \quad (7)$$

(2) and (3) ensure that each customer is served by exactly one vehicle. (5) performs subtour elimination. (6) ensures that the vehicle capacity is respected. We remark that in (4), we use the equality constraint because in practice, the company must use all the available vehicles. The only modification from the standard CVRP is that instead of minimizing the distance, we maximize the joint probability. To transform the product in the objective function into a sum, we consider log probabilities in the objective function and minimize the following:

$$\min_x \sum_{(s,r) \in A} -\log \Pr(r|s) x_{sr} \quad (8)$$

In the subsequent discussions, the (s, r) -th entry of matrix P would contain $\Pr(r|s)$.

3.5 Transition Probability Estimation by Markov Counting

The goal of the Markov Counting approach is to estimate all the conditional probabilities $\Pr(r|s)$ over the set of all stops in the data: $S^{\text{all}} = \bigcup_t S^t$. From conditional probability theory, we have:

$$\Pr(r|s) = \frac{\Pr(s \rightarrow r)}{\Pr(s)}, \quad (9)$$

where $\Pr(s) = \sum_u \Pr(s \rightarrow u)$. By defining the frequency of a transition $(s \rightarrow r)$ in the historical dataset \mathcal{H} as $f_{sr} = \sum_t \mathbb{I}(s \rightarrow r \in \mathbf{X}^t)$, where $\mathbb{I}[\cdot]$ equals 1 if the statement inside the bracket is true and 0 otherwise, the conditional probabilities from the dataset can be estimated by:

$$\Pr(r|s) = \frac{f_{sr}}{\sum_u f_{su}}. \quad (10)$$

We point out that with this formulation, we can solve the standard CVRP which minimizes the distance if we replace $\Pr(r|s)$ by a *distance-based probability* $\Pr_{\text{dist}}(r|s)$:

$$\Pr_{\text{dist}}(r|s) = \frac{e^{-d_{sr}}}{\sum_u e^{-d_{su}}}. \quad (11)$$

The transition probability matrix construction algorithm presented in [1] makes use of *weighing schemes*, where a variable weight w_t is defined for each historical instance in \mathcal{H} . This weight varies according to the properties of the tuple (S^t, z^t, \mathbf{X}^t) . Giving varying weights to

each historical instance affects the way the transition frequencies are counted, hence each weighing scheme results in a different transition matrix. Exponential weighing is one of the most used scheme, where instances far-off in the past receive decaying weights. Therefore, in our experiments, we will compare our approach with the Markov model with the exponential weighing scheme.

4 Learning the Transition Probabilities Using Neural Network

One limitation of the Markov counting approach introduced in section 3.5 is that it only uses past data to arrive at the probabilities. We want the transition probabilities to be a function of other attributes such as the day of week and the distances between stops, among others. This is the motivation behind using a neural network model.

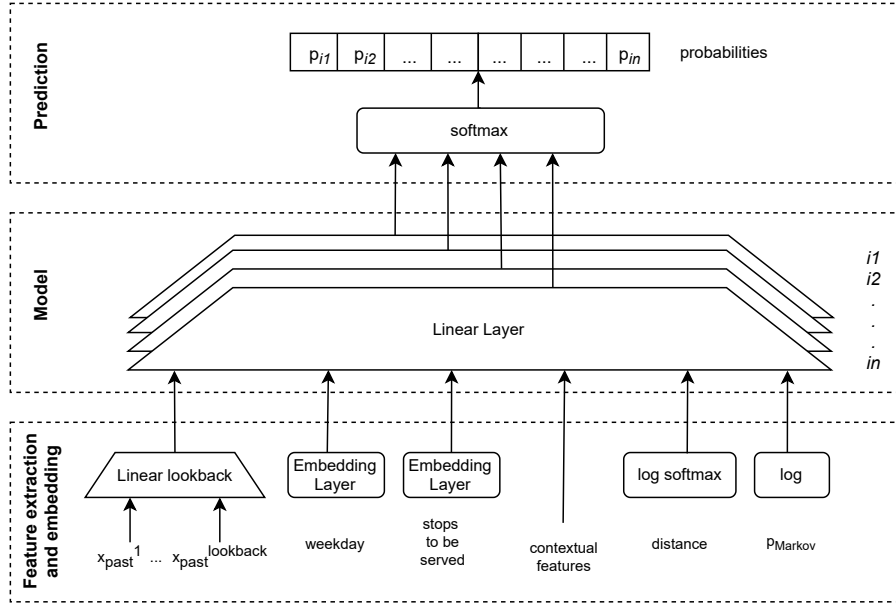
Neural networks are made of interconnected units called neurons. A single neuron takes a series of inputs z_0, \dots, z_n and returns an output o as a function of the inputs $o = f(\sum w_i z_i)$, or in matrix form $o = f(Wz)$, where f is an activation function and w_i 's are the weights. Many choices for the activation function exist – *sigmoid*, *ReLU*, *tanh* are some of the widely used activation functions. A network consists of several layers, and multiple neurons are stacked in each layer where the inputs are connected to each neuron. The output of the layer can be conveniently described in matrix form as $o = f(Wz)$. Here, each row of W corresponds to each neuron. The dimension of output o is controlled by the dimension of matrix W . In a multilayer network, the subsequent layers use the outputs of the preceding layers as inputs. Obviously, the designer has the option to transform the output between two layers.

A multilayer neural network is considered as a universal function approximator [16], which tries to learn the functional relationship between the output and the input. To do so, the parameters of the neural network must be learned using training data. This is done by backpropagating the loss between the predicted output and the target output. During backpropagation, the derivative of the final loss with respect to the weights is computed and then the weights are updated by gradient descent. The choice of the loss function is dependent on the problem at hand. For a multiclass classification approach, categorical cross entropy loss is the preferred choice.

We propose to learn the transition probabilities between the stops from the historical data using a neural network. For a single day t , we have (S^t, z^t, \mathbf{X}^t) as explained in section 3.5.

Feature variables. We want the predicted probabilities to be a function of the feature variables. Different types of features can be considered: time-lagged temporal features, features related to the set of stops to be served (S^t), the distance between the stops and contextual features such as day of week, number of vehicles. The motivation behind using the time-lagged solutions as features is to learn from past solutions. We define the *look back period* (L) as the maximum number of past observations considered in our model. The motivation of this look back period is two-fold: 1. it allows us to model the fact that past observations lose their relevance over time, and 2. to avoid problems of over-fitting due to lack of observations. We can also consider the output of the Markov counting model as a feature, as it subsumes past information. Moreover, this can be computed easily on the fly.

From this discussion, it is evident that some of the features are specific to an arc. This includes the time-lagged features, the distances and the Markov probabilities. On the other hand, features such as day of week and stops to be served are the same across all the stops. All of these are considered as an input to the neural network in Figure 1.



■ **Figure 1** Neural network architecture to estimate transition probabilities from a source node s .

Architecture of the neural network. Our goal is to build a network that estimates the transition probabilities for each arc $(s, r) \in A$. We have $|A| \times |A|$ distance features, $L \times |A| \times |A|$ time lagged features and $|K|$ number of contextual features. Because of limited data, our aim is to build a parsimonious network with as few parameters as possible.

Authors in [1] propose a linear combination of Markov probabilities (Eq. 10) and distance-based probabilities (Eq. 11), $P^t = \omega P_{Markov}^t + (1 - \omega) P_{dist}^t$. Essentially, this approach considers these two factors while computing the probabilities. Furthermore, this can be extended to more number of feature variables in general. The advantage of using a neural network is that it learns the weights of the linear combination itself.

The final layer of the proposed architecture in Figure 1 performs this linear combination. This linear layer outputs unnormalized scores and a softmax operation upon these would result in the probabilities. As the outputs are unnormalized scores, so should be the inputs and that is why we log-transform the Markov probabilities. The distance based probabilities are arrived by considering the softmax of the distances from the source stop s .

We treat the categorical features such as day of week and stops to be served, by passing them to embedding layers before feeding them to the linear layer. The embedding layer computes dense vector representations of the categorical variables. The numerical feature variables such as the number of vehicles are passed directly to the final layer.

We treat the time-lagged solutions between (s, r) by considering linear combinations of L previous solutions of (s, r) . The first linear layer considers past solutions of the look back period as inputs and its output is a score based on that. Thus, the stop which has been chosen more often as the next stop, would be assigned a higher score. We remark that there are other ways to treat time-lagged variables such as LSTM [7], but they might be prone to overfitting because they have a large number of parameters.

■ **Algorithm 1** Transition probability estimation from source stop s .

Input :

- Historical solutions up to day T : $\{(S^t, z^t, \mathbf{X}^t)\}_{t=1}^T$
- s : the source stop

```

1 active days  $\leftarrow$  list of days where  $s$  is active
2 Model.initialize();
3 for  $t$  in active days do
4    $x_{(s)}^t \leftarrow \mathbf{X}^t[s, :]$  // target solution
5    $Past_{(s)}^t \leftarrow \{\mathbf{X}^{t'}[s, :] | t' \in \text{active days}[-(\text{look back period}):] \}$ 
6   if  $\text{length}(\text{active days}) < \text{look back period}$  then
7     Fill the remaining days with equiprobable probabilities vector
8   end
9    $\mathcal{H}^t \leftarrow (z_{(s)}^t, Past_{(s)}^t)$ 
10  for training epochs do
11     $\hat{P}^t[s, :] \leftarrow \text{Model.Predict}(\mathcal{H}^t)$ 
12     $L \leftarrow \text{Cross Entropy}(\hat{P}^t[s, :], x_{(s)}^t)$ 
13    Update Model by backpropagating  $\nabla_{\hat{P}^t}(L)$ 
14  end
15 end
```

The final linear layer outputs a score between s and a destination node. So essentially, there are $|A|$ number of such linear layers. The use of neural network gives us the flexibility to use separate weights for the linear layers of every destination node. Obviously, this may result in overfitting as the number of parameters increase.

Other than the contextual features, which are the same for all (s, r) , the other inputs to the linear layer are specific to (s, r) and so is the output. We use a **separate model for each source node** s and each of them generates transition probabilities from the source node to all the other nodes.

Algorithm. Algorithm 1 proposes a training scheme for estimating probabilities from a single source stop s to the others. We use all the features including the time-lagged solutions day t , to estimate the transition probabilities on day t . This past data ($Past_{(s)}^t$) is obtained by extracting the corresponding row from the incidence matrix. As the training is for s , we only consider the next stop visited after it in the past. Hence, we formulate it as a multiclass classification problem, where the classes are the possible next stops and the objective is to classify them correctly. We use different models for different stops and while training the model for a stop s , we only consider past days where s was served. For any stop that does not have enough past data, to fill the look back period, the **remainder of the look back period are filled with uniform distribution** over the set of possible next stops.

Loss function. We formulate the learning problem as a multiclass classification task. The classification problem is to identify the next stop after s . While training we do not consider any VRP constraints, i.e., the transition probabilities of all the stops can be nonzero regardless of whether they are active or not. Once the neural network predicts the transition probability vector $\mathbf{p}_{(s)}$, we compute the cross entropy loss with respect to the actual solution $\mathbf{x}_{(s)}$.

$$L(\mathbf{p}_{(s)}, \mathbf{x}_{(s)}) = - \sum_{u \in V} x_{su} \log(p_{su}) \quad (12)$$

Algorithm 2 Evaluation of Maximum Likelihood Routing.

Input : $S^{T+1}, z^{T+1}, \{\mathbf{X}^t\}_{t=1}^{T+1}$

```

1 for each stop  $s$  in  $S^{T+1}$  do
2   active days  $\leftarrow$  list of days  $s$  is active
3   Model  $\leftarrow$  Algorithm 1 ( $\{(S^t, z^t, \mathbf{X}^t)\}_{t=1}^T, s$ ) // Model training
4   if  $\text{length}(\text{active days}) \geq \text{look back period}$  then
5      $Past_{(s)}^T \leftarrow \{\mathbf{X}^{T'}[s, :] | T' \in \text{active days}[-(\text{look back period}):]\}$ 
6   end
7   else
8      $Past_{(s)}^T \leftarrow \{\mathbf{X}^{T'}[s, :] | T' \in \text{active days}\}$ 
9   end
10   $\mathcal{H}^{T+1} \leftarrow \left( z_{(s)}^{T+1}, Past_{(s)}^{T+1} \right)$ 
11  Model  $\leftarrow$  Model dictionary [ $s$ ]
12   $\hat{P}^{T+1}[s, :] \leftarrow \text{Model.predict}(\mathcal{H}^{T+1})$ 
13 end
14 MLE Routing ( $-\log(P^{T+1})$ ); Compare with  $X^{T+1}$ 

```

Finally this loss is backpropagated to update the parameters of the neural network. Algorithm 2 shows how we utilize the estimated transition probabilities to come up with the maximum likelihood solutions. Once we train the models for each stop using the data available until day T , we use it for routing on day $T + 1$. To do so, first we estimate the transition probabilities for each stop using the trained models. The (s, r) -th entry of the matrix \hat{P} contains the estimated transition probability of going from stop s to stop r . Then using the estimated transition probability matrix \hat{P} , we solve the maximum likelihood routing problem.

5 Decision Focused Learning

The approaches proposed so far consider the prediction of the transition probabilities and the VRP optimization separately. Such approaches can be viewed as two-stage approaches [4], where a neural network model is separately trained to estimate the unknown coefficients of an optimization problem.

One drawback of such a two-stage approach is the neural network model fails to incorporate information from the optimization problem. As the neural network model is trained without regard for the downstream optimization problem, the loss function fails to consider the impact of the predicted coefficients on the final objective value of the optimization problem.

Decision focused learning approaches [5, 25], on the other hand, consider how effective the predicted values are to solve the optimization problem and is trained with respect to the *optimization task loss* rather than a prediction loss such as cross entropy loss.

In Algorithm 3 we show our implementation of the decision focused learning approach for this problem. We implement the methodology of [20] to differentiate a combinatorial optimization problem with linear objective.

Algorithm 3 Decision Focused Learning Algorithm.

Input : Historic solutions till days T: $\{(s^t, z^t, \mathbf{X}^t)\}_{t=1}^T$

```

1 for training epochs do
2   for  $t$  in 1 to  $T$  do
3     for each stop  $s$  in  $S^{T+1}$  do
4        $x_{(s)}^t \leftarrow \mathbf{X}^t[s, :]$ 
5        $Past_{(s)}^t \leftarrow \{\mathbf{X}^{t'}[s, :] | t' \in \text{active days} \wedge |Past_{(s)}^t| \leq \text{lookback period}\}$ 
6        $\mathcal{H}^t \leftarrow (z_{(s)}^t, Past_{(s)}^t)$ 
7        $\hat{P}^t[s, :] \leftarrow \text{Model}^{(s)}.Predict(\mathcal{H}^t)$ 
8     end
9      $\hat{\pi} \leftarrow -\log(\hat{P}^t)$ 
10     $\hat{X}^t \leftarrow \text{MLE Routing}(\pi)$ 
11     $L \leftarrow \text{sum}(\text{ReLU}(X^t - \hat{X}^t))$ 
12     $\tilde{\pi} \leftarrow \hat{\pi} - \lambda \frac{dL}{d\hat{X}^t}$ 
13     $\tilde{X} \leftarrow \text{MLE Routing}(\tilde{\pi})$ 
14     $\nabla_{\pi}(L) \leftarrow -\frac{1}{\lambda}[\hat{X}^t - \tilde{X}]$ 
15    for each stop  $s$  in  $S^{T+1}$  do
16       $\text{Model}^{(s)}.backpropagate(\nabla_{\pi}(L)[s, :])$ 
17    end
18  end
19 end

```

They consider an optimization problem $\min_{X \in \chi} f(\pi, X)$ with a linear objective. $\hat{X}^*(\hat{\pi})$ is the solution by using predicted $\hat{\pi}$ and the final optimization task loss is $L(X^*(\pi), \hat{X}^*(\hat{\pi}))$. The gradient of this task loss with respect to $\hat{\pi}$ is the following

$$\nabla_{\hat{\pi}} L(X^*(\pi), \hat{X}^*(\hat{\pi})) = -\frac{1}{\lambda}[\hat{X}^*(\hat{\pi}) - \tilde{X}^*(\tilde{\pi})] \quad (13)$$

where $\tilde{\pi}$ is a perturbation around the predicted $\hat{\pi}$, given by

$$\tilde{\pi} = \hat{\pi} + \lambda \frac{dL(X^*(\pi), \hat{X}^*(\hat{\pi}))}{d\hat{X}^*(\hat{\pi})} \quad (14)$$

In our setting, π is the matrix of negative log probability vectors i.e. $\pi = -\log(P)$, and X is the resulted routing. The final task is to minimize the difference between the actual route and the proposed route. So a suitable choice for the *task loss* is to consider arc difference, the number of arcs present in the actual solution but not in the predicted solution. Formally,

$$L(X^*(\pi), \hat{X}^*(\hat{\pi})) = \text{Sum}(\text{ReLU}(X^*(\pi) - \hat{X}^*(\hat{\pi}))) = \sum_{(i,j) \in \text{dim}(X)} \max(x_{ij} - \hat{x}_{ij}, 0) \quad (15)$$

here *Sum* is the summation of all the elements of the matrix. The derivative of L can be computed as follows

$$\frac{dL(X^*(\pi), \hat{X}^*(\hat{\pi}))}{d\hat{X}^*(\hat{\pi})} = \begin{cases} -1 & \text{if } \hat{x}_{ij} < x_{ij} \\ 0 & \text{otherwise} \end{cases} \quad \forall (i, j) \in \text{dim}(\hat{X}^*) \quad (16)$$

If we consider a squared loss instead of the ReLU, we replace $\max(x_{ij} - \hat{x}_{ij}, 0)$ with $(x_{ij} - \hat{x}_{ij})^2$. In this case the derivative would be

$$\frac{dL(X^*(\pi), \hat{X}^*(\hat{\pi}))}{d\hat{X}^*(\hat{\pi})} = \begin{cases} -2 & \text{if } \hat{x}_{ij} < x_{ij} \\ 2 & \text{if } \hat{x}_{ij} > x_{ij} \\ 0 & \text{otherwise} \end{cases} \quad \forall (i, j) \in \dim(\hat{X}^*) \quad (17)$$

Intuitively, if $i \rightarrow j$ is present in X , but not in \hat{X} , then we lower π_{ij} by λ and with this generate a new solution with. A scaled difference between these two solutions is the gradient with respect to π .

6 Experimental Evaluation

6.1 Evaluation Criteria

We are interested in how the MLE routing solutions differ from the used routes. To do so, we evaluate the performance using the following two evaluation measures.

Arc Difference (AD). measures the number of arcs traveled in the actual solution but not in the MLE routing solution. It is calculated by taking the set difference of the arc sets of the test and predicted solutions. The percentage is computed by dividing AD by the total number of arcs in the whole routing.

Route Difference (RD). indicates the number of stops that were incorrectly assigned to a different route. Intuitively, RD may be interpreted as an estimate of how many moves between routes are necessary when modifying the predicted MLE solution to match the actual routing. To compute RD, the pair of routes with the smallest difference in stops is greedily selected without replacement. The total number of incorrectly assigned stops is considered as RD. The percentage is computed by dividing RD by the total number of stops in the whole routing.

We also present the cross entropy (CE) loss as the neural network models are trained with respect to this criterion.

6.2 Data Description

For empirical evaluation¹, we use actual historical data from a logistics company to compare the performance of our proposed approaches against the Markov model presented in [1]. The data consists of 201 daily routings collected in a span of 39 weeks. It has 73 unique customers, each representing a node other than the depot. In each instance, an average of 31 stops are serviced by an average of 8 vehicles. We group the instances by day of the week, giving us an average of 29 instances per weekday. In training and testing the models, we used a 75%-25% split while ensuring that we have exactly 7 testing instances per weekday. We use a rolling window model for valuation, where the lookback period remains fixed and counts backwards from the most recent observation.

In Table 3, we present the percentage AD and RD of the Markov approaches on the test instances for each day of week. The Markov (allday) approach arrives on the transition

¹ The code and the anonymised data are available at <https://github.com/JayMan91/CP2021-Data-Driven-VRP>.

probability by considering all past days. On the contrary, the Markov (weekday) approach considers only those past instances which occurred on the same day of week as the evaluation instances. Both the approaches use Eq. (10) to compute the probabilities. We can see in Table 3 that Markov (allday) performs better on the weekdays, but its result worsens on the weekends. Due to the operational characteristics of the company, the number of customers, number of available vehicles, and hence the routing decisions tend to be highly dependent on the day-of-week. So there is a strong influence of the day on the probabilities. Probably, this is why [1] preferred the Markov (weekday) approach. The motivation behind the neural network approach is that it can consider the day of week as feature, so that we do not have to compute the probability separately for each day. Moreover, other feature variables can easily be passed into the neural network.

In our experiment we consider the following feature variables to predict the transition probabilities— a. day of week, b. the set of stops to be served, c. distance between the stops, d. number of available vehicles, e. routings used in the past, f. transition probabilities computed by Markov (weekday).

6.3 Experimental Results

In this section, we will address the following research questions

- Choice of the feature variables and the network architecture in a systematic way
- Compare the quality of predictions of the neural network trained with respect to the CE loss with that of Markov counting approach
- The effectiveness of a decision focused approach, which trains the network to directly minimize AD

6.3.1 Choice of Network Architecture and Feature Variables

As mentioned in Section 4, there are many choices for the network architecture and because of limited data we are careful to avoid overfitting.² In Table 1, we first present the impact of feature variables on the quality of predictions. We show the cross entropy loss on both training and test data and AD and RD on test data. We point out that the network presented in Figure 1 results in the lowest training loss among them. On the other hand, a network only with Markov probabilities as input lowers test loss and lower AD, RD. A network without the time-lagged data has even lower CE loss on the test instances and lowest AD and RD suggesting that past information is already subsumed in the Markov probabilities, making the time-lagged information redundant. The Markov probabilities along with the contextual information seem to be the right choice for the feature variables.

In the lower section of Table 1, we present two alternative architecture choices. The first one replaces the linear layer of the lagged solutions with an LSTM. The second one has different weights for different destination stops in the final layer in Figure 1.

It shows that using different weights for different destination stops results in lowest training CE loss. But this model clearly overfits, as the performance is poor on the test data. The LSTM model seems to improve on CE loss but not on AD and RD measures. So overall, the model **without the past data** results in lowest CE loss as well as lowest AD and RD. This model has the Markov probabilities as input, and the past information carried by this probability.

² We use Pytorch [18] and Gurobi [9] for neural network and VRP models respectively.

■ **Table 1** Investigation into feature variables and architectures.

Model	Training CE	Test CE	AD	AD (%)	RD	RD(%)
Experiment on feature variables						
Neural Net	2.14	1.10	6.27	19.80	4.57	18.04
Neural Net (without past data)	2.48	1.04	5.68	18.04	4.30	17.02
Neural Net (without weekday)	2.14	1.09	6.24	19.75	4.59	18.03
Neural Net (without stop information)	2.20	1.13	6.28	19.86	4.56	17.97
Neural Net (without distance)	2.20	1.10	6.18	19.54	4.46	17.62
Neural Net (without Markov probabilities)	2.43	1.49	7.99	26.26	5.32	21.38
Neural Net (only Markov probabilities)	2.58	1.07	5.95	18.85	4.29	16.93
Experiment on architecture choice						
LSTM	2.22	1.01	6.35	20.10	4.49	17.75
Linear Layer different for each stop	1.37	1.82	7.21	22.81	4.74	18.57

6.3.2 Neural Network Predictions

■ **Table 2** Comparison of Neural Network with Markov Counting (Actual Distance is 413 km.)

	CE loss	Arc Difference (AD)		Route Difference (RD)		Distance (km.)
		Absolute	Percent	Absolute	Percent	
Markov (allday)	2.77	10.33	35.69	6.29	25.75	424
Markov (weekday)	2.44	5.86	18.55	4.39	17.26	418
Neural Net	1.04	5.68	18.04	4.30	17.02	414
Conventional VRP	11.90	21.47	73.14	11.65	46.93	366

The last section suggests to consider a network without the lagged variables for this task. Next, we compare the quality of predictions of this model shown to that of Markov counting approach. We present the average of CE loss, AD, RD between the actual solutions and generated solution on test instances in Table 2. We also present the distance of the solutions of these approaches. We point out in Table 2 that a neural network model results in lower CE loss, which is expected as the model is trained with that objective. Moreover, we also observe lower AD and RD with this model. We also present results of a conventional VRP algorithm, which is the best in terms of total distance covered, but clearly very far off from the preferred solution. Table 3 presents this comparison in more detail, where we evaluate for each day of week separately. Although we do not need to train the neural network separately

■ **Table 3** Daywise Analysis of Arc Difference and Route Difference.

	Arc Difference(%)			Route Difference(%)		
	Markov (allday)	Markov (weekday)	Neural Net	Markov (allday)	Markov (weekday)	Neural Net
Monday	51.53	23.62	23.62	27.98	21.75	21.25
Tuesday	24.96	25.61	25.82	29.15	28.87	30.85
Wednesday	19.61	21.30	20.48	17.96	15.12	14.61
Thursday	24.89	22.86	21.17	19.75	18.63	17.08
Friday	19.18	19.38	18.08	13.74	13.19	11.54
Saturday	51.59	0.00	0.00	25.21	0.00	0.00
Sunday	58.09	17.08	17.11	46.48	23.23	23.82
Overall	35.69	18.55	18.04	25.75	17.26	17.02

for each day of week, by considering the Markov probabilities as inputs, it is able to generate predictions which result in lower AD and RD. This demonstrates the advantage of the neural network approach, which can consider multiple inputs, contextual as well as temporal, in a single framework.

Figures 2 to 6 illustrate our approach for one instance. Figures 3 and 4 present the learned transition probabilities and Figures 5 and 6 show the MLE routing of Markov weekday and neural net respectively.

6.3.3 Decision Focused Learning

Next, we experiment with the decision focused learning approach introduced in section 5. We use the same neural network architecture but trained with arc difference as the loss function, and the loss backpropagated through a corresponding subgradient (Eq. (16)). We present the

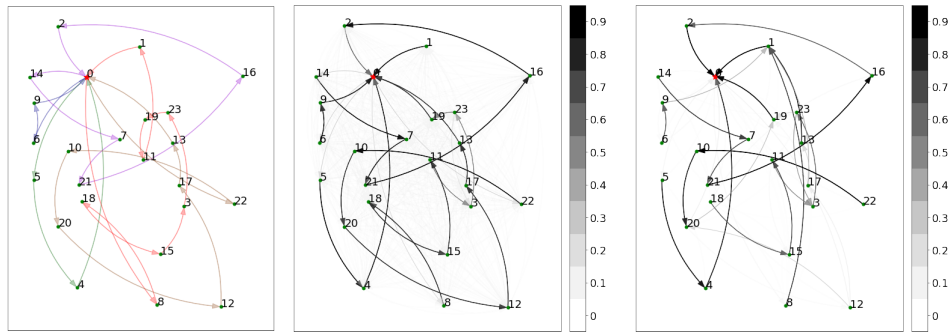
■ **Table 4** AD and RD with Decision Focused Learning.

	CE loss (test)	Arc Difference (AD)		Route Difference (RD)		Distance (km.)
		Absolute	Percent	Absolute	Percent	
Relu loss	2.77	13.76	45.96	9.51	38.54	434
Squared loss	3.97	13.31	44.27	9.10	37.14	436

solution quality of this approach in Table 4. We can see, it fails to generate lower AD and RD on the test instances. In fact, we observe AD reducing on training instances but not on test instances, suggesting a case of overfitting. Only 152 instances is not enough to train a complex model like this. So the poor quality can be attributed to limited amount of data.

7 Conclusion

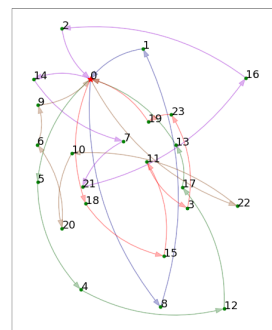
We presented a neural network model which learns the transition probabilities between stops in a CVRP setting. With these transition probabilities, we solve the MLE routing problem instead of the conventional VRP. The resulting solution is able to mimic the solution preferred by the route planners and drivers. In this way, we are able to include the preferences of the planners and the drivers implicitly in the VRP solution.



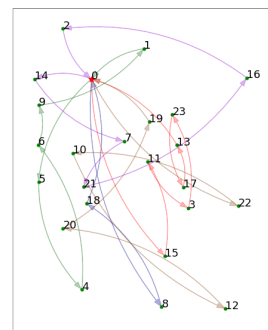
■ **Figure 2** Human-made solution

■ **Figure 3** Learned probabilities (Markov)

■ **Figure 4** Learned probabilities (NN)



■ **Figure 5** Markov solution.



■ **Figure 6** NN solution.

We extend on the work of [1]. The novelty of our approach is to use a neural network model to estimate the probabilities. Key developments are the use of an arc-based architecture to control the number of trainable parameters, and the identification of the standard cross-entropy classification loss as a suitable (and cheap to compute) proxy loss for training. This leads us to develop a general framework for such problem setting, which has the flexibility of taking contextual features including the output of [1] into consideration. By considering the contextual features in a principled way, our approach marginally outperforms [1], emphasising the advantage of a generic approach.

We also use a decision focused learning approach which directly trains the neural network to minimize the final objective of minimizing the difference between the generated solution and the preferred solution. Although this approach considers the structure of the VRP optimization problem, our results show that it fails to generate good quality solutions in the test data. We believe it is due to the limited number of training instances.

Our methodology relies on the presence of recurrent stops in our training set. Future research will aim to extend our methodology to learn preferences over non-recurring stops. It will also be interesting to investigate loss functions that include the structure of the CVRP tackling the challenges of the scalability.

References

- 1 Rocsildes Canoy and Tias Guns. Vehicle routing by learning from historical solutions. In *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, volume 11802, pages 54–70. Springer, 2019.

- 2 Vaida Ceikute and Christian S Jensen. Routing service quality–local driver behavior versus routing services. In *2013 IEEE 14th International Conference on Mobile Data Management*, volume 1, pages 97–106. IEEE, 2013.
- 3 George B Dantzig and John H Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.
- 4 Emir Demirovic, Peter J. Stuckey, James Bailey, Jeffrey Chan, Chris Leckie, Kotagiri Ramamohanarao, and Tias Guns. An investigation into prediction + optimisation for the knapsack problem. In Louis-Martin Rousseau and Kostas Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, volume 11494 of *Lecture Notes in Computer Science*, pages 241–257. Springer, 2019.
- 5 Adam N Elmachtoub and Paul Grigas. Smart “predict, then optimize”. *Management Science*, 2021.
- 6 Stefan Funke, Sören Laue, and Sabine Storandt. Deducing individual driving preferences for user-aware navigation. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–9, 2016.
- 7 Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *IEEE Trans. Neural Networks Learn. Syst.*, 28(10):2222–2232, 2017.
- 8 Chenjuan Guo, Bin Yang, Jilin Hu, Christian S Jensen, and Lu Chen. Context-aware, preference-based vehicle routing. *The VLDB Journal*, pages 1–22, 2020.
- 9 LLC Gurobi Optimization. Gurobi optimizer reference manual, 2020. URL: <http://www.gurobi.com>.
- 10 Xiangpei Hu, Zheng Wang, Minfang Huang, and Amy Z. Zeng. A computer-enabled solution procedure for food wholesalers’ distribution decision in cities with a circular transportation infrastructure. *Computers & Operations Research*, 36(7):2201–2209, 2009.
- 11 Nicolas Jozefowicz, Frédéric Semet, and El-Ghazali Talbi. Multi-objective vehicle routing problems. *European journal of operational research*, 189(2):293–309, 2008.
- 12 Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. [arXiv:1412.6980](https://arxiv.org/abs/1412.6980).
- 13 Christophe Lecluyse, Tom Van Woensel, and Herbert Peremans. Vehicle routing with stochastic time-dependent travel times. *4OR*, 7(4):363–377, 2009.
- 14 Tzong-Ru Lee and Ji-Hwa Ueng. A study of vehicle routing problems with load-balancing. *International Journal of Physical Distribution & Logistics Management*, 1999.
- 15 Julia Letchner, John Krumm, and Eric Horvitz. Trip router with individualized preferences (trip): incorporating personalization into route planning. In *IAAI’06 Proceedings of the 18th conference on Innovative applications of artificial intelligence - Volume 2*, pages 1795–1800, 2006.
- 16 Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.
- 17 Chun Kai Ling, Fei Fang, and J. Zico Kolter. What game are we playing? end-to-end learning in normal and extensive form games. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 396–402. ijcai.org, 2018.
- 18 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

- 19 Yong Peng and Xiaofeng Wang. Research on a vehicle routing schedule to reduce fuel consumption. In *2009 International Conference on Measuring Technology and Mechatronics Automation*, volume 3, pages 825–827, 2009.
- 20 Marin Vlastelica Pogancic, Anselm Paulus, Vít Musil, Georg Martius, and Michal Rolinek. Differentiation of blackbox combinatorial solvers. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.
- 21 J. David Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 93–100, 1985.
- 22 Tomer Toledo, Yichen Sun, Katherine Rosa, Moshe Ben-Akiva, Kate Flanagan, Ricardo Sanchez, and Erika Spissu. Decision-making process and factors affecting truck routing. In *Freight Transport Modelling*. Emerald Group Publishing Limited, 2013.
- 23 P Toth and D Vigo. The family of vehicle routing problem. *Vehicle Routing: Problems, Methods, and Applications*, pages 1–23, 2014.
- 24 Po-Wei Wang, Priya L. Donti, Bryan Wilder, and J. Zico Kolter. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6545–6554, 2019.
- 25 Bryan Wilder, Bistra Dilkina, and Milind Tambe. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1658–1665, 2019.
- 26 Yiyong Xiao, Qihong Zhao, Ikou Kaku, and Yuchun Xu. Development of a fuel consumption optimization model for the capacitated vehicle routing problem. *Computers & Operations Research*, 39(7):1419–1431, 2012.


A Experiment Setup

We use Pytorch [18] and Gurobi [9] for neural network and VRP models respectively. We use Adam optimizer [12] implementation of Pytorch. The hyperparameters of each setup is detailed below.


■ **Table 5** Hyperparameters Configuration (For all experiments the embedding dimension of weekday and stop feature are 6 and 40 respectively).

	Learning rate	Epochs
Neural Net	0.1	50
Neural Net (without past data)	0.1	100
Neural Net (without weekday)	0.1	50
Neural Net (without stop information)	0.1	100
Neural Net (without distance)	0.1	100
Neural Net (without Markov probabilities)	0.1	100
Neural Net (only Markov probabilities)	0.1	100
LSTM	0.1	50
Linear Layer different for stops	0.01	100


Statistical Comparison of Algorithm Performance Through Instance Selection

Théo Matricon ✉ 

Univ. Bordeaux, CNRS, LaBRI, UMR 5800, F-33400, Talence, France

Marie Anastacio ✉ 

Leiden Institute of Advanced Computer Science, Leiden, The Netherlands


Nathanaël Fijalkow ✉ 

CNRS, LaBRI, Bordeaux, France,

The Alan Turing Institute of data science, London, UK

Laurent Simon ✉ 

Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400, Talence, France

Holger H. Hoos ✉ 

Leiden Institute of Advanced Computer Science, Leiden, The Netherlands

University of British Columbia, Vancouver, Canada

Abstract

Empirical performance evaluations, in competitions and scientific publications, play a major role in improving the state of the art in solving many automated reasoning problems, including SAT, CSP and Bayesian network structure learning (BNSL). To empirically demonstrate the merit of a new solver usually requires extensive experiments, with computational costs of CPU years. This not only makes it difficult for researchers with limited access to computational resources to test their ideas and publish their work, but also consumes large amounts of energy. We propose an approach for comparing the performance of two algorithms: by performing runs on carefully chosen instances, we obtain a probabilistic statement on which algorithm performs best, trading off between the computational cost of running algorithms and the confidence in the result. We describe a set of methods for this purpose and evaluate their efficacy on diverse datasets from SAT, CSP and BNSL. On all these datasets, most of our approaches were able to choose the correct algorithm with about 95% accuracy, while using less than a third of the CPU time required for a full comparison; the best methods reach this level of accuracy within less than 15% of the CPU time for a full comparison.

2012 ACM Subject Classification General and reference → Evaluation; Theory of computation → Automated reasoning; Theory of computation → Constraint and logic programming

Keywords and phrases Performance assessment, early stopping, automated reasoning solvers

Digital Object Identifier 10.4230/LIPIcs.CP.2021.43

Supplementary Material The source code can be found at:

Software (Source Code): <https://github.com/Theomat/PSEAS>

archived at `swb:1:dir:0cec861beb644ac8df9cd2f195457cbaabd16773`

1 Introduction

The amount of computational resources required to assess empirically whether a new automated reasoning algorithm exceeds state-of-the-art performance is growing as our ability to run experiments on challenging benchmark instances expands. From the evaluation of early algorithms against the human ability to solve given instances by hand [7] to extensive competitions requiring CPU years to determine a winner [10, 24, 30], the demands for computational power have grown along with the ability of state-of-the-art solvers to tackle larger instances. Moreover, each published idea is often the result of a number of unsuccessful



© Théo Matricon, Marie Anastacio, Nathanaël Fijalkow, Laurent Simon, and Holger H. Hoos; licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 43; pp. 43:1–43:21

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

attempts, which developers either evaluated on a small set of instances, without a principled way of knowing how representative this evaluation has been, or in a more extensive way, implying days of CPU times.

This growth comes at a cost. By requiring large amounts of computational resources for compelling performance comparisons, the community restricts the ability of researchers with limited access to such resources to evaluate their ideas and publish their work. The energy consumption of such computations is also an increasing concern. Our work addresses this issue by proposing principled statistical methods to decide earlier when to stop running a less promising solver.

We introduce the per-set efficient algorithm selection problem (PSEAS): Given two algorithms, an incumbent A_{inc} and a challenger A_{ch} , and a set of problem instances \mathcal{I} , how can we minimise the computational resources (here: CPU time) required to determine, at a required level of confidence, whether A_{ch} performs better than A_{inc} on \mathcal{I} ?

We are not aware of any prior work on this fundamental problem, but methods for addressing variants of it are used in several contexts. This includes, for example, general-purpose algorithm configurators, which compare the performance of several configurations of a single algorithm. While most configurators, such as SMAC [14] or ParamILS [15], simply look at the difference between their objective functions, the racing-based configurator *irace* [20], inspired by prior racing procedures from machine learning [22], addresses the problem of statistical confidence using a statistical test. However, all of them sample the instances uniformly at random. Other related work comes from the area of per-instance algorithm selection, for which Gent et al. [8] introduced a discrimination measure that we adapted to our context.

We describe five methods for selecting on which instances to run the competing algorithms, and three methods for deciding when to stop the evaluation. We compare the 15 resulting approaches on four benchmarks for classic computational problems: the propositional satisfiability problem (SAT), the constraint satisfaction problem (CSP) and Bayesian network structure learning (BNSL). On these datasets, our approaches can determine the better-performing algorithm with up to 98% accuracy, while using less than a third of the CPU time required for a full comparison, and the best methods achieve this level of accuracy within less than 15% of the CPU time for an exhaustive comparison.

The remainder of this paper is organised as follows: In Section 2, we introduce the PSEAS problem and present related work, and in Section 3, we describe methods for solving it. Section 4 presents implementation details and the datasets used in our experiments. Section 5 explains some of our design choices with experimental results, and the results from our main series of experiments are reported in Section 6. Finally, in Section 7, we draw some general conclusions and discuss future work.

2 Background and related work

The PSEAS problem formalises the following question: How to compare a new solver against the state of the art with as little computational power as possible? More specifically, how to select on which instances to run the new algorithm to do this comparison and on which criterion can this comparison be stopped if one algorithm performs significantly better than the other? The PSEAS problem or small variations of it also appears in competitions, where it could be used to disqualify low performing algorithms, or in algorithm configurators to abandon less promising configurations faster. For simplicity, we consider an algorithm as a method or solver with fixed parameter values and the state of the art as a single algorithm.

To answer these questions, we suppose that we have prior knowledge about the performance of the state-of-the-art algorithm, features describing the problem instances and performance samples or performance distributions on those instances.

Definition of the per-set efficient algorithm selection problem (PSEAS)

We let \mathcal{I} denote the set of instances, $T_{cut} \in \mathbb{R}^+$ the cutoff threshold, m the performance metric that evaluates an algorithm on an instance, and c the cost function that evaluates the cost of running an algorithm on an instance. We consider two algorithms: the incumbent A_{inc} and the challenger A_{ch} , and assume that the cost $c(A_{inc}, I)$ and performance $m(A_{inc}, I)$ of running A_{inc} on an instance I is known for all instances, whereas these quantities are unknown on all instances for A_{ch} . This assumption is consistent with the fact that A_{inc} represents the state of the art, hence can be assumed to have been evaluated on many problems. The problem is to determine which of the two algorithms performs best according to $\sum_{I \in \mathcal{I}} m(A_{ch}, I)$ while running A_{ch} only on a subset $\mathcal{I}_{run} \subset \mathcal{I}$ that minimises the cost $\sum_{I \in \mathcal{I}_{run}} c(A_{ch}, I)$.

Scope of this work

We pose $\mathcal{A}_+ = \mathcal{A} \cup \{A_{ch}\}$ where \mathcal{A} is a set of algorithms containing A_{inc} and providing background knowledge. Unless stated otherwise, we write I for an instance in \mathcal{I} and A for an algorithm in \mathcal{A} . For simplicity we consider the algorithms to be deterministic, hence for an algorithm $A \in \mathcal{A}_+$, we define the running time as $rt(A, I) \in [0, T_{cut}]$ for an instance I . We define $m(A, I) = c(A, I) = rt(A, I)$: the running time of an algorithm is considered as a proxy for the energy cost of running it.

The performance of an algorithm is the sum on all instances of the running times bounded by a fixed cutoff time. It is consistent with the typical performance metric used in programming competitions, the Penalised Average Running time (PAR), which penalises algorithms that do not solve an instance before the cutoff time by assigning them the score of α times the cutoff time, for a constant α .

Some methods we describe rely on background knowledge about the set of instances. The required knowledge varies from one to another but is similar to the one used in the algorithm selection problem and thus readily accessible. We consider the following ways of specifying the background knowledge:

- *Sample-based*: for each instance I and algorithm A we have the running time $rt(A, I)$ of A on I .
- *Feature-based*: for each instance I we have a feature vector f_I .
- *Statistics-based*: for each instance I we have a prior in the form of a probability distribution δ_I over $[0, T_{cut}]$, expressing that $\delta_I(t)$ is the probability that A_{ch} solves the instance I at time t . In practice, we obtain this prior by fitting a distribution to the running times of A .

Note that above, $A \neq A_{ch}$, the background knowledge that is based on other algorithms. The implicit assumption is that running times of algorithms from \mathcal{A} and feature vectors of the instances are both predictive of the running times of A_{ch} : for instance, if all algorithms in \mathcal{A} solve an instance I very quickly, then so should A_{ch} . In other words A_{ch} is expected to have similar behaviour as the algorithms in \mathcal{A} . Similarly, if two feature vectors f_I and $f_{I'}$ are close for two instances I, I' , then their running times should be close. These assumptions are prominently made in running-time prediction such as in Hutter et al. [16] and per-instance algorithm configuration (see e.g. Kerschke et al. [17]). In other words, the key insights and mathematical formalisation of our work are using the background knowledge described above to evaluate the expected performance of A_{ch} .

Related work

To the best of our knowledge, the problem we address has not been studied as such in the literature. However, similar questions appear in other settings.

In the early SAT Competitions (see e.g. the 2002 competition [27]), the competition consisted of two stages: first they ran all the solvers on a subset of instances to extract the top solvers, then the latter were run on all instances. The selection was done by experts: we propose to address this problem in an automated manner on a solver pair basis.

In the context of instance generation for CP problems, Gent et al. [8] propose a way to define how discriminating an instance is in order to generate instances for model selection based on samples of running times. This method does not answer our aim to reduce the running time but could lead to choose relevant instances. Thus, we included it in our experiments with a minor change to account for running time minimisation.

When we decide on which instance to run our algorithm, a score is assigned to each instance (see Section 3) which relates to the fitness functions used in evolutionary algorithms. At the time of writing, we found no published method that could be easily applied to our problem.¹

For algorithm configuration (see e.g. Hoos [11]), which tries to find a set of parameter that optimises the performances of a configurable algorithm, comparing the performance of two configurations is a key element. SMAC and ROAR [14], as well as *irace* [20], pick uniformly at random the instances on which they run it, without considering prior knowledge they gathered. Racing procedures like *irace* are based on prior work from Maron and Moore [22] which aimed at comparing many machine learning models on a subset of test points to estimate their accuracy with a certain statistical confidence. In this line of work, *irace* requires evidence in the form of a statistical test to decide when to stop running a less promising configuration. SMAC and ROAR on the other hand compare the raw performance metric. We included the statistical test from *irace* in our experiments.

Our problem is also related to the per-instance algorithm selection problem (see e.g. Kerschke et al. [17]) in which one tries to know on which algorithm a specific instance should be run to be solved with the best possible performance. There are key differences that prevent us from using selection algorithms; typically their problem comes with prior knowledge in the form of instances features, that we do not always assume to have, and running time of the algorithms on other instances, which we do not have available for the new algorithm. Also, our main goal is to reduce the time needed to evaluate which one is the best.

Finally, there is a significant link with problems tackled by active learning methods [29], in particular the pool-based selective sampling problem, that tries to decide which instance among a set of unlabeled instances should be evaluated next. Those methods are aimed at a machine learning model and the choice of an instance is based on the impact it may have on the model (e.g. reducing its variance or expected error). In this work we limit our investigation to model-free methods.

3 Instance Selection and Discrimination Methods for PSEAS

Our goal is to define a *strategy* that sequentially chooses the instances on which to run A_{ch} and decides if the evidence so far gives sufficient confidence to stop the comparative evaluation. Algorithm 1 formalises this iterative process using a score-based approach: each

¹ We cover the recently published work of Bossek & Wagner [5] in Appendix B.

instance is assigned a score, which may be updated along the comparison to – intuitively – reflect the interest in running this instance. There are two main components in this algorithm: one for score computation (lines 2, 4, 7, see Section 3.1) and one for confidence (lines 1, 3, 6, see Section 3.2). The score enables to choose the best instance to run whereas the confidence tells when to stop the comparison. These two components will be explained in more details later.

■ **Algorithm 1** Determine which of A_{inc} , A_{ch} performs best on \mathcal{I} with a confidence threshold of C_{thres} ; $C_{current}$ is the current confidence and depends on A_{inc} , \mathcal{I}_{torun} are the instances on which A_{ch} has not been run.

```

1: set  $\mathcal{I}_{torun} = \mathcal{I}$  and  $C_{current} = 0$ 
2: compute  $score(I)$  for all  $I \in \mathcal{I}$ 
3: while  $C_{current} < C_{thres}$  do
4:   pick  $I^* \in \arg\max_{I \in \mathcal{I}_{torun}} score(I)$  and remove  $I^*$  from  $\mathcal{I}_{torun}$ 
5:   evaluate  $rt(A_{ch}, I^*)$ 
6:   update  $C_{current}$ 
7:   update  $score(I)$  for  $I \in \mathcal{I}_{torun}$ 
8: end while
9: return best performing algorithm from  $(A_{inc}, A_{ch})$ 

```

Strategy evaluation

We consider two metrics for evaluating strategies: the *cost* and the *accuracy*.

We measure the computational effort (which we want to minimise) as the ratio of the total running time for instances in \mathcal{I}_{run} , the set of instances on which A_{ch} has been run by the strategy, over the total running time over all instances; this results in a number between 0 and 1. Note that the goal is not to minimise the *number* of instances A_{ch} is run on, but rather the total running time of A_{ch} on these instances. To evaluate our strategy, we determine this cost over many ordered pairs of algorithms (A_{inc}, A_{ch}) and consider the median. Formally, for a set of ordered pairs \mathcal{P} :

$$cost(\mathcal{P}) = median \left[\left(\frac{\sum_{I \in \mathcal{I} \setminus \mathcal{I}_{torun}} rt(A_{ch}, I)}{\sum_{I \in \mathcal{I}} rt(A_{ch}, I)} \right)_{(A_{inc}, A_{ch}) \in \mathcal{P}} \right],$$

where \mathcal{I}_{torun} are the instances that have not been run by the strategy during its execution, as defined in Algorithm 1. We note that $cost(\mathcal{P})$ only depends on A_{ch} , since A_{inc} is assumed to have already been run.

We measure the *accuracy* of a strategy (which we want to maximise), as the ratio of correct guesses made by the strategy when deciding which algorithm from an ordered pair of algorithms (A_{inc}, A_{ch}) performs best. Formally, for a set of ordered pairs \mathcal{P} :

$$accuracy(\mathcal{P}) = \frac{\sum_{(A_{inc}, A_{ch}) \in \mathcal{P}} 1_{\{\hat{A}_{best} = A_{best}\}}}{|\mathcal{P}|},$$

where A_{best} is the true best performing algorithm in (A_{ch}, A_{inc}) , and \hat{A}_{best} is the best algorithm given by the strategy. Our definition of *accuracy* uses the mean, since the median over the results of the indicator function would produce too limited a range of results to be useful for comparing strategies.

We note that the choice made in line 4 of Algorithm 1 aims at balancing the effects of two contradicting goals. The *instance selection component* tries to minimise the computational effort by deciding on which instances to run A_{ch} , based on a score given to each instance. The *discrimination component* decides, based on the data gathered so far, whether the expected accuracy, or confidence, is high enough to stop the comparison.

3.1 The instance selection component

With the aim of minimising the overall computational effort, our algorithm iteratively chooses the most relevant instance, according to a score (lines 2 and 7 in Algorithm 1). Instances with the highest score are expected to be the most relevant ones (i.e. intuitively giving the most information at the lowest cost).

Baseline: Uniform random sampling

As a baseline, we use a random sampling approach. In our algorithm, this corresponds to giving the same score to all instances, and thus to a uniform random choice at each iteration.

The discrimination-based selection method

This sample-based method is inspired by Gent et al. [8]; they developed it as a way to find optimal parameters of instances in an instance selection method for automated constraint model selection. The intuition is to choose the most discriminating instances first. Let $\rho > 1$ be a constant; an algorithm A is ρ -dominated on an instance I if there exists another algorithm A' such that $rt(A', I) \leq \rho \cdot rt(A, I)$. The *discrimination quality* of an instance I , denoted $G(I)$, is the fraction of algorithms that are ρ -dominated on this instance. Using this measure as-is would not take into account our goal of minimising the running time, so we divide the discrimination quality by the mean running time of the instance. The obtained score only needs to be computed once:

$$score(I) = \frac{G(I)}{\text{mean}[rt(A, I)]_{A \in \mathcal{A}}}.$$

The variance-based selection method

This statistics-based method uses the intuition that the most interesting instances are the ones most likely to have very different running times for A_{inc} and A_{ch} . For each instance I we have a prior δ_I , which is the running time distribution of A_{ch} . We want to choose an instance with the highest variance $\text{argmax}_{I \in \mathcal{I}_{torun}} V(\delta_I)$. As for the discrimination-based selection method, since we want to minimise the running time we divide by the mean running time of the instance. The obtained score only needs to be computed once:

$$score(I) = \frac{V(\delta_I)}{\mathbb{E}[\delta_I]}.$$

The information-based selection method

This statistics-based method is based on a similar intuition as the previous method. We are interested in instances from which we gain as much information as possible; the variance is only one (natural) indicator of this information. Following this approach, we can also estimate

the information gained from a specific instance. The concrete information we are after² is given by the discrete random variable stating that A_{ch} is better than A_{inc} , formally defined as $sign(\Delta_{tot})$. Let Δ_I be the random variable defined as $\Delta_I := rt(A_{ch}, I) - rt(A_{inc}, I)$; we compute the expected information brought by Δ_I ; hence the information gain is defined as follows for $I \in \mathcal{I}$:

$$\begin{aligned} IG_I(sign(\Delta_{tot})) &:= \mathbb{E}_{e_I \sim \Delta_I} [D_{KL}(P_{+i} || P)] \text{ with} \\ P &= sign(\Delta_{tot}) | \forall J \in \mathcal{I}_{run}, \Delta_J = e_J \\ P_{+i} &= sign(\Delta_{tot}) | \forall J \in \mathcal{I}_{run}, \Delta_J = e_J, \Delta_I = e_I \end{aligned}$$

where D_{KL} is the Kullback–Leibler divergence, with the e_J being realizations of the Δ_J since the difference for the instances in \mathcal{I}_{run} is known.

As for the previous method, to balance information and running time, we divide by the expected running time, and therefore use the following score function, which we update at each iteration:

$$score(I) = \frac{IG_I(sign(\Delta_{tot}))}{\mathbb{E}[\delta_I]}.$$

The feature-based selection method

In this feature-based and statistics-based method, we assume that for each instance I , we have a feature vector $f_I \in \mathbb{R}^n$ in some dimension n . The implicit assumption is that features are predictive of the running times of A_{ch} . We proceed in two steps:

- Constructing a distance metric $d : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}_{\geq 0}$, such that if $d(f, f')$ is small, then two instances with features f and f' have similar running times.
- Assigning a score to each instance $I \in \mathcal{I}_{torun}$.

Constructing a distance metric. The objective is to define a distance predictive of the running times; to this end, we introduce a weight for instance features, represented by a weight vector $\theta \in \mathbb{R}^n$. Let us consider distances of the form:

$$d_\theta(f_I, f_J) = \sqrt{\sum_{x=1}^n (\theta(x) \cdot (f_I(x) - f_J(x)))^2}.$$

Intuitively, for a feature x , the parameter $\theta(x)$ determines the importance of x in predicting the running times. Let us write m_I for the median time over all algorithms on instance I . We optimise over θ by considering:

$$\theta^* \in \operatorname{argmin}_{\theta \in \mathbb{R}^n} \sum_{I, J \in \mathcal{I}} (d_\theta(f_I, f_J)^2 - |m_I - m_J|)^2;$$

i.e., d_{θ^*} is the best distance in this family for predicting differences in median running time. The parameter vector θ^* is the solution of a non-negative ordinary least square optimisation problem and can therefore be computed efficiently [18]. Note that the space complexity is quadratic in the number of instances and linear in the feature space dimension.

² See Section 3.2 for one possible expression of this concrete information: the variable Δ_{tot} .

Assigning a score. Given a distance metric d , we now define a score for a given problem instance. Here, it is convenient to minimise rather than maximise the following quantity with respect to d :

$$S(I) = \sum_{J \notin \mathcal{I}_{torun}} \frac{rt(A_{ch}, J)}{d(f_I, f_J)} + \sum_{J \in \mathcal{I}_{torun}} \frac{\mathbb{E}[\delta_J]}{d(f_I, f_J)} \quad \text{and} \quad score(I) = \frac{1}{S(I)}.$$

The score is updated at each iteration. In all previous methods, the score of an instance I only uses the information on I ; the strength of this method is to gather and weight information over all instances. Indeed, the score of I is a weighted average over all running time predictions, meaning $\mathbb{E}[\delta_J]$ when $J \in \mathcal{I}_{torun}$ and $rt(A_{ch}, J)$ otherwise, and the prediction for J contributes to the prediction of I up to the multiplicative factor $\frac{1}{d(f_I, f_J)}$.

3.2 The discrimination component

The discrimination component aims at estimating the accuracy of the current decision of which among A_{inc} and A_{ch} performs best. However, this measure can never be accessed, since the complete data is not available. Hence we introduce the expected accuracy, or *confidence*, as a proxy for accuracy. We note that this is not an expectation in the statistical sense. The confidence has a different meaning and is computed differently for each discrimination method, as explained later. It provides a measure of the current state of the strategy. When the confidence reaches a threshold C_{thres} (line 3 of Algorithm 1), the strategy stops and returns the algorithm evaluated as being the best.

Baseline: Subset method

As a baseline, we use a fixed-size subset of instances: we fix $\gamma \in [0; 1]$ and decide to stop when A_{ch} has been run on $\lfloor \gamma |Z| \rfloor$ instances³. The confidence for this method is 0 until all instances of the subset have been executed then the confidence is 1.

Wilcoxon test

There is a large body of literature on statistical tests, and many of them can be used in the context of racing algorithms [3]. For instance, the F-Race [2] algorithm uses a Friedman two-way analysis of variance by ranks. However, this test concerns a family of candidates, while here, we are interested in an ordered pair of algorithms. When only two configurations remain, the F-Race algorithm switches to a *Wilcoxon matched-pairs signed-ranks test* [6], because it is more powerful and data-efficient than the Friedman test in that scenario [26].

The test we want to apply should satisfy the following requirements: it should be nonparametric, it should be applicable to paired data. Such a test would *not need any background knowledge*. We chose the Wilcoxon test because it satisfies our requirement while exploiting other properties of our data: data is measured on an interval scale, the differences (between running times) are symmetric and the magnitudes of the differences between our paired data is exploited. This test assumes that running times are independent and the two samples are mutually independent, that is not truly the case; however, we find that assuming independence is a good first approximation. This test is only based on observed data, it does not take into account the remaining instances. Through hypothesis testing we can find out

³ Note that this does not ensure a ratio of γ for the total running time, as the total running time of A_{ch} over all instances is not available and therefore cannot be used for discrimination.

when there is enough evidence to stop, at which point the best algorithm is the one with the lowest mean running time. In this case, our confidence threshold C_{thres} is compared to the p-value of the alternative two-sided hypothesis. Let us note that other statistical tests, such as the Mann-Whitney U test, the permutation test, the Kolmogorov-Smirnov test, or the paired t-test do not satisfy our assumptions.

The distribution-based discrimination method

This method requires statistics-based background knowledge. Let us consider the following random variable computing the difference in performances:

$$\Delta_{tot} = \underbrace{\sum_{I \in \mathcal{I}_{run}} rt(A_{ch}, I) - rt(A_{inc}, I)}_{\text{constant}} + \sum_{I \in \mathcal{I}_{torun}} \underbrace{rt(A_{ch}, I)}_{\text{random variable}} - \underbrace{rt(A_{inc}, I)}_{\text{constant}}.$$

We are interested in determining the sign of Δ_{tot} , meaning which of the two algorithms performs best. For a fixed confidence threshold $C_{thres} = 1 - \varepsilon$, we estimate $\mathbb{P}(\Delta_{tot} > 0)$ and stop if:

- $\mathbb{P}(\Delta_{tot} > 0) \geq 1 - \varepsilon$, in which case A_{ch} performs worse than A_{inc} ,
- or $\mathbb{P}(\Delta_{tot} > 0) \leq \varepsilon$, meaning $\mathbb{P}(\Delta_{tot} \leq 0) \geq 1 - \varepsilon$, i.e. A_{ch} performs better than A_{inc} .

The confidence is $\mathbb{P}(\Delta_{tot} > 0)$ for the former case and $1 - \mathbb{P}(\Delta_{tot} > 0)$ for the latter. Looking at the definition of the random variable Δ_{tot} , its probability law can be described using translations and convolutions of the distributions $(\delta_I)_{I \in \mathcal{I}_{torun}}$. In practice, many natural classes of distributions (for instance Gaussian and Cauchy distributions) are closed under translations and convolutions, so $\mathbb{P}(\Delta_{tot} > 0)$ can be effectively computed or approximated.

Because running times are positive and algorithms are stopped when they reach the cutoff time T_{cut} , the running times are bounded. A distribution matching this behaviour would be a truncated distribution, but most are not closed under convolution, which we have stated above as a necessary property, so they cannot be used directly. Nevertheless, the sum of the bounds on individual running times can be used as bounds for Δ_{tot} , which we can model as a truncated distribution. For heavy-tailed distributions, such as the Cauchy distribution, the confidence is higher with a truncated distribution than without, as impossible cases are not taken into account, enabling to stop earlier.

4 Experimental setup

To empirically evaluate our approaches, we implemented them and ran them on all ordered pairs of algorithms from well-known benchmark scenarios.

4.1 Datasets

We use ASlib [4], a benchmark library for algorithm selection that contains datasets from competitions for various challenging problems, including Boolean satisfiability and constraint programming. It provides very relevant data on which our strategies can be tested, because such problems are the typical use-case scenario that we envisioned.

From ASlib, we use three datasets: the CSP MiniZinc 2016 (20 algorithms, 100 instances) dataset, which comprises performance data from the 2016 MiniZinc Challenge ('Free Search' Category) [19, 28]; the BNSL 2016 (8 algorithms, 1178 instances) dataset [21] from Bayesian Network structure learning; the SAT18 (37 algorithms, 353 instances) dataset, which consists of performance data from the EXP track of the 2018 SAT Competition [10]; and, to account

■ **Table 1** Discrimination difficulty of our datasets according to our metric.

Dataset	CSP MiniZinc	BNSL	SAT 18	SAT 20
Mean	59.74	9.363	2458	78.88
Median	3.28	1.15	9.65	5.51
Top-3 mean	24.7	77.5	47.7	49.9

for more recent advances in SAT, we created the SAT20 (67 algorithms, 400 instances) dataset from the results of the main track of the 2020 SAT Competition [1]. Those datasets were chosen to cover a broad range of prominent problems and instance sets.

For our feature-based approaches, we decided to replace missing features by the mean value, as done by Hutter et al. [16]. Hence, no information can be extracted from such instances from a distributional point of view.

Discrimination difficulty

To get a sense of how difficult it is to discriminate between the algorithms from each dataset, we introduce a measure of difficulty based on how different the algorithms behave on our set of instances. We propose to use the following ratio: $\mathcal{D}_{discr}(A_{inc}, A_{ch}) = \frac{\sum_{I \in \mathcal{I}} \text{median}[(rt(A, I))_{A \in \mathcal{A}}]}{|\sum_{I \in \mathcal{I}} rt(A_{ch}, I) - rt(A_{inc}, I)|}$. This measure has been chosen, because it grows when the two algorithms have similar performance, and it is invariant under scaling, so that the difficulty remains the same if running times are multiplied by a constant factor. It is also symmetric: exchanging A_{inc} and A_{ch} leads to the same result.

In Table 1, we report the mean difficulty, the median difficulty over all pairs and the mean difficulty of the subset of the best 3 algorithms for all of our datasets. Based on this measure, we expect it to be easy to discriminate between algorithms from BNSL, while SAT18 should provide a bigger challenge. The large discrepancy between the mean and median value, seen for SAT18 in particular, is caused by small groups of algorithms with very close performances. Pairs of algorithms from those groups usually have very high difficulty, reaching up to a million for SAT18, which affects the mean.

4.2 Implementation details

Our implementation is available on GitHub (see supplementary materials).

To estimate the parameter of running time distributions, we use maximum likelihood estimation; and we use a Cauchy distribution for the distribution-based discrimination method, as motivated in Section 5.2. For the timeout correction, the seed was set to 0.

For the random instance selection method, the seed was also set to 0. The parameter ρ for the discrimination-based selection method was set to 1.2. For the information-based method, we use the expression of Δ_{tot} defined for the distribution-based method, and to compute the expected value, which is an integral, we use Simpson’s rule.

For the Wilcoxon discrimination method, Conover [6] recommends at least 20 samples; however, this would represent up to 20% of our instance for the CSP Minizinc dataset. Thus, we decided to follow `irace` [20], which requires 5 samples in a context similar to ours. We found no significant performance change between different methods for managing zero differences, when paired data from both population is equal, as such we report the performance using Pratt’s method [23].

5 Estimation of the running time distribution

Our approach relies heavily on our ability to estimate the distribution of running times of algorithms on the instances. This distribution is used in 3 out of the 5 instance selection methods and one of our 3 discrimination methods. As such, the choice of the distribution could significantly impact the performance of those strategies. Fitting a distribution on our data requires us to decide how to handle the cutoff time and which distribution to use.

We note that when predicting a running time, a log transformation is typically used [12, 16]. This transformation showcases better performance for predicting running times, because running times distributions tend to be heavy-tailed as shown in the work of Gomes et al. [9]. Since in our case we are mostly interested in predicting the mean or the sum over instances, we do not apply this log transformation.

5.1 Handling censored running times

As explained in the problem definition, after a given cutoff time T_{cut} , the given algorithm is stopped. Running times are thus right-censored, which limits our ability to estimate the true distribution.

Our method for handling time-outs is based on the one proposed by Hutter et al. [13], which itself is based on a prior work from Schmee & Hahn [25]. The resulting algorithm is Algorithm 2 for instance I , with parameters $M \in \mathbb{N}$ and $t_{max} \in \mathbb{R}_+$.

■ **Algorithm 2** Correcting timeouts for a sample $(t_{I,A})_{A \in \mathcal{A}}$.

```

1: fit Distribution on  $(t_{I,A})_{A \in \mathcal{A}}$  without the timeouts
2: set  $N$  to the number of timeouts in  $(t_{I,A})_{A \in \mathcal{A}}$  and  $n$  to 0
3: while not converged do
4:   set  $S$  to  $M \cdot N + n$  samples from Distribution then increment  $n$ 
5:   for  $k = 1$  to  $N$  do
6:     set  $q_k$  to quantile  $\frac{k}{N+1}$  of  $S$ 
7:     replace timeout  $k$  with  $\min(q_k, t_{max})$  in  $(t_{I,A})_{A \in \mathcal{A}}$ 
8:   end for
9:   fit Distribution on  $(t_{I,A})_{A \in \mathcal{A}}$ 
10: end while
11: return Distribution and  $(t_{I,A})_{A \in \mathcal{A}}$ 

```

There is a slight difference from the original algorithm, in the fact that at each iteration, we increment the number of samples used to enable convergence when there is a majority of timeouts on an instance. The parameter M enables to reduce the sampling variance; it is most important on instances with many timeouts. The parameter t_{max} prevents overly large variations of the samples. There are two steps in this algorithm: first we estimate the parameters of the distribution, second we replace the timeouts in the sample. They are repeated until convergence, when the estimated parameters of the distribution are stable. We decided to stop, when the squared difference between the parameters between two iterations is less or equal to 1. Schmee & Hahn [25] use the mean instead of the quantiles of a sample; however, heavy-tailed distributions such as the Cauchy distribution have an undefined mean. We chose to use the sampling approach used by Hutter et al. [13] which enabled them to translate the incertitude and improved the likelihood for their random forest models.

■ **Table 2** Median log likelihood of Maximum Likelihood Estimation for Levy and Cauchy distributions over the instances of each dataset. The highest likelihood for each dataset is shown in boldface.

	CSP MiniZinc	BNSL	SAT 18	SAT 20
Levy	-129.6	-58.08	-299.7	-573.5
Cauchy	-107.5	-62.88	-183.8	-364.9

5.2 Choosing a distribution

What is the distribution satisfying the imposed constraints that gives the best performances?

In practice, since only a set of running times are provided, the distribution parameters must be estimated. We explained how the parameters were estimated in practice in Section 4.2, where here, we explain our choice of distribution. This choice can be motivated by choosing the best candidate distribution that has the lowest error on the set of all instances.

Since many running time distributions are heavy-tailed, we tested two heavy tailed distributions on our four datasets. We report on Table 2 the median log likelihood for each distribution; the parameters of these distributions were estimated using maximum likelihood estimation. The Cauchy distribution provides a clear advantage over the Levy distribution. The only case in which the Levy distribution yields a higher likelihood shows a much smaller difference between the two distributions.

6 Experiments

We designed and conducted extensive experiments, in order to answer the following questions:

Q1 – Can our strategies reduce the CPU time required for evaluating a new algorithm?

Q2 – How do the selection methods affect the accuracy of the strategies?

Q3 – Can our strategies discriminate well between top ranking algorithms?

A run consists of selecting an ordered pair (A_{ch}, A_{inc}) and running the strategy. On each run, all strategies have to compare the same A_{ch} and A_{inc} . In all of our experiments, we ran all of our strategies on each ordered pair of a given dataset.

6.1 General Performance Comparison

To answer Q1, we plotted our strategies in Figure 1, with a target confidence threshold $C_{thres} = 0.95$ (see Algorithm 1). For each of them, the y-axis shows accuracy (in percent) and the x-axis the median time used over all ordered pairs of algorithms, as defined in Section 3. As this corresponds to a multi-objective setup, we highlight the Pareto fronts induced by our results. This does not imply that we can produce a strategy that follows the Pareto front between points; however, by changing the confidence threshold C_{thres} , we can obtain local curves around the performance of each strategy (see Section 6.2). Note that while we show the performance of our strategies without applying a penalty for timeouts, using penalty coefficients from $[1; 10]$ did not affect our findings.

On all datasets, we observe that our random baseline (random sampling a subset of 20% of the instances) shows rather strong performance, with 89% to 100% accuracy for about 20% running time. Further investigation (see Section 6.2) shows that the accuracy of the random baseline increases steeply as we add more instances, until reaching about 20% of the instances, after which the increase in accuracy is substantially slower. Thus increasing the amount of instances does not lead to significantly higher accuracy. Moreover, more than half of the time, this strategy takes 17 to 22% of the running time, which means that the running

times of the instances follow a distribution such that they are as many easy instances as hard ones. We expect that this behaviour is linked to the nature of the competition datasets we are using; instances were gathered by experts to be representative and to show various levels of difficulty. We also note that the BNSL dataset, which is the one that gives the largest advantage to the random baseline, contains very few instances that are not solved within the cutoff time. Choosing these instances incurs a high penalty, because they offer no new information for deciding between the two algorithms, while using up a large amount of running time (see Appendix A).

On all datasets we see that the Wilcoxon method is superior and reaches the desired accuracy in less than 15% of the time; it thus represents the left-hand side of our Pareto front. The subset baseline uses consistently around 20% of the time but hardly reaches 90% accuracy on the hardest dataset; it contributes to the Pareto front only for BNSL. The distribution-based method tends to be more conservative and run longer but often reaches higher accuracies than the desired C_{thres} and thus marks the right-hand side of our Pareto front on our two SAT scenarios; however, it performs very poorly on BNSL, which is the scenario with the least background knowledge due to its low number of algorithms.

The instance selection methods do not show such a clear pattern. We notice, however, that the information-based method lies near or on the Pareto front when combined with Wilcoxon, whereas the discrimination-based and variance-based methods show strong performance when used in combination with distribution-based discrimination.

The evaluated strategies reach up to 95.5% accuracy using 8.21% of the time on the MiniZinc dataset, 95.6% accuracy using 12.3% of the time on SAT18, and 97.1% accuracy using 4.96% of the time on SAT20. For the BNSL dataset, we observed a surprising 100% accuracy while using only 0.0001% of the time using the discrimination-based selection with Wilcoxon discrimination that is hidden behind on Figure 1b, running a median number of 6 instances. The observed performance of our strategies is consistent with the ranking of the datasets according to our difficulty metric (see Table 1 in Section 4) for the distribution-based methods, but not for Wilcoxon, where SAT20 should have been harder than MiniZinc. Overall, in the worst-case, we manage to save 87.6% of CPU time while being 95.6% accurate and in the best case, we saved 95.0% of CPU time while being 97.1% accurate.

6.2 Accuracy over time

To answer Q2, we ran our strategies without stopping criterion, measuring regularly the percentage of accuracy and the time spent running A_{ch} . Figure 2 shows the accuracy (in percent) of the Wilcoxon and distribution-based discrimination methods on all our datasets.

Unlike Figure 1, which did not show any clear pattern regarding the instance selection methods, this analysis reveals two groups of methods. On all but the BNSL dataset, the information-based, variance-based and discrimination-based selection methods lead to a very high accuracy after 55 to 60% running time. This is consistent with the ratio of instances for which most algorithms time out, thus providing little discriminatory power. The feature-based method shows the lowest accuracy, and the random sampling comes in second to last after 40% of the running time.

The BNSL dataset is different, due to a low number of timeouts and large performance differences between the algorithms. In this case, randomly sampling instances offers very good accuracy after a few instances. None of the selection methods offers a clear advantage, because all instances provide evidence towards the algorithm performing best. This suggests that on easy datasets, the random method is a good choice, while on harder datasets containing instances that are not solved within the given cutoff time, more sophisticated selection methods can save running time.

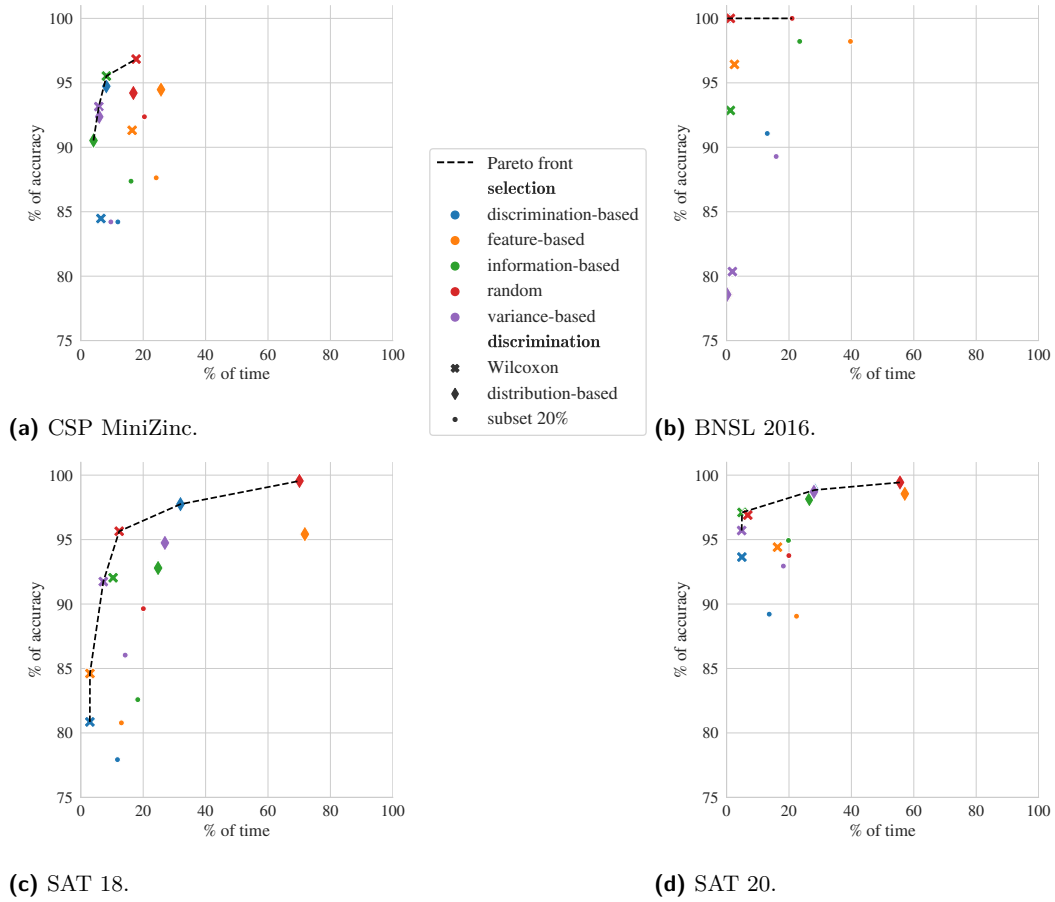


Figure 1 Accuracy over median running time. *y-axis*: percentage over all ordered pairs of algorithms in the dataset. *x-axis*: the time spent running the new algorithm.

6.3 Top ranking

To answer our last question, we decided to keep the top 10 ranking algorithms of the SAT20 dataset and use our strategies on this new dataset; this reflects that fact that often, the primary interest is in discriminating between top-ranking algorithms, be it to compare a new algorithm to the state of the art or to discriminate between the winners of a competition. As per our difficulty measure introduced in Section 4.1, the mean difficulty of the dataset thus obtained is 163, and the median is 22, which is higher than for any of our other datasets. Furthermore, the number of algorithms is reduced, which should reduce the performance of our methods based on prior knowledge. We report the results in Figure 3 analogous to what was done in Section 6.1; for comparative purposes, we also plot the performances on the full SAT20 dataset. The performance of the subset method decreases by more than 10% in accuracy. The distribution-based discrimination method requires more time for this subset, and the discrimination-based selection method drops out of the Pareto front. Because they require prior knowledge, these methods encounter difficulties with this more challenging dataset. The Wilcoxon method is least affected, since it does not depend on prior knowledge; consequently, 3 out of the 4 strategies on the Pareto front use this method. The selection methods in combination with the Wilcoxon test are affected in different ways.

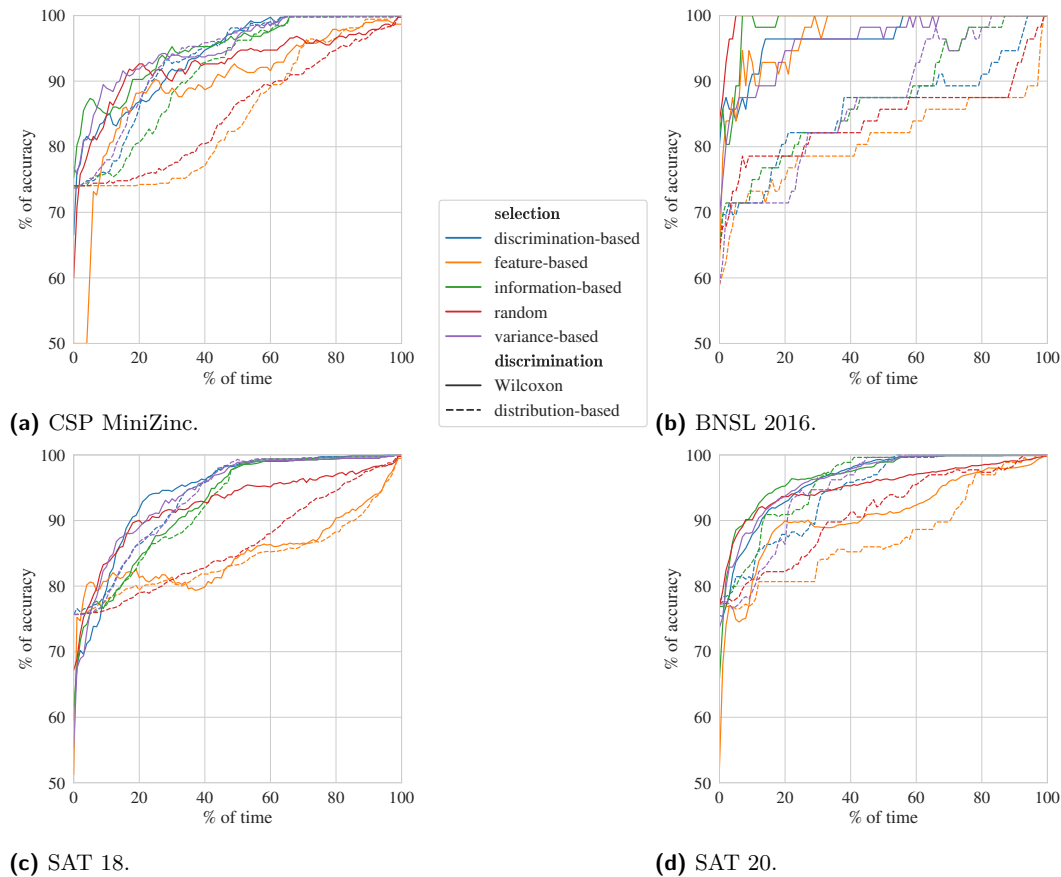


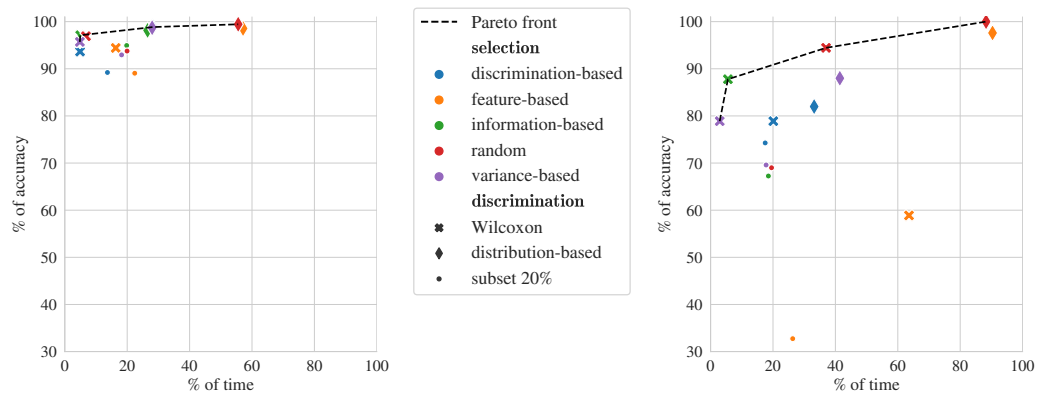
Figure 2 Accuracy over running time used. *y-axis*: percentage over all ordered pairs of algorithms in the dataset. *x-axis*: time spent running the new algorithm.

The information-based and variance-based approaches lead to a quick but less accurate decision, while random sampling leads to a slower decision, achieving 94.4% accuracy for 37.0% of running time.

In this experiment, which compares algorithms with similarly good performance, the information-based method using the Wilcoxon test suffers less than the other strategies, both in terms of cost and accuracy. All other methods lead to either high cost or poor accuracy.

7 Conclusions and future work

In this work, we have investigated methods for reducing the computational effort required for comparing the performance of two automated reasoning algorithms, while gathering sufficient statistical evidence to correctly identify the solver that performs better on a given set of problem instances. We defined the per-set efficient algorithm selection problem (PSEAS) in Section 2. We studied the case in which the performance of a given algorithm is evaluated based on its running time on a set of instances. We described a set of strategies in Section 3, inspired by related problems from the literature and by novel considerations, and tested these on four datasets covering SAT, CSP and structure learning in Bayesian networks. Our experimental evaluation in Section 6 shows that on these datasets, some of our strategies consistently return the correct answer with at least 95% accuracy, while using less than 15%



(a) SAT 20, full Dataset, 67 algorithms.

(b) SAT 20, top-10 algorithms.

■ **Figure 3** Accuracy over running time used for the full and reduced SAT20 datasets. *y-axis*: percentage over all ordered algorithms' pairs in the dataset. *x-axis*: time spent running the new algorithm.

of the CPU time it would take to run the full comparison. In particular, using a Wilcoxon test to decide when to stop, while deciding the next instance to run based on the expected amount of information it can provide, is consistently near or among the best-performing approaches.

A finer-grained analysis of our instance selection methods (see Section 6.2) provides additional insights. We found that deciding on which instance to run based on its discrimination power, following the work of Gent et al. [8], or simply on a notion of running time variance, has the potential to reduce the time required to take a decision when a significant fraction of the given instances are difficult.

Furthermore, we tested our methods on a smaller but more challenging set of algorithms, keeping the 10 best algorithms of the SAT20 competition. While the overall performance is lower than on the full dataset, the Wilcoxon method still reaches an accuracy of 94.4% in 37.0% of the overall running time. Overall, we found that for easy datasets, which discriminate very different algorithms on instances that can be solved quickly, random sampling offers good performance, but when facing hard instances or comparing well performing algorithms, it is beneficial to use more sophisticated methods.

In future work, it would be interesting to consider randomised algorithms. Incorporating empirical performance models [16] such as the ones used in algorithm configuration [14] and algorithm selection [31] could also open new avenues, e.g., involving the use of active learning methods. Finally, while the scope of our work presented here has been limited to comparing two algorithms, one interesting area of future work is focused extensions to many algorithms, in order to devise principles mechanisms for running competitions and other large-scale performance comparisons more efficiently.

References

- 1 Tomáš Balyo, Nils Froleyks, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, Finland, 2020. URL: <http://hdl.handle.net/10138/318450>.

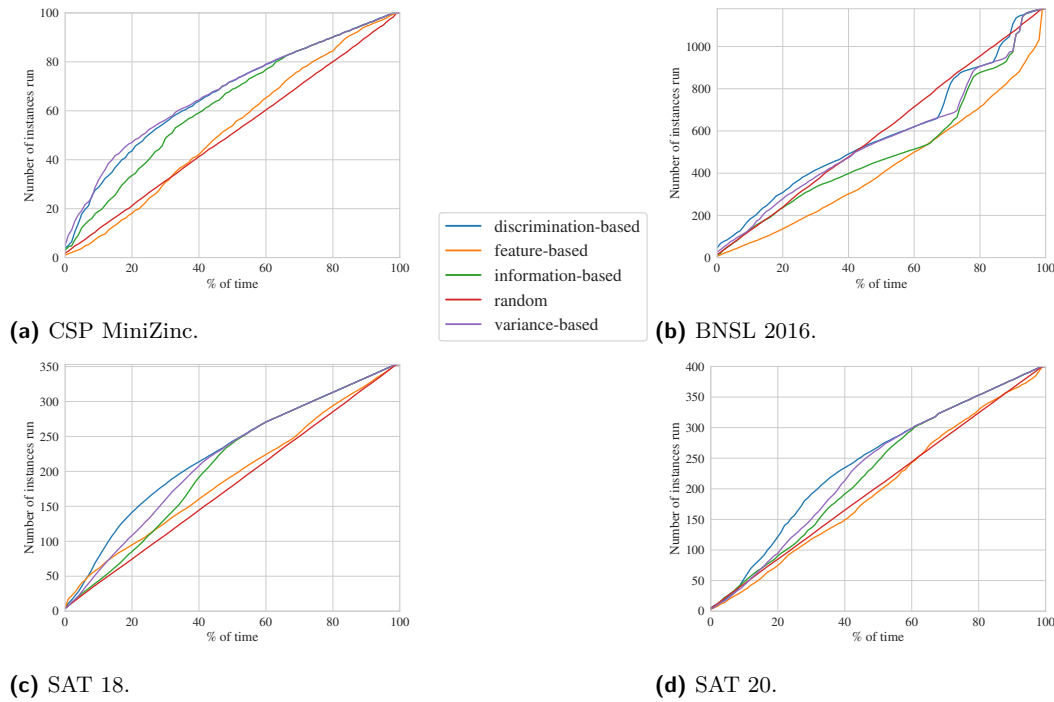
- 2 Mauro Birattari. *Tuning Metaheuristics: A Machine Learning Perspective*. Springer Publishing Company, Incorporated, 1st ed. 2005. 2nd printing edition, 2009. doi:10.1007/978-3-642-00483-4.
- 3 Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, GECCO'02, pages 11–18, San Francisco, CA, USA, January 2002. Morgan Kaufmann Publishers Inc. URL: <http://gpbib.cs.ucl.ac.uk/gecco2002/AAAA223.pdf>.
- 4 Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchet, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. Aslib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58, 2016. doi:10.1016/j.artint.2016.04.003.
- 5 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*, GECCO '21, page 1423–1432, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3449726.3463165.
- 6 William Jay Conover. *Practical nonparametric statistics*, volume 350. John Wiley & Sons, 1998. URL: <https://www.math.ttu.edu/~wconover/book.html>.
- 7 Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960. doi:10.1145/321033.321034.
- 8 Ian P. Gent, Bilal Syed Hussain, Christopher Jefferson, Lars Kotthoff, Ian Miguel, Glenna F. Nightingale, and Peter Nightingale. Discriminating instance generation for automated constraint model selection. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, pages 356–365, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-10428-7_27.
- 9 Carla Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24:67–100, January 2000. doi:10.1023/A:1006314320276.
- 10 Marijn Heule, Matti Järvisalo, and Martin Suda. Sat competition 2018. *Journal on Satisfiability, Boolean Modeling and Computation*, 11:133–154, September 2019. doi:10.3233/SAT190120.
- 11 Holger H. Hoos. Automated algorithm configuration and parameter tuning. In Youssef Hamadi, Eric Monfroy, and Frédéric Saubion, editors, *Autonomous Search*, pages 37–71. Springer, 2012. doi:10.1007/978-3-642-21434-9_3.
- 12 Barry Hurley and Barry O'Sullivan. Statistical regimes and runtime prediction. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 318–324. AAAI Press, 2015. URL: <https://www.ijcai.org/Proceedings/15/Papers/051.pdf>.
- 13 Frank Hutter, Holger Hoos, and Kevin Leyton-brown. Bayesian optimization with censored response data. In *In NIPS workshop on Bayesian Optimization, Sequential Experimental Design, and Bandits*, 2011. arXiv:1310.1947.
- 14 Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, pages 507–523, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-25566-3_40.
- 15 Frank Hutter, Thomas Stützle, Kevin Leyton-Brown, and Holger H. Hoos. Paramils: An automatic algorithm configuration framework. *CoRR*, abs/1401.3492, 2014. arXiv:1401.3492.
- 16 Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014. doi:10.1016/j.artint.2013.10.003.
- 17 Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary Computation*, 27(1):3–45, 2019. doi:10.1162/evco_a_00242.

- 18 Charles L. Lawson and Richard J. Hanson. *Solving Least Squares Problems*. Society for Industrial and Applied Mathematics, 1995. doi:10.1137/1.9781611971217.
- 19 Marius Lindauer, Jan N. van Rijn, and Lars Kotthoff. Open algorithm selection challenge 2017: Setup and scenarios. In Marius Lindauer, Jan N. van Rijn, and Lars Kotthoff, editors, *Proceedings of the Open Algorithm Selection Challenge*, volume 79 of *Proceedings of Machine Learning Research*, Brussels, Belgium, September 11–12, 2017. PMLR. URL: <http://proceedings.mlr.press/v79/lindauer17a.html>.
- 20 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. doi:10.1016/j.orp.2016.09.002.
- 21 Brandon Malone, Kustaa Kangas, Matti Järvisalo, Mikko Koivisto, and Petri Myllymäki. Empirical hardness of finding optimal bayesian network structures: algorithm selection and runtime prediction. *Machine Learning*, 107:247–283, January 2018. doi:10.1007/s10994-017-5680-2.
- 22 Oded Maron and Andrew W. Moore. The racing algorithm: Model selection for lazy learners. *Artif. Intell. Rev.*, 11(1-5):193–225, 1997. doi:10.1023/A:1006556606079.
- 23 John W. Pratt. Remarks on zeros and ties in the wilcoxon signed rank procedures. *Journal of the American Statistical Association*, 54:655–667, 1959. doi:10.1080/01621459.1959.10501526.
- 24 Luca Pulina and Martina Seidl. The 2016 and 2017 qbf solvers evaluations (qbfeval’16 and qbfeval’17). *Artificial Intelligence*, 274:224–248, 2019. doi:10.1016/j.artint.2019.04.002.
- 25 Josef Schmee and Gerald J. Hahn. A simple method for regression analysis with censored data. *Technometrics*, 21(4):417–432, 1979. URL: <http://www.jstor.org/stable/1268280>.
- 26 Sidney Siegel and N John Castellan Jr. *Nonparametric statistics for the behavioral sciences*. McGraw-Hill Book Company, 1988. doi:10.2307/2332896.
- 27 L. Simon, Daniel Leberre, and E. Hirsch. The SAT 2002 Competition. *Annals of Mathematics and Artificial Intelligence*, vol.43, Issue 1-4:307–342, 2005. URL: <https://hal.archives-ouvertes.fr/hal-00022662>.
- 28 Peter James Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The minizinc challenge 2008-2013. *AI Magazine*, 35(2):55–60, 2014. doi:10.1609/aimag.v35i2.2539.
- 29 Li-Li Sun and Xi-Zhao Wang. A survey on active learning strategy. In *2010 International Conference on Machine Learning and Cybernetics*, volume 1, pages 161–166, 2010. doi:10.1109/ICMLC.2010.5581075.
- 30 G. Sutcliffe. Proceedings of the 10th ijcar atp system competition (casc-j 10). *IJCAR*, 2020. URL: <http://www.tptp.org/CASC/J10/Proceedings.pdf>.
- 31 Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008. doi:10.1613/jair.2490.

A Choice of instances

To obtain deeper insight into the instances chosen by our selection methods, we performed some additional empirical analyses. In particular, we plotted the mean number of instances run for each percentage of overall running time used, averaging both quantities over all ordered pairs of algorithms from a given dataset. Figure 4 shows these plots for all our selection methods combined with the distribution-based discrimination method.

We notice that most selection methods give precedence to instances with low running times, as intuitively expected. The feature-based method, however, does not exhibit this behaviour. As in most of our experiments, our methods show significantly different behaviour on BNSL compared to the other datasets (see Figure 4b); here, our methods tend to select instances with running times similar to or higher than instances selected uniformly at random.



■ **Figure 4** Number of instances run over running time used. *y-axis*: number of instances run. *x-axis*: time spent running the new algorithm.

B The ranking-based selection method

B.1 The instance selection method

The ranking-based selection method is a sample-based method, inspired by Bossek & Wagner [5], who developed the explicit-ranking method as a fitness function for evolutionary algorithms, in order to generate instances that follow a given ranking. Their ranking is a lexicographical order, maximising three criteria in sequence: first, the similarity between the algorithms' ranking on the instance and the overall ranking, then two quantities that describe how different the running times of the algorithms are on this instance. The intuition is that given the samples, there is a ranking of algorithms over all instances; we want instances that are good at predicting this ranking – that is, instances on which the algorithms have a ranking closest to the overall ranking. Furthermore, we would also like that given two instances that have the same ranking, the instance that has the highest variance in running times between algorithms is chosen first.

In our case, the desired ranking is the ranking of algorithms with respect to their total performance. We associate each algorithm of \mathcal{A} with an integer, and introduce the desired ranking π , such that $\pi(j)$ is the j th best performing algorithm. Then, for a given instance I , we can define the good pairs as $G_I = \{(j, j+1) \mid rt(A_{\pi(j)}, I) \leq rt(A_{\pi(j+1)}, I)\}$ and the bad pairs as $B_I = \{(j, j+1) \mid rt(A_{\pi(j)}, I) > rt(A_{\pi(j+1)}, I)\}$. The order in which the instances are run is the lexicographical order of the scoring function over instances:

$$score(I) = (|G_I|, \frac{f_B(I)}{median[(rt(A, I))_{A \in \mathcal{A}}]}, \frac{f_G(I)}{median[(rt(A, I))_{A \in \mathcal{A}}]}),$$

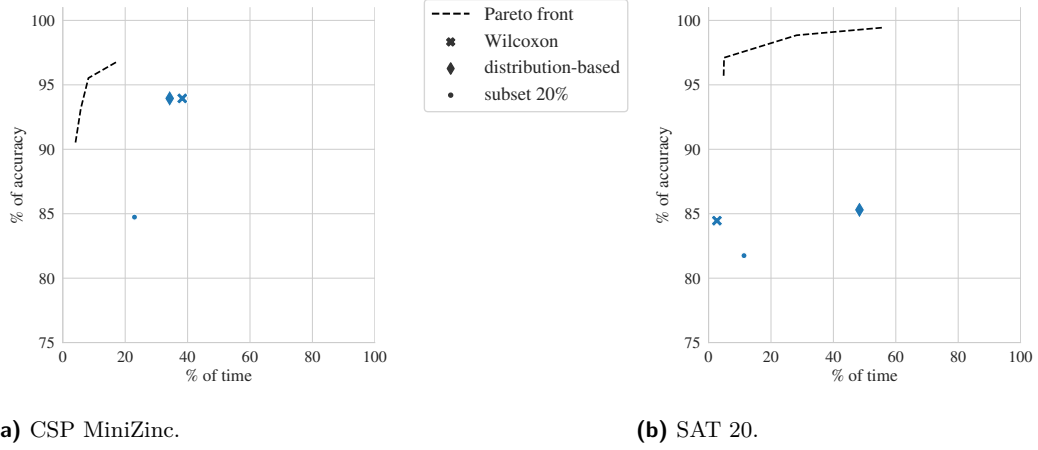


Figure 5 Percentage of accuracy with over median running time for the ranking-based selector. The Pareto front represents the front obtained from our results in Section 6.1. *y-axis*: percentage over all ordered pairs of algorithms in the dataset. *x-axis*: time spent running the new algorithm.

where

$$f_B(I) = \sum_{(j,j+1) \in B_I} (rt(A_{\pi(j+1)}, I) - rt(A_{\pi(j)}, I))$$

$$f_G(I) = \begin{cases} \sum_{(j,j+1) \in G_I} (rt(A_{\pi(j+1)}, I) - rt(A_{\pi(j)}, I)) & \text{if } |G_I| > 0 \\ -\infty & \text{else.} \end{cases}$$

As a normalisation step in the context of running time minimisation, we divided the original f_B and f_G by the median running time, which was not done by Bossek & Wagner [5]. The score is computed once in the beginning of Algorithm 1 and does not need to be updated at line 7.

B.2 Experimental results

Figure 5 shows the performance of the ranking-based selection methods combined with the three discrimination methods described in Section 3.2 (Wilcoxon, distribution-based and subset) on the CSP MiniZinc and on the SAT20 datasets. The results are presented the same manner as those in Section 6.1 and compared to the Pareto front obtained from the results of the methods presented there. As seen in the figure, in all but one case, the ranking-based approach is dominated in terms of performance by our methods discussed earlier. The one exception was observed on the SAT20 dataset, where combined with the Wilcoxon discrimination method, the ranking-based approach exhibits very short running time, but low accuracy, and thus shows performance to the lower left of our previously observed Pareto front. On the CSP MiniZinc dataset, the same combination of methods achieves good accuracy at the cost of high running time.

C Bias analysis

To better understand the behaviour of the discrimination component, we investigated the confidence achieved by our discrimination strategy. This should be correlated with the accuracy of the outcome (as described in Section 3.2). To test this, we ran our strategies,

without stopping criterion, measuring regularly the percentage of accuracy with respect to the confidence level of the discrimination component. Figure 6 shows the percentage of confidence over the accuracy for the distribution-based discrimination method. The black line indicates the desired behaviour where confidence is equal to accuracy. Points above this line represent overconfidence, while points below the line reflect underconfidence.

The discriminator starts, after one instance, with a confidence level of 90% for MiniZinc and 78% for SAT20, while the accuracy is around 75% in both cases. On CSP MiniZinc, it stays highly overconfident until the end, though the gap between confidence and accuracy diminishes. On SAT 20, confidence changes with accuracy, although we observe a tendency for underconfidence, except for the feature-based instance selector. This confidence discrepancy is clearly dependent on the dataset and does not seem correlated with our difficulty measure of the dataset.

Compared to the general performance from Figure 1, the closer the strategies are to correctly estimating the accuracy around our criterion of 95% of confidence, the better they perform.

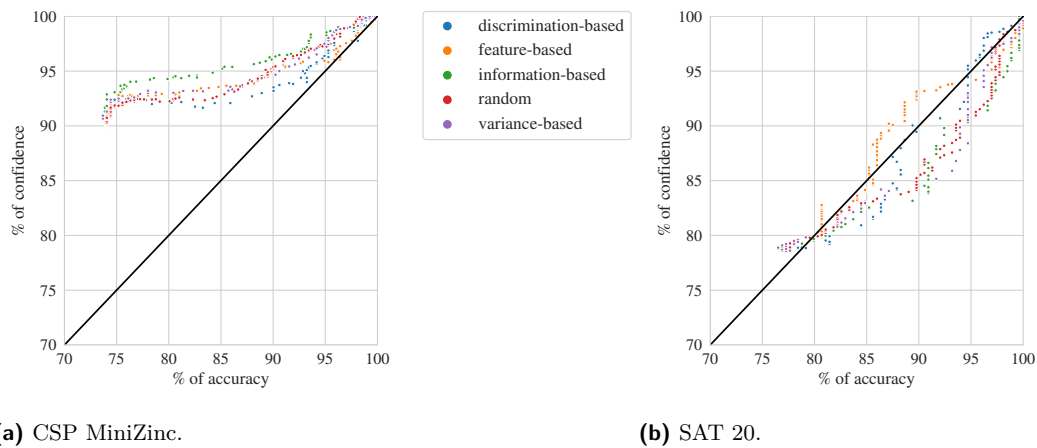


Figure 6 Percentage of confidence with respect to percentage of accuracy of the distribution-based discrimination for all instance-selection methods. *y-axis*: percentage of confidence over all ordered pairs of algorithms in the dataset. *x-axis*: percentage of accuracy over all ordered pairs of algorithms in the dataset.

Enabling Incrementality in the Implicit Hitting Set Approach to MaxSAT Under Changing Weights

Andreas Niskanen 

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

Jeremias Berg 

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

Matti Järvisalo 

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

Abstract

Recent advances in solvers for the Boolean satisfiability (SAT) based optimization paradigm of maximum satisfiability (MaxSAT) have turned MaxSAT into a viable approach to finding provably optimal solutions for various types of hard optimization problems. In various types of real-world problem settings, a sequence of related optimization problems need to be solved. This calls for studying ways of enabling incremental computations in MaxSAT, with the hope of speeding up the overall computation times. However, current state-of-the-art MaxSAT solvers offer no or limited forms of incrementality. In this work, we study ways of enabling incremental computations in the context of the implicit hitting set (IHS) approach to MaxSAT solving, as both one of the key MaxSAT solving approaches today and a relatively well-suited candidate for extending to incremental computations. In particular, motivated by several recent applications of MaxSAT in the context of interpretability in machine learning calling for this type of incrementality, we focus on enabling incrementality in IHS under changes to the objective function coefficients (i.e., to the weights of soft clauses). To this end, we explain to what extent different search techniques applied in IHS-based MaxSAT solving can and cannot be adapted to this incremental setting. As practical result, we develop an incremental version of an IHS MaxSAT solver, and show it provides significant runtime improvements in recent application settings which can benefit from incrementality but in which MaxSAT solvers have so far been applied only non-incrementally, i.e., by calling a MaxSAT solver from scratch after each change to the problem instance at hand.

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

Keywords and phrases Constraint optimization, maximum satisfiability, MaxSAT, implicit hitting set approach

Digital Object Identifier 10.4230/LIPIcs.CP.2021.44

Supplementary Material *Software (Source Code and Experiment Data):*

<https://bitbucket.org/coreo-group/incremental-maxhs/>

archived at `swb:1:dir:9102ca0393f3004eefdd4468b8ba96346cc34476`

Funding Work financially supported by Academy of Finland under grants 322869, 328718 and 342145.

1 Introduction

Maximum satisfiability (MaxSAT) constitutes today a viable approach to solving various types of NP-hard real-world optimization problems (see [6] for a recent survey). This is in particular due to various recent algorithmic advances applied in readily-available MaxSAT solvers. Iteratively solving a series of decision problems with Boolean satisfiability solvers gives a basis for most if not all current state-of-the-art MaxSAT solvers [4, 5]. MaxSAT solvers make heavy use of the incremental APIs of SAT solvers [13], through which SAT solvers can provide explanations as unsatisfiable subsets of soft constraints (i.e., unsatisfiable



© Andreas Niskanen, Jeremias Berg, and Matti Järvisalo;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 44; pp. 44:1–44:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

cores). Two main paradigms adhering to this framework are the core-guided approach (see e.g. [24, 23, 25, 1, 2, 17]) and the implicit hitting set (IHS) approach [11, 12, 10, 29]. In the core-guided approach, cores iteratively obtained from the SAT solvers are used for transforming the original MaxSAT instance in a controlled way so that, once satisfiability is achieved, any satisfying truth assignment reported by the SAT solver constitutes an optimal solution to the original MaxSAT instance. The IHS approach, on the other hand, leaves the original MaxSAT instance unchanged, and computes at each iteration a hitting set of the so-far accumulated set of cores. In the next SAT solver call, the soft clauses contained in the most recently computed hitting set are ignored. This loop is continued essentially (i.e., ignoring various search techniques applied in actual solver implementations of the IHS approach) until the SAT solver returns a satisfying truth assignment.

Both of these SAT-based MaxSAT solving paradigms make heavy use of incremental computations on the level of the SAT solver. However, enabling incremental computations on the actual MaxSAT solving level, i.e., gearing MaxSAT solvers towards solving sequences of related MaxSAT instances without restarting search for each instance, remains today an underdeveloped research direction. Indeed, MaxSAT solvers offer little for such incremental settings, with the exception of a few solver implementations offering an API for adding hard and soft clauses in-between the MaxSAT solver calls [29, 17]. This kind of incremental solving has been further investigated in the context of core-guided solving by adaptively restarting the solver when the quality of the cores degrades [30]. Note that, while so-called incremental cardinality constraints have been proposed and are applied in core-guided MaxSAT solvers [22, 21, 20], this notion refers to incrementality on the SAT-level within the core transformations in the core-guided approach rather than incrementality on the level of MaxSAT, i.e., in incrementally solving a sequence of MaxSAT instances. The lack of support for more generic forms of incrementality on the level of MaxSAT is indeed problematic: various types of recent real-world applications of MaxSAT solvers [19, 9, 33, 27] could evidently benefit in terms of runtime improvements with the help of incremental computations, but currently have to resort to calling a MaxSAT solver from scratch for each instance that needs to be solved towards finding an optimal solution to the problem at hand.

In this work, we make progress on enabling incremental computations on the MaxSAT level. Specifically, we focus on enabling incrementality in problem settings constituting of solving a sequence of MaxSAT instances which differ from each other in the weights of the soft clauses in the instances. In particular, we consider the general setting where the soft clause weights of the next instance in the sequence are adaptively assigned based on the previous instances in the sequence and the optimal solutions found to those instances. Interestingly, this form of incrementality can be identified to be intrinsically present in various application settings of MaxSAT in the context of interpretable machine learning [18, 15, 16, 32] (though various other types of application settings can naturally be imagined), but is not supported by any of the state-of-the-art MaxSAT solvers.

Specifically, we focus on enabling incrementality under changing soft clause weights in the context of the IHS approach to MaxSAT solving. The IHS approach is particularly appealing for incrementality due to the very fact that the solving process does not essentially alter (through core transformations, as in the core-guided approach) the original MaxSAT instance. This allows for ensuring that cores between different MaxSAT instances under changing weights can be reused across the different instances and hence need not be re-computed from scratch. However, the various intricate search techniques and optimizations implemented in the state-of-the-art IHS MaxSAT solver MaxHS make adapting the solver for incremental computations under changing weights a non-trivial task in practice. To this end, we describe

in detail the search techniques that can and cannot be applied in incremental computations under changing soft clause weights, and provide a first IHS solver implementation supporting incrementality under changing weights, building on MaxHS. Most concretely, we apply this incremental adaptation of MaxHS to two recent applications of MaxSAT solving in the context of interpretable machine learning, namely, decision tree boosting and decision set learning, identifying that both of these problem settings could at least in principle benefit in terms of overall runtimes of incremental computations under changing weights. Indeed, we show that our adaptation of MaxHS supporting incrementality under changing weights provides significant runtime improvements compared to a current version of (non-incremental) MaxSAT, despite the fact that not all performance-optimizing techniques applied in the non-incremental version can be applied by the incremental adaptation.

2 Maximum Satisfiability

For a Boolean variable x , there are two literals x and $\neg x$. A clause C is disjunction of literals, viewed as a set, and a (CNF) formula F is a conjunction of clauses, again viewed as a set. A truth assignment τ maps Boolean variables to 1 (true) or 0 (false). The semantics of truth assignments are extended to literals l , clauses C and formulas F in the standard way: $\tau(\neg l) = 1 - \tau(l)$, $\tau(C) = \max\{\tau(l) \mid l \in C\}$ and $\tau(F) = \min\{\tau(C) \mid C \in F\}$. An assignment τ is a model of a formula F if $\tau(F) = 1$. A formula F is satisfiable if it has a model, otherwise it is unsatisfiable.

An instance \mathcal{F} of (weighted partial) MaxSAT consists of two CNF formulas, the hard clauses $H(\mathcal{F})$ and the soft clauses $S(\mathcal{F})$, and a weight function $w(\mathcal{F}): S(\mathcal{F}) \rightarrow \mathbb{R}^+$ that assigns a positive weight to all soft clauses. When the instance \mathcal{F} is clear from context, we use H , S and w to denote $H(\mathcal{F})$, $S(\mathcal{F})$ and $w(\mathcal{F})$, respectively. A model τ of H is a solution to \mathcal{F} . We assume that MaxSAT instances have at least one solution, i.e., that H is satisfiable. A solution τ to \mathcal{F} has cost $\text{COST}(\mathcal{F}, \tau) = \sum_{C \in S} w(C)(1 - \tau(C))$, i.e., the sum of weights of the soft clauses it falsifies. A solution τ is optimal if $\text{COST}(\mathcal{F}, \tau) \leq \text{COST}(\mathcal{F}, \tau')$ holds for all solutions τ' of \mathcal{F} . We denote the cost of the optimal solutions to \mathcal{F} by $\text{COST}(\mathcal{F})$. When convenient, we treat a solution τ as the set of literals the assignment satisfies, i.e., as $\tau = \{l \mid \tau(l) = 1\}$.

In order to simplify notation we will assume that each soft clause $C \in S$ is a unit soft clause containing a negation of a variable, i.e., $C = (\neg b)$. This assumption can be made without loss of generality as any soft clause $C \in S$ can be transformed into the hard clause $C \vee b$ and the soft clause $(\neg b)$ with $w((\neg b)) = w(C)$ where b is a new variable. A variable b that appears in a soft clause $(\neg b) \in S$ is a *blocking* variable; we denote the set of blocking variables of \mathcal{F} by $\mathcal{B}(\mathcal{F})$. As assigning a blocking variable $b = 1$ corresponds to falsifying the corresponding soft clause $(\neg b)$, we treat blocking variables and soft clauses interchangeably, and extend the weight function w to blocking variables by $w(b) = w((\neg b))$ and to sets $B_s \subset \mathcal{B}(\mathcal{F})$ by $\text{COST}(\mathcal{F}, B_s) = \sum_{b \in B_s} w(b)$. The cost of a solution τ is then $\text{COST}(\mathcal{F}, \tau) = \sum_{b \in \mathcal{B}(\mathcal{F})} \tau(b)w(b)$.

The IHS algorithm for computing optimal MaxSAT solutions, focused on in this work, makes use of so-called (unsatisfiable) cores of MaxSAT instances. A core $\kappa \subset S$ is a set of soft clauses that is unsatisfiable together with the hard clauses. As each soft clause $C \in \kappa$ is a negation of a variable $C = (\neg b)$, the fact that $H \wedge \kappa = H \wedge \bigwedge_{(\neg b) \in \kappa} (\neg b)$ is unsatisfiable implies that any solution to \mathcal{F} assigns $b = 1$ for at least one $(\neg b) \in \kappa$. Thus a core can be expressed as the clause $\{b \mid (\neg b) \in \kappa\}$ that is entailed by H , i.e., a clause over blocking variables that is satisfied by all solutions to \mathcal{F} . Note that κ can also be expressed as the

■ **Algorithm 1** IHS for MaxSAT.

```

1 IHS( $\mathcal{F}$ )
  Input: An instance  $\mathcal{F} = (H, S, w)$ 
  Output: An optimal solution  $\tau$ 
2   $lb \leftarrow 0; ub \leftarrow \infty;$ 
3   $\tau_{best} \leftarrow \emptyset; \mathcal{C} \leftarrow \emptyset;$ 
4  while ( $TRUE$ ) do
5     $hs \leftarrow \text{Min-Hs}(\mathcal{B}(\mathcal{F}), \mathcal{C});$ 
6     $lb = \text{COST}(\mathcal{F}, hs);$ 
7    if ( $lb = ub$ ) then break;
8     $(K, \tau) \leftarrow \text{Extract-Cores}(H, \mathcal{B}(\mathcal{F}), hs);$ 
9    if ( $\text{COST}(\mathcal{F}, \tau) < ub$ ) then
       $\tau_{best} \leftarrow \tau; ub \leftarrow \text{COST}(\mathcal{F}, \tau);$ 
10   if ( $lb = ub$ ) then return  $\tau_{best};$ 
11    $\mathcal{C} \leftarrow \mathcal{C} \cup K;$ 

```

$$\begin{array}{ll}
\text{minimize} & \sum_{b \in \mathcal{B}(\mathcal{F})} w(b) \cdot b \\
\text{subject to} & \\
& \sum_{b \in \kappa} b \geq 1 \quad \forall \kappa \in \mathcal{C} \\
& b \in \{0, 1\} \quad \forall b \in \mathcal{B}(\mathcal{F})
\end{array}$$

■ **Figure 1** An integer program for computing a hitting set over a set \mathcal{C} of cores of an instance \mathcal{F} .

linear inequality $\sum_{(-b) \in \kappa} b \geq 1$. We will mostly treat cores as clauses over (or sets of) blocking variables. Given a set \mathcal{C} of cores, a hitting set $hs \subset \mathcal{B}(\mathcal{F})$ is a set of blocking variables that has non-empty intersection with each $\kappa \in \mathcal{C}$. A hitting set hs is minimum-cost if $\text{COST}(\mathcal{F}, hs) \leq \text{COST}(\mathcal{F}, hs')$ holds for all hitting sets over \mathcal{C} . IHS-based algorithms to MaxSAT rely on the well-known fact that hitting sets over sets of cores provide lower bounds on the optimal cost of instances.

► **Proposition 1.** *Let hs be a minimum-cost hitting set over a set \mathcal{C} of cores of an instance \mathcal{F} . Then $\text{COST}(\mathcal{F}, hs) \leq \text{COST}(\mathcal{F})$.*

An important remark to make here for understanding the IHS approach to MaxSAT solving is that Proposition 1 holds for *any* set of cores of an instance. For an minimum-cost hitting set hs over the set \mathcal{C} of *all* cores of \mathcal{F} , it holds that $\text{COST}(\mathcal{F}, hs) = \text{COST}(\mathcal{F})$ and there is a solution τ to \mathcal{F} that sets all blocking variables not in hs to 0.

► **Example 2.** Consider the MaxSAT instance \mathcal{F} with $H = \{(b_1 \vee b_X), (b_2 \vee b_X), (b_3 \vee b_X)\}$ and $\mathcal{B}(\mathcal{F}) = \{b_1, b_2, b_3, b_X\}$ with $w_1(b_1) = w_1(b_2) = w_1(b_3) = 1$ and $w_1(b_X) = 2$. An optimal solution τ_1 to \mathcal{F} is $\tau_1 = \{\neg b_1, \neg b_2, \neg b_3, b_X\}$ and has $\text{COST}(\mathcal{F}, \tau) = \text{COST}(\mathcal{F}) = 2$. The instance has three subset-minimal cores (MUSes): $\kappa_1 = \{b_1, b_X\}$, $\kappa_2 = \{b_2, b_X\}$ and $\kappa_3 = \{b_3, b_X\}$. For a set $\mathcal{C} = \{\kappa_1, \kappa_2\}$ of cores, an example minimum-cost hitting set hs_1 is $\{b_X\}$ which has $\text{COST}(\mathcal{F}, hs_1) = 2 \leq \text{COST}(\mathcal{F})$. If we instead have $w_2(b_1) = w_2(b_2) = w_2(b_3) = 1$ and $w_2(b_X) = 4$, then an optimal solution τ_2 is $\tau_2 = \{b_1, b_2, b_3, \neg b_X\}$ and has $\text{COST}(\mathcal{F}, \tau_2) = 3$. Now a minimum-cost hitting set hs_2 over \mathcal{C} is $hs_2 = \{b_1, b_2\}$ which has $\text{COST}(\mathcal{F}, hs_2) = 2$. Notice that changing weights can significantly alter the minimum-cost hitting sets. Specifically, hs_1 is not minimum-cost w.r.t. w_2 while hs_2 is also a minimum-cost hitting set w.r.t. w_1 .

3 The Implicit Hitting Set Approach to MaxSAT Solving

Algorithm 1 details IHS, the implicit hitting set algorithm to computing an optimal solution to a single MaxSAT instance \mathcal{F} . In short, the algorithm decouples MaxSAT solving into separate core extraction (the **Extract-Cores** subroutine) and an optimization step (the **Min-Hs** subroutine). The core extraction makes use of a SAT solver to extract an increasing number of cores, which are stored in the set \mathcal{C} . As a by-product, the procedure also computes a

solution τ to \mathcal{F} . The solution allows refining the upper bound ub on $\text{COST}(\mathcal{F})$, i.e., comparing $\text{COST}(\mathcal{F}, \tau)$ to the known upper bound ub and updating it if the new solution has lower cost. The optimization steps compute a minimum-cost hitting set hs over the set \mathcal{C} of cores extracted so far using an IP solver. By Proposition 1 the cost $\text{COST}(\mathcal{F}, hs)$ of such a set is a lower bound lb on $\text{COST}(\mathcal{F})$. IHS terminates once $lb = ub$ and returns τ_{best} , the solution for which $\text{COST}(\mathcal{F}, \tau_{best}) = ub$, which is guaranteed to be an optimal solution.

In more detail, when invoked on an instance \mathcal{F} , IHS begins by initializing the lower bound lb to 0, the upper bound ub to ∞ , the best known model τ_{best} to \emptyset and a set \mathcal{C} of cores of \mathcal{F} (represented as sets of blocking variables) to \emptyset (Lines 2-3). Then the main search loop (Line 4-11) is started. During each iteration of the loop, a hitting set hs over \mathcal{C} is computed on Line 5 by solving the integer program detailed in Figure 1 via the procedure **Min-Hs**($\mathcal{B}(\mathcal{F}), \mathcal{C}$). The procedure returns a minimum-cost set of blocking variables hs that contains at least one variable from each $\kappa \in \mathcal{C}$, i.e., a minimum-cost hitting set over \mathcal{C} . The cost $\text{COST}(\mathcal{F}, hs)$ of the set is then used to update the lower bound lb on $\text{COST}(\mathcal{F})$ on Line 6. Since no cores are removed from \mathcal{C} during the execution of IHS, $\text{COST}(\mathcal{F}, hs)$ is non-decreasing over the iterations.

After updating the lower bound, the termination criteria is checked on Line 7. If the known upper bound matches the known lower bound, the algorithm terminates and returns the current best solution τ_{best} . Otherwise, the core extraction step **Extract-Cores** is invoked on Line 8. The procedure uses a SAT solver iteratively in order to extract previously unseen cores of \mathcal{F} in the form of a disjoint set K of cores s.t. each $\kappa \in K$ is a subset of $\mathcal{B}(\mathcal{F}) \setminus hs$. The cores are extracted using the assumption interface offered by most modern SAT solvers [13, 8] that allows inputting a CNF formula F and a set \mathcal{A} of assumptions in the form of literals. The SAT solver then solves the formula $F \wedge \bigwedge_{l \in \mathcal{A}} (l)$ and returns either (i) a model τ of F that sets $\tau(l) = 1$ for all $l \in \mathcal{A}$ or (ii) a subset $\mathcal{A}_s \subset \mathcal{A}$ such that $F \wedge \bigwedge_{l \in \mathcal{A}_s} (l)$ is unsatisfiable (which is equivalent to F entailing the clause $\{\neg l \mid l \in \mathcal{A}_s\}$).

Extract-Cores invokes the internal SAT solver on the hard clauses H under the assumptions $\{\neg b \mid b \in \mathcal{B}(\mathcal{F}) \setminus hs\}$. If the SAT solver reports unsatisfiability, the subset of assumptions returned by the SAT solver corresponds to a core of \mathcal{F} . The literals in the core are then removed from the assumptions and the SAT solver reiterated. The procedure terminates when K is a maximal set of disjoint cores over $\mathcal{B}(\mathcal{F}) \setminus hs$ and returns K and τ , the final model returned by the SAT solver that satisfies the hard clauses and all soft clauses that are not in hs nor any of the cores in K .

Since τ satisfies H , it is a solution to \mathcal{F} . Thus its cost $\text{COST}(\mathcal{F}, \tau)$ is compared to the current upper bound ub and updated if needed on Line 9. If the new bounds match, the algorithm terminates on Line 10. Otherwise, the new cores in K are added to \mathcal{C} and the loop reiterated. An important intuition here is that all cores in K are disjoint from the hitting set hs and are thus not hit by hs . Adding the new cores to \mathcal{C} results in hs not being a hitting set over \mathcal{C} in subsequent iterations. With this intuition, the termination of IHS follows by the finite number of cores and hitting sets of \mathcal{F} . A detailed argument for the correctness of IHS can be found in [3].

► **Example 3.** Invoke IHS on the instance \mathcal{F} from Example 2 with $w(\mathcal{F}) = w_1$. In the first iteration, the set \mathcal{C} of cores is empty, so **Min-Hs** returns an empty hitting set $hs = \emptyset$ which does not allow increasing the lower bound. At this point $0 = lb < \infty = ub$ so IHS does not terminate but instead moves on to **Extract-Cores** to extract a disjoint set of cores over $\mathcal{B}(\mathcal{F}) \setminus hs = \{b_1, b_2, b_3, b_X\}$. There are a number of different possibilities that could be returned. However, all of them contain at most one core that contains at least one of the variables b_1, b_2 or b_3 . Say **Extract-Cores** returns $K = \{b_1, b_X\}$ and $\tau = \{\neg b_1, \neg b_2, \neg b_3, b_X\}$.

Since $\text{COST}(\mathcal{F}, \tau) = 2 < \infty = ub$ the upper bound is updated to 2 and the best model τ_{best} is set to τ . At this point, IHS has found an optimal solution to \mathcal{F} . However, since $lb = 0 \neq 2 = ub$ the algorithm does not terminate, but instead augments \mathcal{C} with $\{b_1, b_X\}$ and reiterates. Informally speaking, the optimality of τ_{best} has not been proven yet.

In the next iteration **Min-Hs** is invoked with $\mathcal{C} = \{\{b_1, b_X\}\}$. There exists one minimum-cost hitting set $hs = \{b_1\}$ over \mathcal{C} . This hitting set allows refining $lb = 1$ and **Extract-Cores** to extract one more core that is a subset of $\mathcal{B}(\mathcal{F}) \setminus hs = \{b_2, b_3, b_X\}$, say $\{b_2, b_X\}$. In the next iteration, **Min-Hs** computes either $hs = \{b_1, b_2\}$ or $hs = \{b_X\}$. In both cases $lb = \text{COST}(\mathcal{F}, hs) = 2 = ub$ so the algorithm terminates, and returns τ_{best} .

We end this section by discussing abstract cores, a recently proposed improvement to IHS [8]. In short, an abstract core is a compact representation of a large – potentially exponential – number of regular cores that the IHS algorithm can reason with more efficiently. In more detail, an abstraction set $ab \subset \mathcal{B}(\mathcal{F})$ is a subset of n blocking variables that are augmented with count variables $ab.c[1] \dots ab.c[n]$. Informally speaking, the count variables count the number of variables in ab set to true. More precisely, the *definition* of the count variable $ab.c[k]$ is the constraint $ab.c[k] \leftrightarrow \sum_{b \in ab} b \geq i$. An abstract core of an instance \mathcal{F} w.r.t. a collection AB of abstraction sets is then clause κ that: (i) contains only blocking variables or count variables and (ii) is entailed by the conjunction of hard clauses of \mathcal{F} and the definitions of count variables. Following [8] we require that all of the blocking variables assigned to the same abstraction set ab have the same weight. This allows the count variables of ab to have well-defined weights; each count variable of ab being assigned to 1 corresponds to one more $b \in ab$ also being assigned to 1, incurring $w(b)$ more cost.

For some intuition on their usefulness, note that an abstract core κ containing a count variable $ab.c[i]$ corresponds to $\binom{|ab|}{|ab|-i+1}$ non-abstract cores where the count variable $ab.c[i]$ variable is exchanged with any subset of ab containing $|ab| - i + 1$ elements. More details can be found in [8].

An IHS algorithm using abstract cores, **ihs-abscores**, extracts both abstract and regular cores during search. Additionally it maintains and dynamically updates a collection AB of abstraction sets over which the abstract cores are then extracted. The abstraction sets are computed based on a graph G that initially has the blocking variables as nodes and an edge between any two variables with the same weight that have been found in a core together. The weight of each edge in G between the nodes n_1 and n_2 is the number of times that the variables corresponding to n_1 and n_2 have appeared in cores together. The abstraction sets are then computed by clustering G and using the clusters as abstraction sets. The intuition here is that we wish two variables that often appear in cores together (and are as such in some sense related) to be included in the same abstraction set. During search the quality of the abstraction sets in AB are monitored. If the extracted (abstract) cores are not driving up the lb computed by the optimizer (**Min-Hs**), then the graph G is reclustered by merging the nodes in the current clusters and then re-clustering the graph. Note that after re-clustering, one single node in G might correspond to several blocking variables of \mathcal{F} .

4 **MaxSAT with Changing Weights**

We move on to our proposal for extending the IHS approach to MaxSAT for computing optimal solutions to MaxSAT instances under changing weights. After formulating in more detail the incremental problem setting we consider, we will describe an extension of IHS capable of solving sequences of MaxSAT instances with different weights in an incremental fashion.

■ **Algorithm 2** IHS-INC for computing optimal solutions to k different instances of different weights.

```

1 IHS-INC( $\mathcal{F}$ , next-w,  $k$ )
2    $\mathcal{C} \leftarrow \emptyset$     $AB \leftarrow \emptyset$ ;
3    $\tau_{best} \leftarrow \text{SAT}(\text{H}(\mathcal{F}))$ ;
4   for  $i = 1, \dots, k$  do
5     if  $i > 1$  then  $w(\mathcal{F}) = \text{next-w}()$ ;
6     deactivate-abs( $AB, w(\mathcal{F})$ );
7      $ub \leftarrow \text{COST}(\mathcal{F}, \tau_{best})$ ;
8      $\tau_{best} \leftarrow \text{ihs-abscores}(\mathcal{F}, \mathcal{C}, AB, ub)$ ;
9   output  $\tau_{best}$ ;

```

4.1 Problem Formulation

Given a MaxSAT instance \mathcal{F} and k different weights w_i for the soft clauses in \mathcal{F} , our objective is to compute k solutions $\tau_1 \dots \tau_k$ to \mathcal{F} such that each τ_i is an optimal solution w.r.t. the weights w_i . We do not put any requirements on how the weights are computed. Our solution algorithm solves the problem sequentially. In particular, the i th weights w_i can depend on the optimal solutions $\tau_1 \dots \tau_{i-1}$ computed in previous iterations. More formally, we assume that the first weights w_1 are given as part of the input and abstract the computation of all other weights to a black-box oracle **next-w** that is assumed to have access to all information (optimal solutions, previous weights, etc.) from previous iterations.

4.2 Incremental IHS for MaxSAT with Changing Weights

Algorithm 2 details IHS-INC, an extension of the IHS algorithm **ihs-abscores** with abstract cores, for solving a MaxSAT instance \mathcal{F} under k different weight functions. The algorithm takes as input the instance \mathcal{F} , a function **next-w** for computing the weight functions used in subsequent iterations and k , the number of iterations required. After initializing a set \mathcal{C} of cores and a set AB of abstraction sets on Line 2 as well as obtaining an initial solution τ_{best} by invoking a SAT solver on the hard clauses of \mathcal{F} on Line 3, the algorithm enters its main search loop (Lines 4-9).

In each iteration of the loop, the algorithm computes an optimal model w.r.t. the i th weights. Each iteration starts with the new weights being obtained on Line 5 and an initial upper bound ub computed from the current best model τ_{best} on Line 7. In the first iteration, τ_{best} will be the model obtained by checking the satisfiability of the hard clauses (on Line 3). In subsequent iterations, τ_{best} will be the optimal model computed in the previous iteration. Afterwards, an optimal model w.r.t. the current weights is computed using the function **ihs-abscores** implementing the IHS algorithm with abstract cores for computing one optimal solution to the instance.

A central fact to note in IHS-INC is that – on every iteration except the first one – **ihs-abscores** is invoked with a set \mathcal{C} of cores and AB of abstraction sets that are *non-empty*. Indeed, all of the cores and abstract cores that are computed during previous iterations are preserved and used in subsequent iterations as well. Similarly, as many abstraction sets as possible are also preserved between iterations. Recall that **ihs-abscores** assumes that all blocking variables assigned to the same abstraction set ab have the same weight. As the weights of blocking variables can change between iterations, we stop extracting new abstract cores over ab if the weights of the blocking variables in ab are changed to be unequal. More

precisely, we say that an abstraction set ab is valid if $w(b_i) = w(b_j)$ holds for any $b_i, b_j \in ab$. In Algorithm 2 the `deactivate-abs` method loops over the set AB to check which ones are not valid anymore. The `ihs-abscores` method then only extracts abstract cores over valid abstraction sets. However, since the definition of an abstract core is independent from the weights of blocking variables, abstract cores containing count variables in an invalid ab are still preserved and used in subsequent iterations, the definitions of count variables of invalid abstraction sets are kept in the SAT and IP solvers. We also allow blocking variables from invalid abstraction sets to be assigned to other abstraction sets in later iterations. More specifically, the blocking variables from invalid abstraction sets are reintroduced into the graph and allowed to be clustered in later iterations.

The correctness of **IHS-INC** follows from the fact that the definition of a core and an abstract core depends only on the clauses in \mathcal{F} , and the clauses defining count variables. Neither of these change between iterations so all of the cores computed in previous iterations can be kept in subsequent ones.

► **Example 4.** Invoke **IHS-INC** on the instance \mathcal{F} from Example 2 and assume the weight function w_1 for the first instance in a sequence of instance to be solved. To keep the example simple, we also assume that no abstraction sets or abstract cores are used in the execution.

Assume that the initial SAT solver call on Line 3 on the clauses of \mathcal{F} obtains a model $\tau_{best} = \{b_1, b_2, b_3, \neg b_X\}$ and an initial upper bound $ub = \text{COST}(\mathcal{F}, \tau_{best}) = 3$. The algorithm then invokes `ihs-abscores` with $\mathcal{C} = \emptyset$ and $ub = 3$. As `ihs-abscores` without abstract cores corresponds exactly to **IHS** detailed in Algorithm 1, Example 3 details one possible execution when solving \mathcal{F} . After that execution, the procedure returns $\tau_{best} = \{\neg b_1, \neg b_2, \neg b_3, b_X\}$ and updates $\mathcal{C} = \{\{b_1, b_X\}, \{b_2, b_X\}\}$.

Assume then that the weights of \mathcal{F} are updated to w_2 as detailed in Example 2. The new weights are then used to update the upper bound to $\text{COST}(\mathcal{F}, \tau_{best}) = 4$ before `ihs-abscores` is invoked again. In the first iteration of the search loop of `ihs-abscores`, the set \mathcal{C} already contains two cores. As such **Min-Hs** returns the minimum-cost hitting set $hs = \{b_1, b_2\}$ and updates $lb = \text{COST}(\mathcal{F}, hs) = 2$. Afterwards, **Extract-Cores** extracts the core $\{b_3, b_X\}$ and returns (for example) the solution $\tau = \{b_1, b_2, b_3, \neg b_X\}$. This solution has $\text{COST}(\mathcal{F}, \tau) = 3$, so the ub and τ_{best} is updated. In the next iteration, **Min-Hs** computes the hitting set $hs = \{b_1, b_2, b_3\}$ which has $\text{COST}(\mathcal{F}, hs) = 3$ and allows the algorithm to terminate.

Example 4 demonstrates how **IHS-INC** is able to solve the second iteration just by extracting one more core. In contrast, it can be shown that restarting the search from scratch (i.e., invoking **IHS**) results in at least 3 cores being extracted when solving \mathcal{F} with $w(\mathcal{F}) = w_2$ from Example 2.

4.3 Realizing IHS-INC

On an abstract level, as demonstrated by Algorithm 2, **IHS-INC** is relatively straightforward to implement given a procedure for `ihs-abscores`. However, in reality, the engineering aspects are less trivial. We continue by detailing our implementation which is built on top of MaxHS [11, 8], a state-of-the-art IHS MaxSAT solver. In practice this requires several changes to the underlying data structures and procedures of MaxHS, especially those concerning the internal representation of soft clauses and their weights.

4.3.1 Handling Weight Changes

Our goal is to provide an API function `changeWeight(i, w)` which can be called incrementally to change the weight of the i th input soft clause to $w \geq 0$. The necessary changes to MaxHS are applied to the following components.

WCNF Simplification

Before solving, MaxHS performs a series of simplifications to the input instance. In particular, after simplifying, the list of soft clauses is not in general equal to the soft clauses of the input instance¹; since some soft clauses are removed due to either always being satisfied or impossible to satisfy given the hard clauses, and some soft clauses are merged. In order to have access to `changeWeight(i, w)`, we implement a mapping `currentIndex` which takes an index of the original soft clauses as input, and returns either: (a) an index of the internal list of the soft clauses (note that since some soft clauses have been merged, the same index may correspond to several indices of the input instance), (b) `SAT` if the soft clause has been removed since it is implied by the hard clauses or (c) `UNSAT` if the soft clause has been removed because it is unsatisfiable given the hard clauses. Furthermore, we also keep track of all preimages of this map in order to perform updates to it correctly. After simplifying, `currentIndex` will remain constant. During simplification, MaxHS also computes `baseCost` as the sum of the weights of soft clauses which cannot be satisfied, and `totalWeight` as the sum of the weights of soft clauses remaining after simplification. These numbers may also naturally change due to changing weights of the original instance.

In more detail, the simplification procedures are the following:

- *Hardening of soft clauses.* MaxHS checks the input weights of the soft clauses and determines whether some soft clauses can be hardened due to their high weight. Since in our setting such a high weight may change to an arbitrarily low one, this feature is disabled.
- *Unit hard clauses and equalities.* MaxHS performs unit propagation over the hard clauses, checks for equalities implied by the hard clauses, and performs pure literal elimination. Although these procedures do not concern the weights, they may modify the original list of soft clauses. In particular, a soft clause may be satisfied due to e.g. containing a literal which has been assigned to true via unit propagation, in which case the soft clause is removed; we update `currentIndex` by setting the corresponding index to `SAT`. If a soft clause becomes empty due to e.g. containing only literals which have been assigned to false via unit propagation, it is removed and `baseCost` is updated; we update `currentIndex` by setting the corresponding index to `UNSAT`. Finally, tautologies are removed; for a (tautological) soft clause we set the corresponding index to `SAT`.
- *Contradictory unit clauses.* If there is a pair of contradictory unit clauses one of which is soft and the other is hard, the soft clause is falsified, so we update `currentIndex` by setting its index to `UNSAT`. If there is a pair of contradictory soft unit clauses, the base-version of MaxHS would only preserve the clause with higher weight, setting its new weight as the difference and incrementing `baseCost` with the smaller weight. In our setting we need to preserve both; we additionally set the new weight of the lower-weight clause to zero, and keep track of such contradictory unit soft clauses within the `contradictoryUnit` datastructure. In particular, MaxHS initializes blocking variables in such a way that unit soft clauses are used as blocking variables, and new variables are declared only for non-unit softs. In contrast, we declare new blocking variables for unit soft clauses for which `contradictoryUnit` is true.
- *Duplicate clauses.* If there is a pair of duplicate clauses with a hard clause and a soft clause, the soft clause is subsumed by the hard clause. In this case, we update `currentIndex` at the corresponding index to `SAT`. If there is a duplicate of two soft clauses, they are joined into one by setting the weight as the sum of the two weights. We update `currentIndex` by setting the index of the removed soft clause to the same index as the preserved one.

¹ Note that, in contrast to the pseudocode, MaxHS does not assume that every soft clause is a unit negation of a blocking variable. Instead the solver maintains the full clause and declares a blocking variable for it internally.

■ **Algorithm 3** Procedure for changing the weight of the i th soft clause to w .

```

1  changeWeight( $i, w$ )
2  |  $\delta \leftarrow w - \text{originalWeights}[i]$ ;
3  | if  $\text{currentIndex}[i] = \text{UNSAT}$  then
4  |   |  $\text{baseCost} \leftarrow \text{baseCost} + \delta$ ;
5  | else if  $\text{currentIndex}[i] = \text{SAT}$  then
6  |   | return;
7  | else
8  |   | if  $\text{contradictoryUnit}[\text{currentIndex}[i]]$  then
9  |   |   |  $\text{resolve unit softs}$ ;
10 |   | else
11 |   |   |  $\text{totalWeight} \leftarrow \text{totalWeight} + \delta$ ;
12 |   |   |  $\text{weights}[\text{currentIndex}[i]] \leftarrow \text{weights}[\text{currentIndex}[i]] + \delta$ ;
13 |  $\text{originalWeights}[i] \leftarrow w$ ;
14 |  $\text{update CPLEX}$ ;

```

- *Flipping literals.* If there is a unit soft clause with a positive literal, that literal is flipped in the instance in order to use it as a blocking variable (so that setting the blocking variable to true incurs the cost of the soft clause). We only do this in the case that the soft clause does not have a contradictory unit soft clause.

Note that hardening of soft clauses is disabled since it may change the set of cores of the instance being solved and lead to IHS-INC computing cores that are not valid to preserve between iterations.

► **Example 5.** Consider the instance \mathcal{F} with $H = \{(b_1 \vee b_2)\}$ and $\mathcal{B}(\mathcal{F}) = \{b_1, b_2\}$ with $w(b_1) = 1$ and $w(b_2) = 10$. During hardening, MaxHS invokes a SAT solver on H in hopes of finding a good model that allows hardening of soft clauses. Assume that the model $\tau = \{b_1, \neg b_2\}$ is computed. Since $\text{COST}(\mathcal{F}, \tau) = 1 < w(b_2)$ MaxHS concludes that the soft clause $(\neg b_2)$ can be hardened and invokes `ihs-abscores` on the instance $\mathcal{F}^H = \{(b_1 \vee b_2), (\neg b_2)\}$ with $\mathcal{B}(\mathcal{F}^H) = \{b_1\}$. While the optimal solutions of both \mathcal{F} and \mathcal{F}^H are the same, the set of cores are not, $\kappa^H = \{b_1\}$ is an example of a core of \mathcal{F}^H that is not a core of \mathcal{F} . In other words, κ^H could not in general be preserved between the iterations of IHS-INC as it is not a core of any instance where $(\neg b_2)$ can not be hardened.

CPLEX Interface

The underlying IP solver CPLEX, used for solving the hitting set problems, has to be updated between iterations with the new sequence of weights. We implement this within the CPLEX interface of MaxHS by using `CPXXchgcoef`² to change the coefficient of the objective function corresponding to the weight of a blocking variable. This update is performed only if the corresponding blocking variable exists in CPLEX.

The resulting procedure for `changeWeight(i, w)` is detailed as Algorithm 3. We compute δ as the difference between the new weight w and the current weight stored in `originalWeights[i]` (line 2). If `currentIndex[i]` is `UNSAT`, it suffices to increment `baseCost` by δ (lines 3 and 4). If `currentIndex[i]` is `SAT`, we simply do nothing (lines 5 and 6). Otherwise `currentIndex[i]` contains the index of the corresponding internal soft clause. Now, if this internal soft clause has a

² <https://www.ibm.com/docs/en/icos/12.10.0?topic=c-cpxchgcoef-cpxchgcoef>

contradictory unit clause, we resolve these two unit soft clauses (lines 8 and 9). Otherwise, we increment `totalWeight` and the internal weight `weights[currentIndex[i]]` by δ (lines 11 and 12). Finally, we set `originalWeights[i]` to the new weight w (line 13) and perform the necessary updates to CPLEX (line 14).

4.3.2 Weight-based Reasoning

In addition to correctly taking into account the simplification procedure and updating the IP solver, during solving MaxHS performs weight-based reasoning, which either has to be disabled or reimplemented by taking into account that weights may potentially change. These reasoning procedures are the following.

Reduced Cost Fixing

MaxHS considers the linear programming (LP) relaxation of the hitting set problem, and using so-called reduced costs corresponding to the optimal solution of the LP, determines whether a soft clause can be hardened [3]. In particular, this is determined via the optimal cost of the LP, the reduced cost corresponding to the blocking variable, and the cost of a feasible solution to the MaxSAT problem. After changing weights, all of these numbers may change arbitrarily. Hence, it is clear that soft clauses hardened due to reduced cost fixing may invalidate the current instance and alter the set of cores the instance (recall the earlier discussion on hardening). Due to this, reduced cost fixing is disabled.

Abstract Cores: Graph and Totalizers

Recall that in order to determine which blocking variables occur in cores often together, MaxHS constructs a weighted undirected graph based on the accumulated cores [8]. Nodes of this graph correspond to partitions of the set of blocking variables, and weights of the edges between nodes to how many times the blocking variables occur together in a core. In particular, it is assumed that blocking variables within a node and in adjacent nodes have the same weight. In order to preserve these invariants, if a node contains several blocking variables which now have different weights, the node is removed from the graph. Similarly, if the incident nodes of an edge contain blocking variables with different weights, the edge is removed from the graph.

In order to encode cardinality constraints over count variables corresponding to abstract cores, MaxHS makes use of totalizers [7]. It is assumed that blocking variables used as inputs of a totalizer have the same weight. Furthermore, each totalizer is assigned this weight in order to compute new lower bounds. In order to preserve this invariant, we check which totalizers contain inputs whose weights have changed. If all weights have changed to the same new weight, we simply reset the current weight of the totalizer to the new weight. If weights are different, the totalizer is invalid, so it is removed, and so are all totalizers containing a subset of the inputs of the totalizer.

4.3.3 Solving Procedure

With all of this in place, for solving the instance at iteration $i > 1$ we recompute the sum of the weights of soft clauses known to be satisfiable from the existing model in the SAT solver (τ_{best} in Algorithm 2) – this weight is used to determine the upper bound by subtracting it from `totalWeight`. Furthermore, we recompute the sum of the weights of blocking variables that are fixed to true by the SAT solver, and set the lower bound to this number. After reinitializing the graph and totalizers related to abstract cores, we may solve the updated instance as usual.

In addition to the techniques discussed in this section, our implementation of IHS-INC also makes use of a number of previously proposed heuristics for extracting a large number (hundreds) of cores from each hitting set [12, 10, 28]. The techniques that have not been discussed in this section are all sound to keep between iterations. Recall that, as long as a core extraction heuristic computes a set of blocking variables that is a core of the current instance, the same set will be a core in subsequent iterations as well.

5 Experimental Evaluation

In this section we provide an empirical evaluation comparing MaxHS (MaxSAT evaluation 2020 version³) to our implementation of IHS-INC built on top of MaxHS. The non-incremental MaxHS is run with default parameters, with all its optimizations including hardening of soft clauses during simplification and in the form of reduced cost fixing enabled.

All experiments were run on 2.60-GHz Intel Xeon E5-2670 8-core machines with 64GB memory and CentOS 7. We set a per-instance time limit of 7200 seconds (2 hours) and a memory limit of 16 GB. Specifically, the 7200-second time limit is for solving a single instance n times with n different weights w_1, \dots, w_n . For both the incremental and non-incremental solver, we record the solving time t_k of each iteration k . The k th iteration (as well as all subsequent ones) is considered as a timeout if $\sum_{i=1}^k t_i > 7200$ seconds.

We consider two different recently-proposed methods for learning interpretable classifiers, namely decision trees and decision rules, via MaxSAT. The input for both scenarios is a dataset (\mathbf{X}_i, y_i) , $i = 1, \dots, n$, of binary examples $\mathbf{X}_i \in \{0, 1\}^m$ and classes $y_i \in \{0, 1\}$. Each coordinate $j = 1, \dots, m$ of an example $\mathbf{X}_i = (x_i^1, \dots, x_i^m)$ is called a feature. The goal is to learn a binary classifier (a function mapping each example in the feature space $\{0, 1\}^m$ to a class in $\{0, 1\}$) which minimizes the training error consisting of the number of misclassified examples in the input dataset. In the context of changing weights, we consider two different methods designed to avoid overfitting. For decision trees, AdaBoost [16] is an algorithm where a sequence of shallow decision trees are learned by iteratively changing the weights of the examples in the training set. For decision rules, we include a regularizing term to the objective function which also enforces the sparsity of the resulting rule [18], and iteratively vary the value of the regularization parameter.

5.1 Case Study 1: MaxSAT for Boosting Decision Trees

Decision trees are classifiers with the structure of a full binary tree for binarized data. Leaf nodes are associated with a particular class (in our setting, 0 or 1), and non-leaf nodes with a feature $j = 1, \dots, m$. An example $\mathbf{X} = (x_1, \dots, x_m)$ is classified by starting from the root node, checking the value of x_j for the feature j associated to the node, and proceeding recursively to the left child if $x_j = 0$ and to the right child if $x_j = 1$. The class is then determined by the leaf node which terminates the recursion.

We consider the MaxSAT encoding for learning a decision tree of depth at most U [16], based on a SAT encoding for learning a decision tree with exactly N nodes [26]. In addition to variables and hard clauses for encoding the structure of a valid binary tree, its depth, and the classification of the training data, the MaxSAT encoding has variables b_i for each example \mathbf{X}_i with the interpretation that b_i is true if and only if example \mathbf{X}_i is classified correctly. The objective is then to minimize the training error via unit-weight soft clauses (b_i) for each example \mathbf{X}_i . An instance formed from a dataset with n examples and m features has $O(n + m)$ variables, $O(n + m)$ hard clauses of length $O(m)$, and n unit soft clauses.

³ <https://maxsat-evaluations.github.io/2020/descriptions.html>

■ **Table 1** Statistics on MaxSAT instances used for AdaBoost.

	Minimum	Maximum	Average	Median
Number of variables	2129	26396	8310.4	7768
Number of hard clauses	8176	96382	30343.8	26772
Literals in hard clauses	38524	5342964	718654.9	211629
Average length of hard clauses	3.45476	70.1593	16.4	10.3187
Number of soft clauses	27	6473	803.6	342

In particular, here we focus on the implementation of AdaBoost [14], an ensemble method where multiple weak classifiers (in this context, shallow decision trees) are trained and then combined into a single classifier via a weighted voting scheme. This is achieved via changing the weights of the soft clauses iteratively [16]. In more detail, after learning a decision tree, the weight $w(b_i)$ for each $i = 1, \dots, n$ is updated via

$$\hat{w}(b_i) = \frac{w(b_i) \cdot f_i}{\sum_{j=1}^n w(b_j) \cdot f_j}$$

where $f_i = \exp(-\alpha)$ if the i th example was classified correctly and $\exp(\alpha)$ otherwise, $\alpha = \frac{1}{2} \ln(\frac{1-\varepsilon}{\varepsilon})$, and ε is the training error. As long as $\varepsilon < 0.5$, this raises the weight of incorrectly classified examples and lowers the weight of correctly classified examples. Intuitively, the following decision tree will consider that misclassified examples are more important than correctly classified ones. Then, weights are discretized by setting

$$w(b_i) = \text{round} \left(\frac{\hat{w}(b_i)}{\min_{j=1, \dots, n} \hat{w}(b_j)} \right).$$

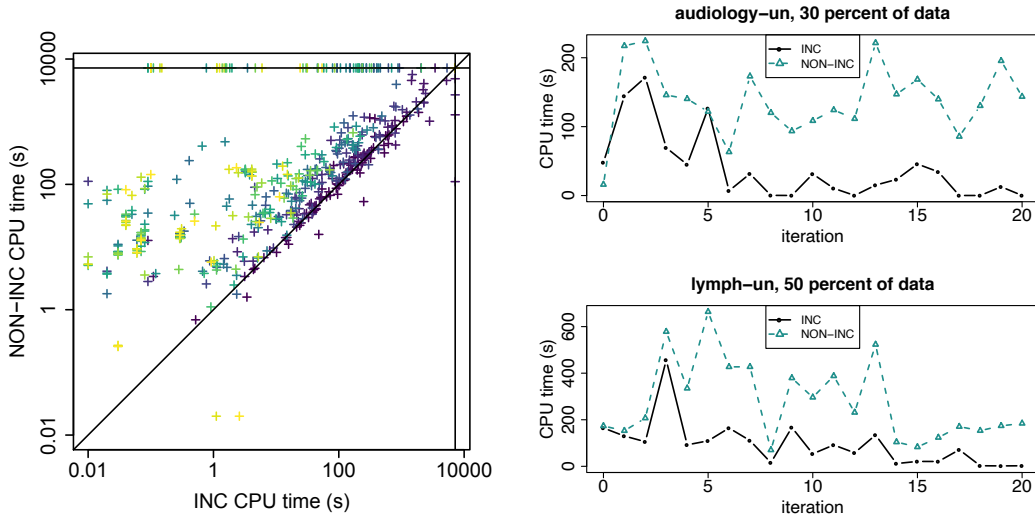
In other words, if an example is classified correctly at each iteration, its corresponding weight will remain constant 1 due to discretization. If an example is classified incorrectly at each iteration, its weight will grow exponentially.

In contrast to using an incomplete MaxSAT solver by starting it from scratch at each iteration [16], we consider solving each iteration exactly in an incremental fashion. For benchmarks, we take the 15 datasets used in [16] (which were downloaded from CP4IM⁴ and discretized⁵). For the exact number of examples and the number of features in these instances, we refer the reader to [16, Table 1]. For each dataset, we generated different training sets by taking 20%, 30%, ..., 80% of the available data, resulting in 105 training sets. We set the maximum depth to $U = 2$ and used 21 iterations for AdaBoost. Detailed statistics on the resulting MaxSAT instances are provided in Table 1.

The results are summarized in Figure 2, where each point is a single iteration, the x-axis is the CPU time-consumed by our implementation, and the y-axis is the CPU time-consumed by MaxHS. Points are colored by the iteration number: the higher the iteration number, the more yellow the point (and the lower, the more blue). We clearly see that almost all lower iterations take approximately the equal amount of time, with a few more timeouts exhibited by our implementation than by MaxHS (four points on the right border of the plot). However, for higher iterations, we see a clear improvement from using the incremental version

⁴ <https://dtai.cs.kuleuven.be/CP4IM/datasets/>

⁵ <https://gepgitlab.laas.fr/hhu/maxsat-decision-trees>



■ **Figure 2** Incremental vs. non-incremental MaxHS for AdaBoost. Each point is an iteration. ■ **Figure 3** Incremental vs. non-incremental MaxHS for AdaBoost on example datasets.

of MaxHS. In particular, some iterations which take 100 seconds to solve using MaxHS are now solved in a matter of seconds, and MaxHS exhibits a significant number of timeouts which are solved using the incremental version (points on the upper border of the plot).⁶

Dataset specific examples of how the runtimes of non-incremental and incremental MaxHS differ when iterating over the sequences of instances the datasets give rise to are provided in Figure 3. We observe that for almost all iterations, the performance of the incremental version is significantly better than that of the standard non-incremental MaxHS. This is the case in particular for the later iteration; evidently, incremental computations start paying off noticeably after solving the first few instances in the sequence.

5.2 Case Study 2: MaxSAT for Learning Decision Rules

Decision rules are classifiers which take the simple and interpretable form of if-then-else rules. Here we consider MLIC [18], a framework for learning decision rules via MaxSAT, in particular decision rules where the implicant is a CNF formula \mathcal{R} over the features containing exactly K clauses, and the consequent is simply “class is 1”. In addition to minimizing the training error, the goal is to learn *sparse* decision rules. Sparsity of the learned rules is enforced by a regularizing term. In particular, the objective is to minimize $\lambda|\mathcal{E}_{\mathcal{R}}| + \|\mathcal{R}\|$, where $|\mathcal{E}_{\mathcal{R}}|$ is the number of misclassified examples and $\|\mathcal{R}\|$ is the number of literals in \mathcal{R} . The choice of the regularization parameter⁷ $\lambda > 0$ is a difficult task. A simple method for choosing λ is to perform an exhaustive grid search over an interval and choosing the λ that minimizes e.g. the cross-validated error. Note the form of the objective function, namely minimizing the linear combination of an error and a regularizing term, is very general, and interestingly similar MaxSAT-based methods for learning sparse decision sets [32] and lists [31] also share a similar objective.

⁶ We also tried using the previous optimal solution to calculate an initial UB in non-incremental MaxHS, but did not observe any significant performance improvements. Note that MaxHS only terminates when the upper bound equals the lower bound, even given an optimal solution, MaxHS has to prove its optimality by extracting cores which yield a hitting set of the same cost.

⁷ Note that, unlike is typically the case, the role of λ in [18] is the weight of the error term, not the sparsity term (which is in fact the regularizer). The sparsity term has coefficient $1/\lambda$.

■ **Table 2** Data on MaxSAT instances used for MLIC.

	Minimum	Maximum	Average	Median
Number of variables	182	183105	26122.9	2339.5
Number of hard clauses	523	49086727	3792656.9	47721
Literals in hard clauses	1473	120037762	9765213.2	216129
Average length of hard clauses	2.21576	132	15.8	2.5
Number of soft clauses	157	48139	8224.0	1627.5

The MaxSAT encoding has variables b_k^j for each clause index $k = 1, \dots, K$ and each feature $j = 1, \dots, m$, and variables η_i for each example \mathbf{X}_i . Here b_k^j is assigned to true if and only if feature j occurs in the k th clause, and η_i is assigned to true if and only if example \mathbf{X}_i is classified incorrectly. In addition to hard clauses encoding the semantics, soft clauses $(\neg\eta_i)$ with unit weights and $(\neg b_k^j)$ with weight λ are used to encode the objective function. An instance resulting from encoding a dataset with n examples and m features has $O(n + m)$ variables, $O(n + m)$ hard clauses with length $O(m)$, and $O(n + m)$ unit soft clauses.

Following [18], we consider computing the optimal decision rules for $\lambda = 0.25, 0.5, \dots, 5.0$, in this exact order. As we start from a low value of λ , the iterative procedure first learns decision rules that are more sparse and less attention is given to correct classification, and as λ is incremented, more importance is given to classifying the examples correctly than to sparsity. We use the same 10 datasets (from the UCI repository⁸) which were discretized via adapting the script provided by the MLIC repository⁹. For the exact number of examples and the number of features, we refer the reader to [18, Table 1]. For each dataset, we generated training sets by taking 10%, 20%, \dots , 90% of the available data, resulting in 90 different training sets. We learned CNF rules consisting of $K = 2, 3$ clauses (as instances with $K = 1$ clauses were observed to be solved directly using the IP solver due to all constraints being seeded into CPLEX). This gave rise to 180 different runs each with 20 iterations. Detailed statistics on the MaxSAT instances are provided in Table 2.

Our results are summarized in Figure 4, where each point corresponds to a single iteration, the x-axis is the CPU time of the incremental version, the y-axis is the CPU time of basic MaxHS, and points are colored by the iteration number. We observe a very clear improvement in favor of the incremental version, especially for higher iterations. MaxHS also exhibits a significant number of timeouts for iterations that are solved using the incremental version.

Dataset specific examples of how the runtimes of non-incremental and incremental MaxHS differ when iterating over the sequences of instances the datasets give rise to are provided in Figure 5, with the value of lambda (corresponding to the iteration) on the x-axis and the CPU time on the y-axis. While we observe some variation in the runtime for both solvers, e.g. the iterations corresponding to $\lambda = 1.25, 2.25$, are slower to solve, the incremental version is clearly faster on most iterations. Some instances are significantly easier, essentially trivial, to solve using the incremental version compared to the non-incremental solver; the bottom plot in Figure 5 provides such an example.

⁸ <https://archive.ics.uci.edu/ml/>

⁹ <https://github.com/meelgroup/MLIC/tree/MLIC>

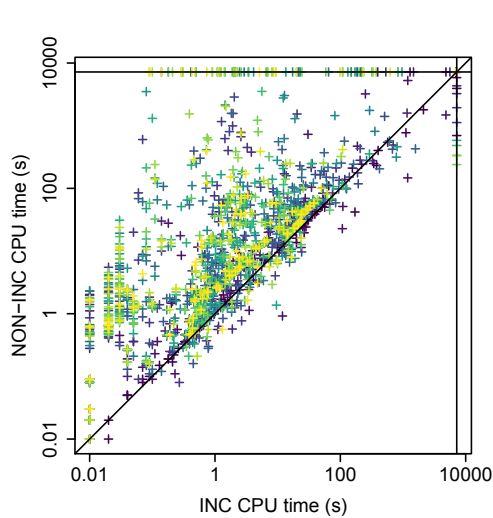


Figure 4 Incremental vs. non-incremental MaxHS for MLIC. Each point is an iteration.

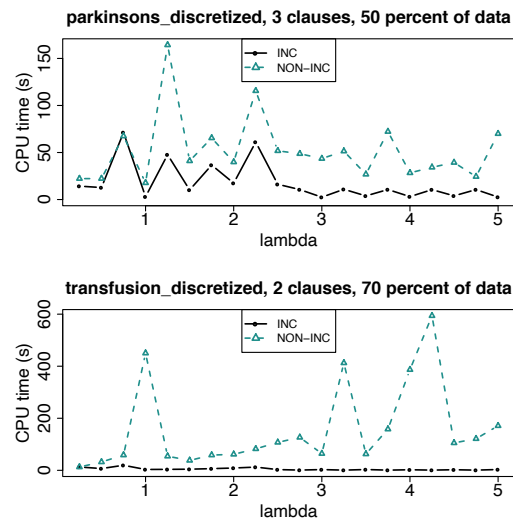


Figure 5 Incremental vs. non-incremental MaxHS for MLIC on example datasets.

6 Conclusions

Various types of real-world optimization problems, requiring solving a sequence of related problem instances, call for solvers that can make use of incremental computations across the instances. Motivated by recent applications of MaxSAT solvers, we adapted one of the key MaxSAT solving approaches – the implicit hitting set approach – to cope with incrementality under changes to the weights of soft clauses. While it is seemingly simple to adapt a rudimentary version of the IHS approach to deal with this type of incrementality, the various search techniques applied in MaxHS, a state-of-the-art IHS MaxSAT solver, make such adaptations non-trivial. In particular, we explained which search techniques can and cannot be adapted for incremental computations under changing weights. Taking these observations into practice, we adapted MaxHS to support incrementality under changing soft clause weights. Using two recent real-world applications of MaxSAT in the context of interpretable machine learning as examples, we showed that the incremental version of MaxHS provides significant runtime improvements over MaxHS (despite all of the performance-improving optimizations used in the non-incremental version) when solving sequences of MaxSAT instances with adaptively changing weights. As future work, we aim to generalize the framework further to allow e.g. efficiently altering the set of hard and soft clauses between iterations without increasing the sizes of extracted cores or hitting set instances met.

References

- 1 Mario Alviano, Carmine Dodaro, and Francesco Ricca. A MaxSAT algorithm using cardinality constraints of bounded size. 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 2677–2683. AAAI Press, 2015. URL: <http://ijcai.org/Abstract/15/379>.
- 2 Carlos Ansótegui, Frédéric Didier, and Joel Gabàs. Exploiting the structure of unsatisfiable cores in MaxSAT. 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 283–289. AAAI Press, 2015. URL: <http://ijcai.org/Abstract/15/046>.

- 3 Fahiem Bacchus, Antti Hyttinen, Matti Järvisalo, and Paul Saikko. Reduced cost fixing in MaxSAT. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming – 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 – September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 641–651. Springer, 2017. doi:10.1007/978-3-319-66158-2_41.
- 4 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. MaxSAT Evaluation 2018: New developments and detailed results. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):99–131, 2019. doi:10.3233/SAT190119.
- 5 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins, editors. *MaxSAT Evaluation 2019: Solver and Benchmark Descriptions*, volume B-2019-2 of *Department of Computer Science Report Series B*. Department of Computer Science, University of Helsinki, Finland, 2019. URL: <https://hdl.handle.net/10138/308068>.
- 6 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability, Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 24, pages 929–991. IOS Press, 2021. doi:10.3233/FAIA201008.
- 7 Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003 – 9th International Conference, CP 2003, Kinsale, Ireland, September 29 – October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003. doi:10.1007/978-3-540-45193-8_8.
- 8 Jeremias Berg, Fahiem Bacchus, and Alex Poole. Abstract cores in implicit hitting set MaxSat solving. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020 – 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 277–294. Springer, 2020. doi:10.1007/978-3-030-51825-7_20.
- 9 Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. A modular approach to MaxSAT modulo theories. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing – SAT 2013 – 16th International Conference, Helsinki, Finland, July 8-12, 2013, Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 150–165. Springer, 2013. doi:10.1007/978-3-642-39071-5_12.
- 10 Jessica Davies. *Solving MAXSAT by Decoupling Optimization and Satisfaction*. PhD thesis, University of Toronto, 2013. URL: <https://hdl.handle.net/1807/43539>.
- 11 Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In Jimmy Ho-Man Lee, editor, *Principles and Practice of Constraint Programming – CP 2011 – 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011, Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2011. doi:10.1007/978-3-642-23786-7_19.
- 12 Jessica Davies and Fahiem Bacchus. Postponing optimization to speed up MAXSAT solving. In Christian Schulte, editor, *Principles and Practice of Constraint Programming – 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013, Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2013. doi:10.1007/978-3-642-40627-0_21.
- 13 Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003. doi:10.1016/S1571-0661(05)82542-3.
- 14 Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997. doi:10.1006/jcss.1997.1504.
- 15 Bishwamittra Ghosh and Kuldeep S. Meel. IMLI: an incremental framework for MaxSAT-based learning of interpretable classification rules. In Vincent Conitzer, Gillian K. Hadfield, and Shannon Vallor, editors, *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society, AIES 2019, Honolulu, HI, USA, January 27-28, 2019*, pages 203–210. ACM, 2019. doi:10.1145/3306618.3314283.

- 16 Hao Hu, Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. Learning optimal decision trees with MaxSAT and its integration in AdaBoost. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 1170–1176. ijcai.org, 2020. doi:10.24963/ijcai.2020/163.
- 17 Alexey Ignatiev, António Morgado, and João Marques-Silva. RC2: an efficient MaxSAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):53–64, 2019. doi:10.3233/SAT190116.
- 18 Dmitry Malioutov and Kuldeep S. Meel. MLIC: A MaxSAT-based framework for learning interpretable classification rules. 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 312–327. Springer, 2018. doi:10.1007/978-3-319-98334-9_21.
- 19 Ravi Mangal, Xin Zhang, Aditya Kamath, Aditya V. Nori, and Mayur Naik. Scaling relational inference using proofs and refutations. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 3278–3286. AAAI Press, 2016. URL: <http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12466>.
- 20 Ruben Martins, Saurabh Joshi, Vasco M. Manquinho, and Inês Lynce. Incremental cardinality constraints for MaxSAT. In Barry O’Sullivan, editor, *Principles and Practice of Constraint Programming – 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014, Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer, 2014. doi:10.1007/978-3-319-10428-7_39.
- 21 Ruben Martins, Saurabh Joshi, Vasco M. Manquinho, and Inês Lynce. On using incremental encodings in unsatisfiability-based MaxSAT solving. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1):59–81, 2014. doi:10.3233/sat190102.
- 22 Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014 – 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014, Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer, 2014. doi:10.1007/978-3-319-09284-3_33.
- 23 António Morgado, Carmine Dodaro, and João Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In Barry O’Sullivan, editor, *Principles and Practice of Constraint Programming – 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014, Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer, 2014. doi:10.1007/978-3-319-10428-7_41.
- 24 António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013. doi:10.1007/s10601-013-9146-2.
- 25 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: <http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8513>.
- 26 Nina Narodytska, Alexey Ignatiev, Filipe Pereira, and João Marques-Silva. Learning optimal decision trees with SAT. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1362–1368. ijcai.org, 2018. doi:10.24963/ijcai.2018/189.
- 27 Matthew Richardson and Pedro M. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006. doi:10.1007/s10994-006-5833-1.
- 28 Paul Saikko. Re-implementing and extending a hybrid SAT-IP approach to maximum satisfiability. Master’s thesis, University of Helsinki, 2015. URL: <http://hdl.handle.net/10138/159186>.

- 29 Paul Saikko, Jeremias Berg, and Matti Järvisalo. LMHS: A SAT-IP hybrid MaxSAT solver. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016 – 19th International Conference, Bordeaux, France, July 5–8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 539–546. Springer, 2016. doi:10.1007/978-3-319-40970-2_34.
- 30 Xujie Si, Xin Zhang, Vasco M. Manquinho, Mikolás Janota, Alexey Ignatiev, and Mayur Naik. On incremental core-guided MaxSAT solving. 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 473–482. Springer, 2016. doi:10.1007/978-3-319-44953-1_30.
- 31 Jinqiang Yu, Alexey Ignatiev, Pierre Le Bodic, and Peter J. Stuckey. Optimal decision lists using SAT. *CoRR*, abs/2010.09919, 2020. arXiv:2010.09919.
- 32 Jinqiang Yu, Alexey Ignatiev, Peter J. Stuckey, and Pierre Le Bodic. Computing optimal decision sets with SAT. 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 952–970. Springer, 2020. doi:10.1007/978-3-030-58475-7_55.
- 33 Xin Zhang, Ravi Mangal, Aditya V. Nori, and Mayur Naik. Query-guided maximum satisfiability. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 – 22, 2016*, pages 109–122. ACM, 2016. doi:10.1145/2837614.2837658.

Solving the Non-Crossing MAPF with CP

Xiao Peng

CITI, INRIA, INSA Lyon, F-69621, Villeurbanne, France

Christine Solnon

CITI, INRIA, INSA Lyon, F-69621, Villeurbanne, France

Olivier Simonin

CITI, INRIA, INSA Lyon, F-69621, Villeurbanne, France

Abstract

We introduce a new Multi-Agent Path Finding (MAPF) problem which is motivated by an industrial application. Given a fleet of robots that move on a workspace that may contain static obstacles, we must find paths from their current positions to a set of destinations, and the goal is to minimise the length of the longest path. The originality of our problem comes from the fact that each robot is attached with a cable to an anchor point, and that robots are not able to cross these cables.

We formally define the Non-Crossing MAPF (NC-MAPF) problem and show how to compute lower and upper bounds by solving well known assignment problems. We introduce a Variable Neighbourhood Search (VNS) approach for improving the upper bound, and a Constraint Programming (CP) model for solving the problem to optimality. We experimentally evaluate these approaches on randomly generated instances.

2012 ACM Subject Classification Computing methodologies

Keywords and phrases Constraint Programming (CP), Multi-Agent Path Finding (MAPF), Assignment Problems

Digital Object Identifier 10.4230/LIPIcs.CP.2021.45

Funding This work was supported by the European Commission under the H2020 project BugWright2 (871260): Autonomous Robotic Inspection and Maintenance on Ship Hulls and Storage Tanks.

1 Introduction

Multi-agent path finding (MAPF) is a very active research topic which has important applications for robotics in industrial contexts (e.g., transport in fulfillment centers, autonomous tug robots). In this paper we consider an extension of MAPF for tethered robots, i.e., robots attached with flexible cables to anchor points, allowing them to have continuous access to fluids such as energy or water, for example. This is the case for our industrial partner in a European project¹ where a fleet of mobile robots is used for inspecting and cleaning large structures. Each robot has a cable which is kept taut between its anchor point and its current position by a system that pulls on the cable when the robot moves back. The main difficulty with these tethered robots comes from the fact that robots are not able to cross cables. Hence, this paper introduces the Non-Crossing MAPF (NC-MAPF) problem which aims at finding paths such that robots never have to cross cables.

¹ H2020 project BugWright2: Autonomous Robotic Inspection and Maintenance on Ship Hulls and Storage Tanks, 2020-24



© Xiao Peng, Christine Solnon, and Olivier Simonin;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 45; pp. 45:1–45:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Related work

In classic MAPF, agents move in a discretized environment (a grid or a graph). The goal is to find a plan for moving all agents from their initial locations to target locations so that no two agents share a same location (grid cell, graph node, or graph edge) at a same moment. Typically, a plan is a sequence of actions for each agent, where an action is either “move to an adjacent location” or “wait at the current location”.

There are two main MAPF variants depending on whether each agent has a known target, or there is a set of targets and each agent must be first assigned to a target before searching for a plan. This latter variant, called *anonymous MAPF*, is more general and also more difficult as the search space is increased. There are two main objective functions, i.e., minimise the *makespan*, corresponding to the latest arrival time of an agent to its target, or minimise the sum of all travel times. In both cases, the problem is \mathcal{NP} -hard [18].

MAPF problems are usually solved by using *Conflict Based Search (CBS)* approaches [15] which are two-level approaches: at the low level, paths are searched (while satisfying constraints added at the high level); and at the high level, path conflicts are resolved. CBS has been extended to agents with a specific geometric shape and volume (e.g., [9, 17]) and to convoys (agents that occupy a sequence of nodes and their connecting edges) [16]. These MAPF variants share some similarities with NC-MAPF as a tethered robot may be viewed as a robot which has a very long body corresponding to its cable.

However, CBS is not suited to solve NC-MAPF because this approach is efficient when conflicts are easily resolved by applying small changes to paths (e.g., waiting for a location to be freed or getting around an occupied location). This is not the case for NC-MAPF. For example, let us consider the case of two paths π_1 (from an anchor point a_1 to a target t_1) and π_2 (from a_2 to t_2) such that the cables cross at some point x . To solve this conflict, a first possibility is to ask the first robot to wait just before reaching x while the second robot continues its path from x to d_2 , achieves its task on d_2 , and returns back to x , thus removing the cable from x and allowing the first robot to continue its path from x to d_1 . As robots usually have to achieve long duration tasks, this way of resolving conflicts dramatically increases the makespan. A second possibility is to search for new paths such that cables do not cross, but this cannot be done by applying small changes to the paths and this problem may have no solution in some cases.

In the robotics literature, few works have investigated path planning for tethered robots. In most cases, cables may be pushed and bent by robots (e.g., [6, 19]), which is not possible in our industrial context. As far as we know, none has considered a case similar to our problem where (i) robots cannot cross neither push or bent cables, (ii) paths cannot be sequentialized (i.e., a robot cannot wait for another robot to have achieved its task and returned back to its anchor point), and (iii) robots do not have assigned targets (anonymous MAPF).

Contributions and outline of the paper

In Section 2, we introduce notations and define the workspace on which robots evolve. This workspace is continuous, and we show in Section 3 how to reformulate our problem in a discrete visibility graph.

In Section 4, we first consider the case where the workspace has no obstacle. We show that the NC-MAPF problem without obstacle is a special kind of assignment problem in a bipartite graph, and we show how to efficiently compute lower and upper bounds by solving well known assignment problems. We also introduce a *Variable Neighbourhood Search (VNS)* approach, to improve the upper bound, and a *Constraint Programming (CP)* model, to compute the optimal solution.

In Section 5, we consider the case where the workspace has obstacles. We prove that optimal solutions of assignment problems still provide bounds in this case. We also show that the optimal solution of the NC-MAPF problem may contain some paths that are not shortest paths. Hence, we introduce an approach for enumerating all relevant paths and, finally, we introduce a CP model for computing the optimal solution.

In Sections 4 and 5, we report experimental results on randomly generated instances and show that our approach scales well enough to solve realistic instances within a few seconds.

2 Definition of the workspace and notations

Robots move on a 2 dimensional workspace $W \subset \mathbb{R}^2$. This workspace is defined by a bounding polygon B and a set O of obstacles: every obstacle in O is a polygon within B , and W is composed of every point in B that does not belong to an obstacle in O . Without loss of generality, we assume that B is convex: if the bounding polygon is not convex, then we can compute its convex hull B and add to O the obstacle corresponding to the difference between the bounding polygon and B . We denote V_O the set of vertices of obstacles in O , and we assume that these vertices belong to W (and, therefore, obstacle boundaries belong to W).

Given two points $u, v \in W$, we denote \overline{uv} the straight line segment that joins u to v , and $|uv|$ the Euclidean distance between u and v (i.e., $|uv|$ is the length of \overline{uv}). We say that a segment crosses an obstacle if $\overline{uv} \not\subset W$. Given two segments \overline{uv} and $\overline{u'v'}$, we say that they are incident if they have one common endpoint (i.e., $|\{u, v\} \cap \{u', v'\}| = 1$), and we say that they cross if they share one point (called the crossing point) which is not an endpoint (i.e., $\{u, v\} \cap \{u', v'\} = \emptyset$ and $\overline{uv} \cap \overline{u'v'} \neq \emptyset$).

A chain of incident segments $\overline{u_0u_1}, \overline{u_1u_2}, \dots, \overline{u_{i-1}u_i}$ is represented by the sequence $\pi = \langle u_0, u_1, u_2, \dots, u_i \rangle$. The length of this chain of segments is denoted $|\pi|$ and is the sum of the lengths of its segments, i.e., $|\pi| = \sum_{j=1}^i |u_{j-1}u_j|$.

We denote $[x, y]$ the set of all integer values ranging between x and y .

3 Definition of the NC-MAPF Problem

We consider an anonymous MAPF problem with a set of n robots such that each robot is attached with a flexible cable to an anchor point in W , and a set of n destinations. The goal is to find a path in W for each robot from its anchor point to a different destination so that the longest path is minimised and robots never have to cross cables.

As the workspace W is continuous, there exists an infinite number of paths from an anchor point a to a destination d . However, as each cable is kept taut, the number of different cable positions that start from a and end on d is finite (provided that we forbid infinite loops). More precisely, the cable position associated with a robot path from a to d is a chain of incident segments $\langle u_0, u_1, \dots, u_i \rangle$ such that (i) $u_0 = a$ and $u_i = d$, (ii) no segment crosses an obstacle, and (iii) every internal point is an obstacle vertex, i.e., $\forall j \in [1, i-1], u_j \in V_O$.

As the length of a robot path cannot be smaller than the length of its cable position, we can simplify our problem by assuming that the path of a robot is its cable position. Hence, we search for paths in a visibility graph [10] defined in Def. 1 and illustrated in Fig. 1.

► **Definition 1** (Visibility graph [10]). *The visibility graph associated with a workspace W , a set of anchor points A and a set of destinations D is the directed graph $G = (V, E)$ such that vertices are either points of A and D or obstacle vertices, i.e., $V = A \cup D \cup V_O$, and edges correspond to segments that do not cross obstacles, i.e., $E = \{(u, v) \in (A \cup V_O) \times (D \cup V_O) \mid \overline{uv} \subset W\}$. The graph is directed because edges from destinations to anchor points are forbidden.*

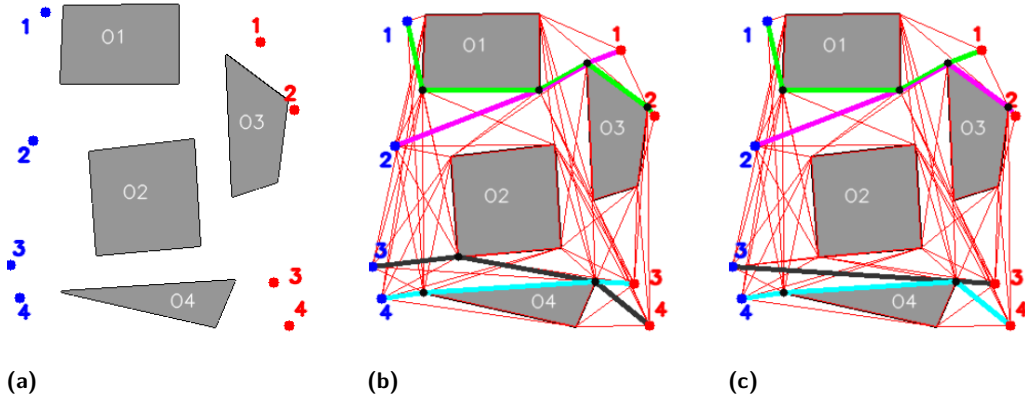
In Def. 1, we implicitly assume that robots are points, which is an acceptable approximation when the actual size of robots is very small compared to the size of obstacles (which is the case in our industrial application). This definition may be extended to the case where robot shapes are approximated by circles with non null radius in a straightforward way by growing obstacles (see [10] for details).

A path in the visibility graph G is a sequence of vertices $\langle u_0, \dots, u_i \rangle$ such that $(u_{j-1}, u_j) \in E, \forall j \in [1, i]$. This path also corresponds to a chain of segments and its length is the sum of the lengths of its segments. We only consider elementary paths, i.e., a vertex cannot occur more than once in a path. Indeed, if a path is not elementary, then it can be replaced by a shorter elementary path obtained by removing its cycles.

Two paths are homotopic if there exists a continuous deformation between them without crossing obstacles [2], and a taut path is the shortest path of a homotopy class. For example, in the workspace of Fig. 1, all paths starting from the anchor point a_1 (point 1 in blue), passing between O_1 and O_2 and then between O_1 and O_3 , and finally reaching the destination d_1 (point 1 in red) are homotopic. Let x be the bottom-left vertex of O_1 , y its bottom-right vertex, z the top-left vertex of O_3 , and t the top-right vertex of O_2 . The paths $\pi = \langle a_1, x, y, z, d_1 \rangle$ and $\pi' = \langle a_1, x, t, z, d_1 \rangle$ are homotopic. π is taut because it is the shortest path of its homotopy class. π' is not taut because it is longer than π .

We say that a path is self-crossing if it contains two crossing segments. We say that two paths π and π' are crossing either if π contains a segment that crosses a segment of π' , or if π contains two incident segments \overline{uv} and \overline{vw} and π' contains two incident segments $\overline{u'v'}$ and $\overline{v'w'}$ such that $v = v'$ and \overline{uv} crosses $\overline{u'w'}$. However, two non crossing paths may share some vertices or some segments, as illustrated in Fig. 1(c). Indeed, as robots are small and cables are thin, a robot can slightly push the cable of another robot without crossing its cable. For example, if the black robot (starting from 3) in Fig. 1(c) arrives on the vertex of obstacle O_4 before the blue robot (starting from 4) then, when the blue robot arrives on this vertex, it can slightly push the black cable to continue its path between O_4 and the black cable.

Let us now formally define our problem.



■ **Figure 1** (a): Example of workspace W with four anchor points (in blue) and four destinations (in red). (b): Visibility graph with paths that are not solution of the NC-MAPF because the green path crosses the pink path and the black path crosses the blue path. Besides, the black path is not taut. (c): Visibility graph with paths that are solution of the NC-MAPF, even though the green and pink paths share a segment, and the black and blue paths share a vertex.

► **Definition 2** (NC-MAPF problem). *Given a workspace W , a set A of n anchor points and a set D of n destinations such that every point in $A \cup D$ belongs to W , the goal of the NC-MAPF problem is to find n paths in the visibility graph G associated with W , A , and D such that (i) every path is taut, (ii) every path starts on a different anchor point of A , (iii) every path ends on a different destination of D , (iv) no path is self-crossing, (v) no two paths are crossing, and (vi) the length of the longest path is minimal.*

4 NC-MAPF problem without obstacles

In this section, we consider the case where the set O of obstacles is empty. In this case, $V_O = \emptyset$ and the visibility graph G is the complete bipartite graph such that $V = A \cup D$ and $E = A \times D$ (every edge of E is included in W as the bounding polygon is convex).

In Section 4.1, we show how to compute lower and upper bounds by solving well known assignment problems. In Section 4.2, we show how to improve the upper bound by performing variable neighbourhood search. In Section 4.3, we introduce a CP model and, in Section 4.4, we experimentally evaluate these approaches.

4.1 Computation of bounds by solving assignment problems

An assignment problem aims at finding a one-to-one matching between tasks and agents [3, 13]. In our context, tasks correspond to destinations and agents to robots, and a matching is a bijection $m : A \rightarrow D$. We say that an edge (a, d) of the visibility graph G is selected whenever $m(a) = d$. The NC-MAPF problem without obstacles is a special case of assignment problem:

- there is an additional constraint that ensures that no two selected edges cross, i.e., $\forall \{a_i, a_j\} \subseteq A, \overline{a_i m(a_i)} \cap \overline{a_j m(a_j)} = \emptyset$;
- there is an objective function that aims at minimising the maximal cost of a selected edge, i.e., $\max_{a_i \in A} |a_i m(a_i)|$.

There exists many other assignment problems [3, 13]. The most well known one is the *Linear Sum Assignment Problem (LSAP)* that aims at minimising the sum of the costs of the selected edges. The LSAP can be solved in polynomial time (e.g., by the Hungarian algorithm [7]). Interestingly, the solution of the LSAP cannot have crossing edges whenever edge costs are defined by Euclidean distances [14]. Indeed, if two selected edges cross, then we can obtain a better assignment by swapping their destinations so that the two edges no longer cross. Hence, the solution of the LSAP provides an upper bound to the NC-MAPF problem without obstacles.

The assignment problem that aims at minimising the maximal cost of a selected edge is known as the *Linear Bottleneck Assignment Problem (LBAP)*, and this problem can also be solved in polynomial time (e.g., by adapting the Hungarian algorithm). However, when adding the constraint that the selected edges must not cross, the problem becomes \mathcal{NP} -hard [4]. Hence, the solution of the LBAP provides a lower bound to the NC-MAPF problem without obstacles.

4.2 Variable Neighbourhood Search

The upper bound computed by solving a LSAP may be tightened by performing local search. We consider a basic VNS framework [11] described below.

- The neighbourhood of a matching m contains every non crossing matching obtained by permuting the destinations of k anchor points, and it is explored in $\mathcal{O}\left(\binom{n-1}{k-1} \cdot k!\right)$: we first search for the longest edge $(a, m(a))$; then, we enumerate subsets of $A \setminus \{a\}$ that

contain $k - 1$ anchor points and, for each subset (to which a is added), we consider every permutation of the destinations without crossing edges, until finding a permutation whose longest edge is smaller than $(a, m(a))$.

- k is initialised to 2, and the search is started from the matching computed by solving the LSAP. We iteratively perform improving moves, by replacing the current matching with one of its neighbours that has a shorter longest edge. When we reach a locally optimal matching (that cannot be improved by permuting the destinations associated with k anchor points), we increase k . When an improving move is performed, k is reset to 2.
- The search is stopped either when a given time limit l is reached or when k becomes greater than a given upper bound k_{max} . (In the classical VNS framework, the current solution is perturbed and k is reset to its lowest possible value when k becomes greater than its upper bound k_{max} . We do not consider this perturbation phase here.)

4.3 Constraint Programming Model

Finally, let us introduce a CP model for the NC-MAPF problem without obstacles. Without loss of generality, we assume that all edge lengths have integer values: if this is not the case, then we can multiply every length by a given constant factor $c > 1$ and then round it to the closest integer value so that for each couple of edges $((u, v), (u', v'))$ such that $|uv| < |u'v'|$, we have $\text{round}(c * |uv|) < \text{round}(c * |u'v'|)$. In this case, the optimal solution of the integer problem is also an optimal solution of the original problem.

Let ub be an upper bound to the optimal solution. The variables are:

- an integer variable x_i is associated with every anchor point $a_i \in A$, and the domain of this variable contains every destination that is within a distance of ub from a_i , i.e., $D(x_i) = \{d \in D : |a_i d| < ub\}$;
- an integer variable y represents the maximal length of a selected edge.

The constraints are:

- for each pair of anchor points $\{a_i, a_j\} \subseteq A$, we post a table constraint $(x_i, x_j) \in T_{ij}$ where T_{ij} is the table that contains every couple $(d, d') \in D(x_i) \times D(x_j)$ such that $d \neq d'$ and the segment $\overline{a_i d}$ does not cross the segment $\overline{a_j d'}$;
- for each anchor point $a_i \in A$, we post the constraint $y \geq |a_i x_i|$;
- we post an *allDifferent* constraint on $\{x_i : a_i \in A\}$. This constraint is redundant as table constraints prevent assigning a same value to two different x_i variables. However, preliminary experiments have shown us that this improves the solution process for a wide majority of instances.

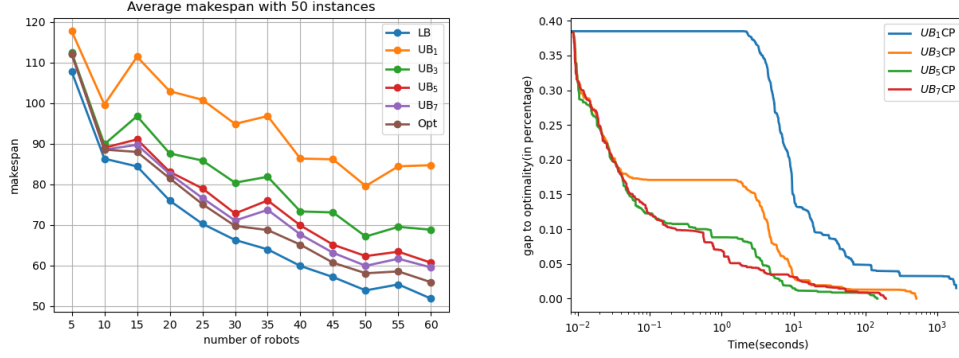
The goal is to minimise y .

4.4 Experimental evaluation

We evaluate our algorithms on randomly generated instances. For all instances, the bounding polygon is the square $B = [0, 200]^2$. To generate an instance with n robots, we randomly generate n anchor points and n destinations that all belong to B and such that the distance between two points is always larger than 4. For each value of n , we generate 50 different instances and report average results on these instances for all figures and tables.

We consider the following approaches:

- LB refers to the computation of a lower bound by solving an LBAP (see Section 4.1).
- UB_i with $i \in \{1, 3, 5, 7\}$ refers to the computation of an upper bound by first solving an LSAP (see Section 4.1) and then improving it by VNS with $l = 60$ seconds and $k_{max} = i$ (see Section 4.2). Note that when $i = 1$, VNS is immediately stopped as k is initialised to 2 and the search is stopped when k becomes greater than k_{max} .



■ **Figure 2** Left: Evolution of the optimal makespan (Opt), the lower bound (LB) and upper bounds (UB_{*i*} with $i \in \{1, 3, 5, 7\}$) when increasing the number n of robots. Right: Evolution of the gap to optimality (in percentage) with respect to time for UB_{*i*}CP with $i \in \{1, 3, 5, 7\}$, on average for the 50 instances with $n = 50$ robots.

- UB_{*i*}CP refers to the sequential combination of UB_{*i*}, for computing an upper bound ub , and CP (with the model described in Section 4.3) for computing the optimal solution.

LB and UB_{*i*} are implemented in Python. The CP model is implemented in MiniZinc [12] and solved with Chuffed [5]. All experiments are run on an Intel Core Intel Xeon E5-2623v3 of 3.0GHz×16 with 32GB of RAM.

On the left part of Fig. 2, we compare the optimal makespan with the lower bound computed by LB, and upper bounds computed by UB_{*i*} with $i \in \{1, 3, 5, 7\}$. We observe that the optimal makespan decreases as the number n of robots increases. Indeed, when n gets larger, anchor and destination points tend to be located more densely and this makes it easier to assign anchor points to closer destinations. LB is always strictly smaller than the optimal makespan, i.e., the solution of the LBAP always contains crossing segments.

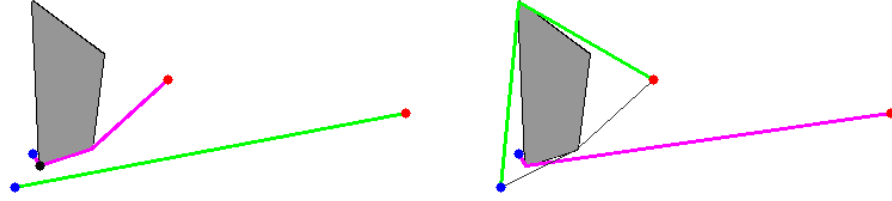
UB₁ corresponds to the solution of the LSAP, and this upper bound is much larger than the optimal makespan. VNS strongly decreases this upper bound, and the larger k_{max} the smaller the bound. Note that when $k_{max} \geq n$, VNS actually finds the optimal makespan as it explores all possible permutations of the n destinations (provided that we do not limit time, i.e., $l = \infty$). Hence, when $n = 5$, the solution of UB₅ is equal to the optimal makespan.

However, if UB_{*i*} finds smaller bounds when increasing i , it also needs more time. This is shown on the right part of Fig. 2, for instances that have $n = 50$ robots. We display the evolution of the average gap to optimality in percentage (i.e., $\frac{s-s^*}{s^*}$ where s^* is the optimal makespan and s is the current makespan) with respect to CPU time. For UB₁CP, the upper bound ub is very quickly computed by solving the LSAP, but it is 38% as large as the optimal makespan. ub is used to filter variable domains of x_i variables. However, as ub is not very tight, the construction of the table T_{ij} for every couple of variables (x_i, x_j) is time consuming. This construction phase corresponds to the horizontal part of the curve. Once the CP model has been constructed, Chuffed finds better solutions and finally proves optimality. When increasing k_{max} , the time spent by VNS to improve ub increases but, as a counterpart, the time spent to build the CP model and the time spent by Chuffed to solve it also decreases.

Table 1 allows us to study scale-up properties when increasing the number n of robots. The time spent by UB_{*i*} (t_1) strongly increases when i increases: from 0.008s when $i = 1$ to more than 16s when $i = 7$ for $n = 60$. This was expected as the time complexity of VNS is exponential with respect to k_{max} . The time limit $l = 60s$ is never reached by VNS when

■ **Table 1** Scale-up properties with respect to the number n of robots. For each $n \in \{20, \dots, 60\}$, we report CPU times of UB_iCP (in seconds), for $i \in \{1, 3, 5, 7\}$: t_1 is the time spent to solve the LSAP and improve the upper bound with VNS when $k_{max} = i$ and $l = 60s$; t_2 is the time to generate the MiniZinc model; t_3 is the time spent by Chuffed; $t_{tot} = t_1 + t_2 + t_3$ is the total time (in blue when minimal). Chuffed is limited to 3600s and the time of a run is set to 3600 when this limit is reached. In this case, t_3 is a lower bound of the actual time (and we display \geq before the time).

n	UB ₁ CP				UB ₃ CP				UB ₅ CP				UB ₇ CP			
	t ₁	t ₂	t ₃	t _{tot}	t ₁	t ₂	t ₃	t _{tot}	t ₁	t ₂	t ₃	t _{tot}	t ₁	t ₂	t ₃	t _{tot}
20	0.001	0.4	0.1	0.5	0.01	0.2	0.1	0.3	0.0	0.2	0.1	0.3	0.8	0.2	0.1	1.1
30	0.002	1.4	≥ 35.4	36.9	0.01	0.9	0.4	1.3	0.1	0.6	0.1	0.9	2.3	0.6	0.2	3.1
40	0.004	3.4	12.4	15.8	0.02	2.1	1.4	3.5	0.3	1.8	0.6	2.6	7.2	1.6	0.5	9.2
50	0.003	6.7	≥ 127.2	133.9	0.03	4.1	13.6	17.7	0.5	3.1	7.5	11.1	7.6	2.8	7.7	18.2
60	0.008	16.8	≥ 529.3	546.1	0.06	9.4	≥ 197.6	207.4	1.3	6.1	27.0	34.4	16.8	5.7	25.5	48.1



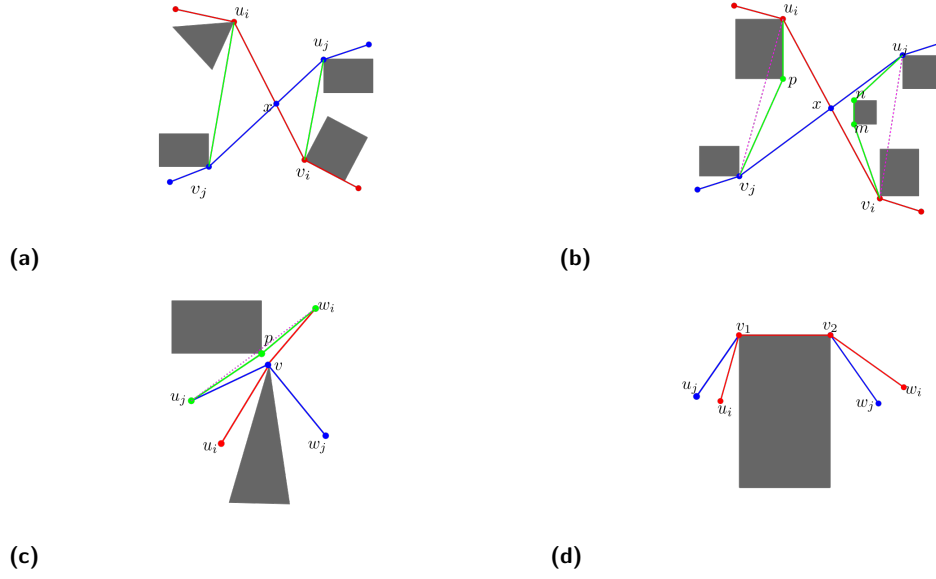
■ **Figure 3** The solution displayed on the left only uses shortest paths, and its makespan is larger than the solution displayed on the right (the green right path is longer than the black path).

$i \leq 5$ whereas it is reached when $i = 7$: for 7 (resp. 1 and 1) instances when $n = 60$ (resp. 50 and 40). However, when increasing i , UB_i computes better bounds and this reduces the time needed to generate the model (t_2) and to solve it (t_3). When $i = 1$, the time limit of 3600s is reached by Chuffed for 6 (resp. 1 and 1) instances when $n = 60$ (resp. 50 and 30). It is also reached once when $i = 3$ and $n = 60$. A good compromise is observed with UB_5CP .

5 NC-MAPF problem with obstacles

Let us now consider the case where the workspace contains obstacles. In this case, the visibility graph is no longer a bipartite graph, and a path from an anchor point to a destination may contain more than one edge. Besides, with the existence of obstacles, there might exist more than one possible path, even when restricting our attention to paths in the visibility graph, and an optimal solution may contain paths that are not shortest paths, as illustrated in Fig. 3. As a consequence, our problem is no longer a simple bipartite matching problem: we must not only choose a different destination for each anchor point, but also choose paths.

The number of paths between two points grows exponentially with respect to the number of obstacles. However, if we have an upper bound on the maximal length of a path, we can reduce the number of paths. Hence, we show how to compute upper bounds on the makespan in Section 5.1. In Section 5.2, we show how to compute all relevant paths. In Section 5.3, we describe a CP model and in Section 5.4 we experimentally evaluate our approach.



■ **Figure 4** Top (Case 1): π_i (in red) and π_j (in blue) contain two crossing segments $\overline{u_i v_i}$ and $\overline{u_j v_j}$. (a): $\overline{u_i v_j}$ and $\overline{u_j v_i}$ (in green) do not cross obstacles and $|u_i v_j| + |u_j v_i| < |u_i v_i| + |u_j v_j|$. (b): $\overline{u_i v_j}$ and $\overline{u_j v_i}$ (dotted lines) cross obstacles but $\pi_{ij} = \langle u_i, p, v_j \rangle$ and $\pi_{ji} = \langle u_j, n, m, v_i \rangle$ (in green) do not cross obstacles and $|\pi_{ij}| + |\pi_{ji}| < |u_i v_i| + |u_j v_j|$. Bottom (Case 2): π_i (in red) and π_j (in blue) cross at a common vertex. (c): By swapping w_i and w_j we obtain non crossing paths which are not shortest paths ($|\langle u_j, p, w_i \rangle| < |u_j, v, w_i|$). (d): By swapping w_i and w_j we obtain non crossing paths that have the same length.

5.1 Computation of bounds

When there are obstacles, the visibility graph G associated with W , A and D is no longer a bipartite graph. However, we can build a bipartite graph $G' = (V', E')$ such that $V' = A \cup D$ and $E' = A \times D$, and define the cost of an edge $(a, d) \in E'$ as the length of the shortest path from a to d in G . In this case, we can compute a lower bound by solving the LBAP in G' .

Let us now show that we can also compute an upper bound by solving the LSAP in G' , as a straightforward consequence of the following theorem.

► **Theorem 3.** *Let $m : A \rightarrow D$ be an optimal solution of the LSAP in G' and, for each anchor point $a_i \in A$, let π_i be the shortest path that connects a_i to $m(a_i)$ in the visibility graph. For each pair of different anchor points $\{a_i, a_j\} \subseteq A$, either π_i and π_j are not crossing, or they can be replaced by two non crossing paths π'_i and π'_j such that $|\pi_i| + |\pi_j| = |\pi'_i| + |\pi'_j|$.*

Proof. Let us suppose that there exist two crossing paths π_i and π_j . There are two cases to consider, depending on whether π_i and π_j contain two crossing segments or not.

Case 1: π_i and π_j contain two crossing segments $\overline{u_i v_i}$ and $\overline{u_j v_j}$. Let us show that this implies that m does not minimise the sum of the selected edge costs. There are two sub-cases to consider.

Subcase a: $\overline{u_i v_j}$ and $\overline{u_j v_i}$ do not cross obstacles, as illustrated in Fig. 4a.

Let π_i^p (resp. π_i^s) be the prefix (resp. suffix) of π_i that precedes (resp. succeeds) $\overline{u_i v_i}$, i.e., $\pi_i = \pi_i^p \cdot \langle u_i, v_i \rangle \cdot \pi_i^s$ where \cdot denotes path concatenation. Similarly, let $\pi_j = \pi_j^p \cdot \langle u_j, v_j \rangle \cdot \pi_j^s$. Let x be the crossing point between $\overline{u_i v_i}$ and $\overline{u_j v_j}$. We have:

$$|u_i v_i| = |u_i x| + |x v_i| \text{ and } |u_j v_j| = |u_j x| + |x v_j|. \quad (1)$$

The triangle inequality implies that

$$|u_i v_j| < |u_i x| + |x v_j| \text{ and } |u_j v_i| < |u_j x| + |x v_i|. \quad (2)$$

From Eq. (1) and (2), we infer that

$$|u_i v_j| + |u_j v_i| < |u_i v_i| + |u_j v_j|. \quad (3)$$

When swapping v_i and v_j , π_i and π_j are replaced by the two paths $\pi'_i = \pi_i^p \cdot \langle u_i, v_j \rangle \cdot \pi_j^s$ and $\pi'_j = \pi_j^p \cdot \langle u_j, v_i \rangle \cdot \pi_i^s$. From Eq. (3), we have $|\pi'_i| + |\pi'_j| < |\pi_i| + |\pi_j|$. This is in contradiction with the fact that m minimises the sum of the costs of the selected edges in G' as the costs of edges $(a_i, m(a_j))$ and $(a_j, m(a_i))$ in G' are smaller than or equal to $|\pi'_i|$ and $|\pi'_j|$, respectively (they may be strictly smaller if π'_i or π'_j are not shortest paths in G).

Subcase b: $\overline{u_i v_j}$ and $\overline{u_j v_i}$ cross obstacles, as illustrated in Fig. 4b.

In this case, we cannot simply exchange the two crossing segments to obtain two non crossing paths. However, let π_{ij} be the path from u_i to v_j corresponding to the convex hull of all vertices that belong to the triangle defined by u_i , v_j and x . This path is displayed in green in Fig. 4b. We can show that $|\pi_{ij}| < |u_i x| + |x v_j|$ by recursively exploiting the triangle inequality (see [1]). Similarly, there exists a path π_{ji} between u_j and v_i such that $|\pi_{ji}| < |u_j x| + |x v_i|$. Therefore, $|\pi_{ij}| + |\pi_{ji}| < |u_i v_i| + |u_j v_j|$. Like in Subcase a, this is in contradiction with the fact that m minimises the sum of the costs of the selected edges in G' .

Case 2: π_i and π_j do not contain crossing segments but they cross at some vertex v . Let π be the longest path that is common to both π_i and π_j , i.e., $\pi_i = \pi_i^p \cdot \pi \cdot \pi_i^s$ and $\pi_j = \pi_j^p \cdot \pi \cdot \pi_j^s$. We can exchange π_i^s and π_j^s to obtain two paths $\pi'_i = \pi_i^p \cdot \pi \cdot \pi_j^s$ and $\pi'_j = \pi_j^p \cdot \pi \cdot \pi_i^s$. There are two sub-cases to consider.

Subcase c: π'_i and/or π'_j are not shortest paths, as illustrated in Fig. 4c. In this case, we can obtain a better assignment by matching a_i with $m(a_j)$ and a_j with $m(a_i)$. This is in contradiction with the fact that m is the optimal assignment.

Subcase d: π'_i and π'_j are shortest paths, as illustrated in Fig. 4d. In this case, we can obtain an assignment which has the same cost as m by matching a_i with $m(a_j)$ and a_j with $m(a_i)$, and π'_i and π'_j no longer cross at vertex v . If they cross at some other vertex, we can recursively apply the same reasoning to either show that π'_i and π'_j are not shortest paths and exhibit a contradiction (Subcase c), or show that there exist two non crossing paths that have the same length as π'_i and π'_j (Subcase d). ◀

Hence, we can compute an upper bound by solving the LSAP in the bipartite graph G' . If some paths are crossing in the optimal solution, then we can exchange sub-paths in the crossing paths in order to obtain a solution with no crossing paths (and the same objective function value), as explained in Subcase d of Theo. 3.

Like for the NC-MAPF without obstacles, this upper bound may be improved by VNS, as explained in Section 4.2. We only have to adapt the procedure that explores the neighbourhood of a matching, in order to check that permutations do not contain crossing paths (instead of crossing edges). Note that this test is done in quadratic time with respect to the number of edges in a path (whereas it is done in constant time when there is no obstacle).

5.2 Relevant paths enumeration

The non crossing assignment in G' that minimises the makespan may not be the optimal solution of the original problem as edges of G' correspond to shortest paths, and as the optimal solution may use non shortest paths. To find the optimal solution, for each couple $(a, d) \in A \times D$, we must consider all relevant paths from a to d in the visibility graph G , where a path π is relevant if it satisfies the three following constraints:

- (C1) Given an upper bound ub on the optimal makespan (or on the maximal length of the cable anchored at a), π must be shorter than ub , i.e., $|\pi| < ub$;
- (C2) π must be elementary and not self-crossing;
- (C3) π must be a taut path (as defined in Section 3).

Before enumerating all relevant paths, we remove from the visibility graph every edge that cannot belong to a taut path, thus obtaining the reduced visibility graph [8]. Then, all relevant paths starting from an anchor point a are enumerated by performing a depth first search starting from a , and pruning branches whenever a constraint is violated. To check constraint (C3), we perform a local geometric test in constant time.

5.3 Constraint Programming Model

Let ub be an upper bound to the optimal solution, and let P be the set of relevant paths as defined in the previous section (paths in P are numbered from 1 to $\#P$). For each path $\pi \in P$, $o(\pi)$, $d(\pi)$, and $l(\pi)$ denote the origin, the destination, and the length of π , respectively. The CP model has the following variables:

- an integer variable x_i is associated with every anchor point $a_i \in A$, and its domain contains every destination that may be reached from a_i , i.e., $D(x_i) = \{d(\pi) : \pi \in P \wedge o(\pi) = a_i\}$;
- an integer variable z_i is associated with every anchor point $a_i \in A$, and its domain is the set of all paths starting from a_i , i.e., $D(z_i) = \{\pi \in P : o(\pi) = a_i\}$;
- an integer variable y represents the maximal length of a selected path.

The constraints are:

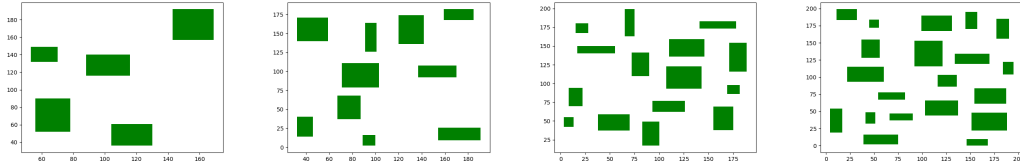
- for each pair of anchor points $\{a_i, a_j\} \subseteq A$, we post a table constraint $(z_i, z_j) \in T_{ij}$ where T_{ij} is the table that contains every couple $(\pi, \pi') \in D(z_i) \times D(z_j)$ such that $d(\pi) \neq d(\pi')$ and path π does not cross path π' ;
- for each anchor point $a_i \in A$, we post the constraint $y \geq l(z_i)$;
- we channel x_i and z_i variables by posting $x_i = d(z_i)$ and we post an *allDifferent*($\{x_i : a_i \in A\}$) constraint. This constraint is redundant as table constraints prevent selecting two paths that have a same destination. However, preliminary experiments have shown us that this improves the solution process for a wide majority of instances.

The goal is to minimise y .

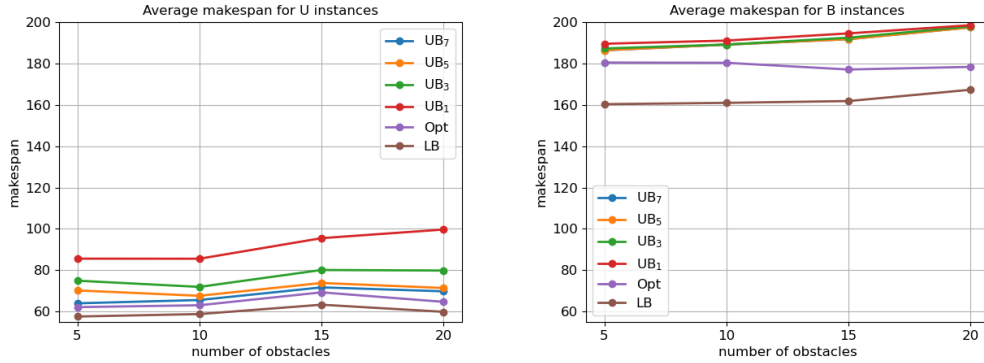
5.4 Experimental evaluation

Like in the case where there is no obstacle, we consider a bounding polygon $B = [0, 200]^2$. We introduce a parameter m to set the number of obstacles. For each obstacle, we randomly generate the coordinates of its lower left corner $(x, y) \in [0, 160]^2$ and the coordinates of its upper right corner (x', y') such that $x + 1 \leq x' \leq x + 40$ and $y + 1 \leq y' \leq y + 40$, while ensuring that the distance between two obstacles is larger than 10. We consider 4 maps with $m = 5, 10, 15, 20$ which are displayed in Fig. 5.

We consider two different kinds of distributions for generating anchor points and destinations, in order to study the impact of this distribution on solution hardness:



■ **Figure 5** Workspace when $m \in \{5, 10, 15, 20\}$ (obstacles are displayed in green).



■ **Figure 6** Evolution of the optimal makespan (Opt), the lower bound (LB) and upper bounds (UB, with $i \in \{1, 3, 5, 7\}$) when increasing the number of obstacles from 5 to 20. Left: U instances (with $n = 40$). Right: B instances (with $n = 20$).

Uniform (U): anchor points and destinations are randomly generated in W according to a uniform distribution;

Bipartite (B): anchor points (resp. destinations) are randomly generated on the left (resp. right) part of W , by constraining their abscissa to be smaller than 60 (resp. greater than 140).

For U instances, we set the number of robots n to 40, whereas for B instances it is set to 20 because these instances are harder, as explained later. For each value of m and each kind of distribution, we have generated 30 instances.

In Fig. 6, we display the optimal makespan, the lower bound computed by LB, and upper bounds computed by UB_i with $i \in \{1, 3, 5, 7\}$, for U and B instances. In both cases, we observe that the number of obstacles has no significant effect on the optimal makespan. However, the optimal makespan is much smaller for U instances than for B instances: For U instances, it is smaller than 80 whereas for B instances it is close to 180. This was expected as anchor points are constrained to be far from destinations in B instances.

For U instances, UB_1 is much larger than UB_3 which is always larger than UB_5 . UB_5 and UB_7 have close values, and UB_7 is also close to the optimal solution. Results are quite different for B instances, where UB_1 and UB_7 have very close values. In other words, VNS does not improve much the upper bound for B instances, whatever the value of k_{max} . However, the optimal solution is much smaller than the upper bounds computed by UB_i . This means that for B instances we more often need to use non shortest paths to improve the solution than for U instances (remember that VNS only considers shortest paths).

In Fig. 7, we display the evolution of the gap to optimality (in percentage) with respect to time, and in Tables 2 and 3 we display the time spent by each step of the solving process.

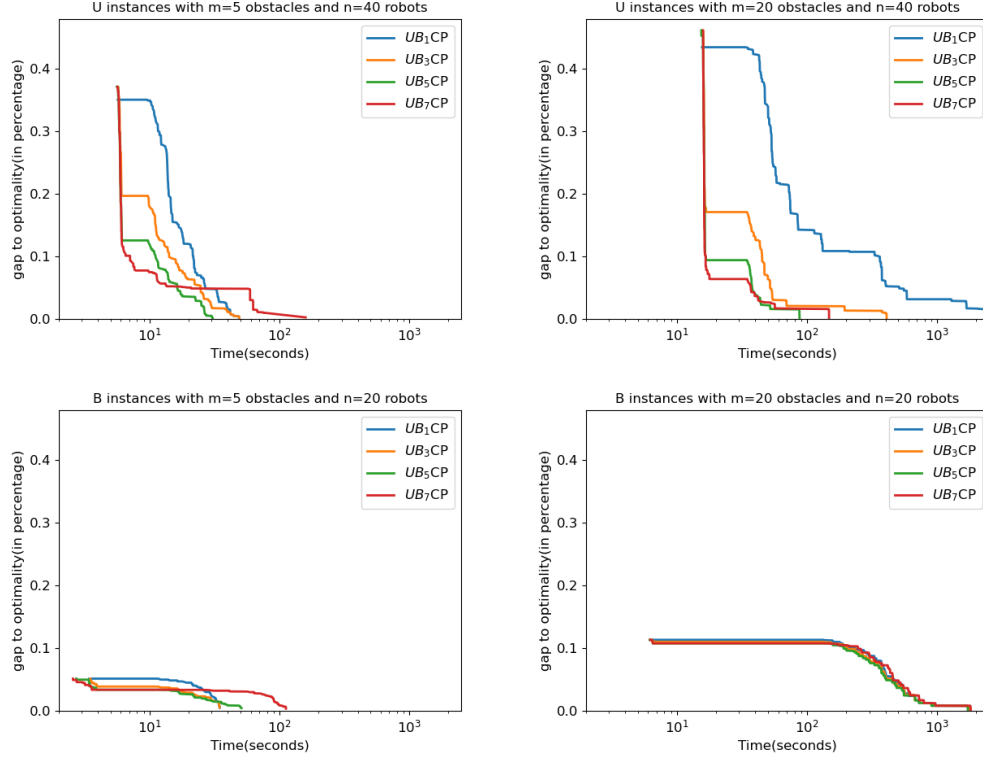


Figure 7 Evolution of the gap to optimality (in percentage) with respect to time for UB_iCP with $i \in \{1, 3, 5, 7\}$, on average for 30 instances. Top left: U instances with $m = 5$. Top right: U instances with $m = 20$. Bottom left: B instances with $m = 5$. Bottom right: B instances with $m = 20$.

For U instances, LSAP is rather long to solve (see row t_1 in the tables): around 3s when $m = 5$, and 13s when $m = 20$. This comes from the fact that the function that decides whether two paths are crossing or not has a quadratic time complexity with respect to the number of vertices in the paths, and this number increases when increasing the number of obstacles. UB_3CP , UB_5CP , and UB_7CP improve the upper bound computed by LSAP with VNS, and we observe a quick drop of the curves. Then, we observe an horizontal part which corresponds to the time needed to enumerate all relevant paths and to generate the CP model. The time needed to enumerate all paths (t_3) strongly increases when increasing the number of obstacles. This was expected as the number of paths grows with respect to the number of obstacles. t_3 slightly decreases when increasing k_{max} because the smaller the bound computed with VNS, the less relevant paths (see row RP). The time needed to generate the CP model (t_4) decreases when increasing k_{max} (because this decreases the number of relevant paths) and it increases when increasing m (because this increases the number of vertices in a path and, therefore, the time needed to decide whether two paths are crossing). Finally, after the horizontal part (corresponding to t_3 and t_4), the curves drop again because CP improves the bound. As expected, the time needed by CP to compute the optimal solution (t_5) decreases when increasing k_{max} (because the initial bound is smaller, and therefore tables are smaller), and it increases when increasing the number of obstacles (because this increases the number of relevant paths).

Now, let us look at B instances. These instances only have $n = 20$ robots (instead of 40 for U instances) because they are harder. This comes from the fact that the bound computed by UB_i is much larger, as seen in Fig. 6. This increases the number of relevant paths, as seen

■ **Table 2** Results of UB_iCP with $i \in \{1, 3, 5, 7\}$ for U instances with $n = 40$ and $m \in \{5, 10, 15, 20\}$ (average on 30 instances). t_1 = time to solve the LSAP; t_2 = time of VNS when $k_{max} = i$; t_3 = time to enumerate all relevant paths for each anchor-destination pair; t_4 = time to generate the CP model; t_5 = time to solve the CP model; $t_{tot} = t_1 + t_2 + t_3 + t_4 + t_5$; IM = number of Improving Moves for VNS; RP = maximum number of Relevant Paths between an anchor point and a destination.

m	UB ₁ CP				UB ₃ CP				UB ₅ CP				UB ₇ CP			
	5	10	15	20	5	10	15	20	5	10	15	20	5	10	15	20
t_1	2.8	5.9	9.4	13.0	2.8	5.8	9.3	12.8	2.8	5.9	9.3	12.9	2.8	5.9	9.3	12.9
t_2	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.2	0.1	0.2	0.3	8.7	4.8	5.5	7.3
t_3	4.4	10.2	21.5	33.7	3.9	8.5	14.5	21.6	3.7	8.0	14.7	21.1	3.4	7.9	14.2	20.7
t_4	5.4	9.7	39.3	75.6	3.2	3.0	4.0	8.4	2.0	2.0	4.4	7.0	1.2	1.8	3.4	6.7
t_5	122.5	23.2	47.6	184.3	2.5	7.6	1.1	9.1	1.3	0.3	1.3	0.8	0.4	0.3	1.7	0.8
t_{tot}	135.1	49.0	117.8	306.5	12.3	24.9	29.1	51.8	10.0	16.3	30.0	42.0	16.6	20.6	34.2	48.3
IM	0	0	0	0	1.4	4.0	1.7	2.0	2.6	4.0	3.4	3.8	4.6	4.6	4.4	4.6
RP	2.5	2.8	3.9	4.7	2.2	2.4	3.1	3.0	2.0	2.2	2.8	2.7	1.9	2.6	2.6	2.5

■ **Table 3** Results of UB_iCP with $i \in \{1, 3, 5, 7\}$ for B instances with $n = 20$ and $m \in \{5, 10, 15, 20\}$.

m	UB ₁ CP				UB ₃ CP				UB ₅ CP				UB ₇ CP			
	5	10	15	20	5	10	15	20	5	10	15	20	5	10	15	20
t_1	0.8	1.6	2.6	3.5	0.8	1.6	2.6	3.6	1.0	1.6	2.6	3.5	1.0	1.6	2.6	3.6
t_2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6	0.5	0.5	0.6	45.7	41.7	37.6	50.6
t_3	3.5	12.2	42.0	96.2	3.4	11.8	40.2	95.5	4.2	12.0	40.4	93.6	4.3	12.0	40.5	93.6
t_4	15.6	32.4	94.6	339.6	13.8	27.6	81.2	329.0	16.4	28.3	79.1	315.9	16.7	28.2	78.9	317.6
t_5	0.4	0.7	1.6	5.6	0.4	0.6	1.3	5.3	0.5	0.6	1.3	5.0	0.4	0.6	1.3	5.1
t_{tot}	20.3	46.9	140.7	445.2	18.4	41.6	125.3	433.4	22.6	43.0	123.8	418.7	68.1	84.0	160.9	470.4
IM	0	0	0	0	0.3	0.2	0.3	0.2	0.4	0.3	0.3	0.2	0.4	0.3	0.3	0.2
RP	6.8	8.0	13.3	23.3	6.4	7.8	12.6	22.9	6.3	7.8	12.4	22.7	6.4	7.8	12.4	22.7

when looking at row RP: when $m = 20$, this number is larger than 20 for B instances whereas it is smaller than 5 for U instances. Also the number of vertices in a path increases. Hence, the time needed to enumerate all relevant paths (t_3) is much larger for B instances than for U instances (e.g., when $m = 20$ and $k_{max} = 7$, 94s for B and 21s for U). Also, the time needed to generate the CP model (t_4) is much larger (e.g., when $m = 20$ and $k_{max} = 7$, 318s for B and 7s for U). However, the time spent by VNS (t_2) is much smaller (e.g., when $m = 20$ and $k_{max} = 7$, 4s for B instead of 13s for U) because n is twice as small for B than for U. Finally, the time needed to solve the CP model increases when increasing m , but it does not decrease when increasing k_{max} . This comes from the fact that VNS does not improve much the upper bound, whatever the value of k_{max} (as seen in Fig. 6). Row IM displays the number of improving moves performed by VNS, and we observe that this number is close to 0 for B instances.

For both B and U instances, we observe a good compromise between the time spent by VNS to improve the bound, and the time spent to enumerate relevant paths, build the CP model and solve it when $k_{max} \in \{3, 5\}$.

As observed on row RP of Tables 2 and 3, the number of relevant paths being searched for each anchor/destination pair increases as m gets larger. Theoretically, this number exponentially grows with the number of obstacles. When the optimal makespan is small and

■ **Table 4** Impact of the parameter p on the time needed to enumerate relevant paths (t_3), to generate the CP model (t_4), and to solve it (t_5), and on the gap to optimality (in percentage) for B instances when $k_{max} = 5$ and $m = 20$.

	p=1	p=2	p=4	p=8	p=16	no limit
t_3	0.0	65.5	76.6	87.1	93.2	93.6
t_4	1.9	8.2	34.4	121.7	265.5	315.9
t_5	0.1	0.2	0.6	2.2	4.1	5.0
$t_{tot} = t_1 + t_2 + t_3 + t_4 + t_5$	6.9	78.0	115.8	215.1	367.9	418.7
gap to optimality	10.8%	5.9%	0.9%	0.0%	0.0%	0.0%

the upper bound computed by VNS is close enough to it, the actual number of relevant paths is rather small (e.g., smaller than 3 for U instances when $k_{max} \geq 5$). However, for B instances, this number is greater than 20 when $m = 20$, and the time needed to enumerate these paths and generate the CP model becomes greater than 400s. To overcome this problem, we can introduce a parameter p and limit the number of relevant paths to p (keeping the p best ones whenever the number of relevant paths is greater than p). Of course, in this case we no longer guarantee optimality as it may happen that the optimal solution uses a path that has been discarded. In table 4 we display the results of UB₅CP for different values of p on B instances when $m = 20$. Not surprisingly t_2 , t_3 , t_4 are all reduced as p decreases, while the average gap to optimality increases up to more than 10% for $p = 1$. In our experiment, $p = 8$ ensures that an optimal solution can always be found, and divides by 2 the total time.

6 Conclusion

We have introduced a new MAPF problem which is motivated by an industrial application where tethered robots cannot cross cables. We have shown that we can compute feasible solutions that provide upper bounds in polynomial time, by solving LSAPs, even when the workspace has obstacles. We have also introduced a VNS approach that improves the feasible solution of LSAP by iteratively permuting k destinations, and a CP model that solves the problem to optimality. Finally, we have proposed to sequentially combine VNS and CP, thus allowing us to use the upper bound computed by VNS to filter domains.

Experimental results on randomly generated instances have shown us that the number of obstacles has a strong impact on the solving time. When there is no obstacle, there is exactly one path between every origin/destination pair of points, and this path is a straight line segment. When increasing the number of obstacles, the number of paths between two points grows exponentially, even when limiting our attention to taut paths. Hence, it is important to have good upper bounds on the optimal solution in order to reduce the number of candidate paths. Also, when increasing the number of obstacles, the number of vertices in a path increases linearly, and this has an impact on the time needed to decide whether two paths are crossing or not.

We have reported experiments on randomly generated instances that allow us to control the number of obstacles and the number of robots. We have considered two models for generating anchor and destination points, and we have observed that the distribution of the points has a strong influence on the solution process. In particular, when anchor points and destinations are constrained to belong to two opposite sides of the workspace, this increases the hardness of the problem because this increases the makespan and, therefore, the number of relevant paths and the number of vertices in a path. We have introduced a parameter to control the number of paths and the solving time, at the price of the loss of optimality.

For future work, we plan to investigate other solving approaches, such as Tabu search or Integer Linear Programming. Also, we want to extend the work to non-point agents by considering robots with a body, generating complementary constraints on their motions and their cables. This will allow to deal with industrial and robotics applications.

References

- 1 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd ed. edition, 2008.
- 2 S. Bhattacharya, M. Likhachev, and V. Kumar. Topological constraints in search-based robot path planning. *Autonomous Robots*, 33(3):273–290, 2012.
- 3 Rainer E. Burkard and Eranda Çela. Linear assignment problems and extensions. *Handbook of Combinatorial Optimization*, pages 75–149, 1999.
- 4 J. Carlsson, B. Armbruster, Saladi Rahul, and Haritha Bellam. A bottleneck matching problem with edge-crossing constraints. *Int. J. Comput. Geom. Appl.*, 25:245–262, 2015.
- 5 Geoffrey Chu and Peter J. Stuckey. Chuffed solver description, 2014. Available at http://www.minizinc.org/challenge2014/description_chuffed.txt.
- 6 Susan Hert and Vladimir J. Lumelsky. The ties that bind: Motion planning for multiple tethered robots. *Robotics Auton. Syst.*, 17(3):187–215, 1996.
- 7 H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.
- 8 Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.
- 9 Jiaoyang Li, Pavel Surynek, Ariel Felner, Hang Ma, T. K. Satish Kumar, and Sven Koenig. Multi-agent path finding for large agents. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):7627–7634, July 2019. doi:10.1609/aaai.v33i01.33017627.
- 10 Tomás Lozano-Pérez and Michael A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Commun. ACM*, 22(10):560–570, 1979.
- 11 Nenad Mladenovic and Pierre Hansen. Variable neighborhood search. *Comput. Oper. Res.*, 24(11):1097–1100, 1997.
- 12 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*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.
- 13 David W. Pentico. Assignment problems: A golden anniversary survey. *Eur. J. Oper. Res.*, 176(2):774–793, 2007.
- 14 Putnam. Problem a4, 1979.
- 15 Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66, 2015. doi:10.1016/j.artint.2014.11.006.
- 16 S. Thomas, Dipti Deodhare, and M. N. Murty. Extended conflict-based search for the convoy movement problem. *IEEE Intelligent Systems*, 30:60–70, 2015.
- 17 Thayne T. Walker, Nathan R. Sturtevant, and Ariel Felner. Extended increasing cost tree search for non-unit cost domains. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI’18*, page 534–540. AAAI Press, 2018.
- 18 J. Yu and S. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *In Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 1444–1449, 2013.
- 19 Xu Zhang and Quang-Cuong Pham. Planning coordinated motions for tethered planar mobile robots. *Robotics and Autonomous Systems*, 118:189–203, 2019. doi:10.1016/j.robot.2019.05.008.

Positive and Negative Length-Bound Reachability Constraints

Luis Quesada ✉ 

Insight Centre for Data Analytics, School of Computer Science, University College Cork, Ireland

Kenneth N. Brown ✉ 

Insight Centre for Data Analytics, School of Computer Science, University College Cork, Ireland

Abstract

In many application problems, including physical security and wildlife conservation, infrastructure must be configured to ensure or deny paths between specified locations. We model the problem as sub-graph design subject to constraints on paths and path lengths, and propose length-bound reachability constraints. Although reachability in graphs has been modelled before in constraint programming, the interaction of positive and negative reachability has not been studied in depth. We prove that deciding whether a set of positive and negative reachability constraints are satisfiable is NP complete. We show the effectiveness of our approach on decision problems, and also on optimisation problems. We compare our approach with existing constraint models, and we demonstrate significant improvements in runtime and solution costs, on a new problem set.

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

Keywords and phrases Reachability Constraints, Graph Connectivity, Constraint Programming

Digital Object Identifier 10.4230/LIPIcs.CP.2021.46

Funding This publication has emanated from research conducted with the financial support of Science Foundation Ireland under grant numbers 16/SP/3804 and 12/RC/2289-P2, which are co-funded under the European Regional Development Fund.

Acknowledgements We thank Seán Óg Murphy, Liam O'Toole and Cormac J. Sreenan for initial discussions on the application that motivated this research, and Jaime Arias for sharing his experience with Visual Studio Code. Finally, we thank the anonymous referees for their help in improving the paper.

1 Introduction

Many application problems require reasoning about reachability, including road network design [10], in-building access control [19, 16] and ecosystem management [6]. In each case, paths must be ensured or denied between sets of locations. Often there are further constraints on the lengths of those paths. For example, in buildings, from every location there must exist an accessible path to a fire escape of less than a specified limit [9], while physical security may require protected assets to be kept at least a minimum distance away from unauthorised users [11]. In some cases, solutions may need to be dynamic, responding to movements of either assets, hazards, or users in order to maintain the reachability requirements. In each case, the problem can be considered as sub-graph design, enabling or disabling edges in a larger graph. Some applications have optimisation criteria, including minimising the number of edges (e.g. maintenance cost), or minimising sums of path lengths (e.g. expected travel distance). In this paper, we model the problems in constraint programming, including constraints on reachability and on path length.

Constraint-based graph design for reachability has been studied before [21, 20, 8, 4, 1], including constraints on the costs of paths [22, 5]. However, these papers focus on positive constraints, requiring paths between pairs of nodes. Little attention has been paid to the



© Luis Quesada and Kenneth N. Brown;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 46; pp. 46:1–46:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

interaction of positive and negative constraints. The interaction of the two makes the problem significantly more complex, and we show that determining whether a graph contains a subgraph that satisfies mixed positive and negative reachability constraints is NP-complete. To incorporate the restrictions on path length, we introduce length-bound reachability constraints, and to support large sets of constrained pairs, we implement propagation based on upper and lower bounds on all-pairs shortest paths. We evaluate our constraints on random instances based on road networks, and consider both decision and optimisation problems. We compare to the Dreachable constraint [4], and we show significantly faster runtimes for decision problems and orders of magnitude improvement in costs for time-limited optimisation problems.

In the remainder of the paper, we briefly summarise related work, and we then establish the problem complexity of mixed positive and negative reachability sub-graph design. We describe the length bound reachability constraint, including transitive closure and dominators. Finally, we present the empirical evaluation, showing the behaviour of different versions of the constraints on random problems.

2 Related Work

Reachability constraints [20, 4] enforce a graph variable to specify a graph that contains paths between designated vertices. Path constraints [22, 7, 4] enforce a graph variable to represent a path in the graph between two specified vertices, including in some cases bounds on the path lengths [22, 5]. Our interest is in being able to express both positive and negative reachability – that is, to deny connections between some vertex pairs, while enforcing connections between others. Reachability can be expressed in terms of path constraints by having a path constraint for each positive pair, and a negated path constraint for each negative pair. However, having one constraint for each reachability pair leads to redundant computation as the constraints do not share information on their selected paths. For instance, if a path constraint enforcing reachability from i to j discovers that i must reach k , the other path constraints cannot take advantage of that knowledge because they only communicate via the decision variables. That is, not only are the space and time complexity of path constraints higher, but the level of pruning achieved is much less than could be expected from a single global constraint that incorporate all positive and negative reachability constraints on the same graph. One-to-many (e.g., [8, 4]) and many-to-many (e.g., [20, 1]) approaches have been proposed to mitigate redundancy and improve pruning when handling several reachability constraints. However, their focus is still on positive reachability constraints rather than on the combination of both positive and negative reachability constraints. We are not aware of any research handling simultaneous positive and negative reachability constraints.

3 Positive and Negative Reachability Constraints

In this section we describe the input and the output of the problems we study, and then establish the complexity of the core problem.

3.1 Input and Output

- **Input.** We have the following input:
 - A directed graph $G = (V, E)$, where each edge $e \in E$ has an integer cost e_c ¹.

¹ We use the terms *cost* and *length* interchangeably.

- A set of positive reachability constraints (*PRC*). A $prc \in PRC$ is represented with a tuple (i, j, λ) meaning that there is at least one path from i to j in the resulting graph whose cost is less than or equal to λ .
- A set of negative reachability constraints (*NRC*). An $nrc \in NRC$ is represented with a tuple (i, j, λ) meaning that there is no path from i to j in the resulting graph whose length is less than λ .

In what follows we may omit λ in a positive reachability constraints if there is no bound on the length of the path from i to j . Similarly, we may omit it in a negative reachability constraint if all paths from i to j are to be denied.

- **Output.** We consider three possible outputs:
 - **Obj 1.** Find G' subgraph of G where all *prcs* and *nrcs* are respected.
 - **Obj 2.** Find G' subgraph of G where all *prcs* and *nrcs* are respected, and the sum of the costs of edges of G' is minimised.
 - **Obj 3.** Find G' subgraph of G where all *prcs* and *nrcs* are respected, and the sum of the costs of the shortest paths in G' connecting the vertices in the *prcs* is minimised.

3.2 Complexity of length-bound reachability constraint problems

In this section we discuss the complexity of the decision problems involved in length-bound reachability constraint problems.

3.2.1 Positive and negative reachability constraints

If we only have positive reachability constraints (i.e., $NRC = \emptyset$), checking whether the set of reachability constraints in PRC is satisfiable is straightforward: we just need to check the existence of a path for every positive reachability constraint. The case where we only have negative reachability constraints is trivial since a totally unconnected graph would satisfy all of them. However, a mix of positive and negative reachability constraints is more challenging. Let us formally define the problem as follows:

► **Definition 1** (The Positive and Negative Reachability Constraints problem (*PNRC*)). *Given a directed graph G , a set of unbounded positive reachability constraints PRC , and a set of unbounded negative reachability constraints NRC , is there a sub graph G' of G that satisfies all constraints in PRC and NRC ?*

► **Theorem 2.** *PNRC is NP complete*

Proof. First, we show PNRC is in NP. We use G' as the certificate, and run Floyd Warshall [3] on it to get the lengths of all-pairs shortest paths. If there is no path between two vertices, it returns ∞ as the length. For each $(s, t) \in PRC$, check that the length of their path is less than ∞ ; for each $(p, q) \in NRC$, check that the length is equal to ∞ . This is polynomial.

We now give a reduction from 3SAT [12] to PNRC. We map a SAT instance to a directed graph (following the approach in [15]) with reachability constraints. We start with a SAT instance $\bigwedge_{i=1}^n x_{i1} \vee x_{i2} \vee x_{i3}$, where each x_{ij} is a literal (i.e., it is either a variable or a negation of a variable), and construct a directed graph $G = (V, E)$, where the vertices are associated with levels from 0 to $n + 1$.

1. Create vertices $s, t \in V$. s is the only vertex at level 0. t is the only one at level $n + 1$.
2. Now create a vertex for each literal, add them to V , and assign the vertices for literals of clause i to level i . That is, the three vertices of level i are x_{i1} , x_{i2} , and x_{i3} .
3. Add to E a directed edge from s to each vertex of level 1.

4. For each level i from 1 to $n - 1$, add to E an edge from each vertex of level i to each vertex of level $i + 1$.
5. Add to E a directed edge from each vertex of level n to t .

To ensure that the SAT instance is satisfied, we add a positive reachability constraint from s to t . A path from s to t represents an assignment of values to the Boolean variables in the literals represented by the vertices in the path. That is, it assigns 1 to the Boolean variable if the literal is positive and 0 otherwise. A path from s to t satisfies all the clauses, and it is consistent if it does not assign two different values to the same variable. To ensure that the assignment to the SAT instance is consistent we add negative reachability constraints as follows. For any two vertices x_{ij} and x_{kl} ($i < k$) representing literals that negate each other, we must ensure that one is not reachable from each other. Since the directed edges only ascend the levels, we only need to add (x_{ij}, x_{kl}) to NRC .

We now show that the SAT instance is satisfiable if and only if there is a subgraph of G that satisfies the constraints. If the SAT instance has a solution, then from it pick one TRUE literal in each clause. These literals define a path from s to t in the graph (so satisfies the *prc* in the *PNRC* instance). We select those vertices (literals) and edges as G' . There cannot be any conflicting literals selected (since it is a 3SAT solution), and so no *nrc* can be violated, and so the *PNRC* instance has a solution. If the *PNRC* instance has a solution, there is an s - t path. This path has one TRUE literal in each clause. The graph obeys the *nrcs*, and we have not added reachability, so there can be no conflicting literals in the path. Every 3SAT variable not yet determined is then set to 0. This is a solution to the SAT instance.

Finally, we show that the construction is polynomial. If there are m clauses in the 3SAT instance, then there are $n = 3 * m$ occurrences of literals. Each clause is processed in turn, building each layer in the graph, with one vertex per occurrence of a literal. For each vertex, we add incoming edges from the vertices in the previous layer, which is $3 + 3 * (n - 3) + 3$ edges. We add one *prc* for (s, t) . For the *nrcs*, each time we add a vertex to the graph, we sweep through the previous clauses and their literals. For each previous literal that negates the current one we add an *nrc* between the corresponding previous vertex and the current vertex. That requires $\mathcal{O}(n^2)$ checks and additions of *nrcs*. ◀

3.2.2 Positive and negative reachability constraints with bounds on the length of the paths

As the problem is already NP-complete without considering bounds on the lengths of the paths, it follows that it is also NP-complete when considering bounds. Note, however, that what makes the problem hard is the interaction between positive and negative constraints. If $NRC = \emptyset$, the decision problem reduces to checking the lengths of the shortest paths for every pair of vertices in *PRC*. As mentioned in the previous section, we are also interested in minimising the sum of the costs of the selected edges and minimising the sum of the costs of the paths connecting the vertices in *PRC*. Both optimisation problem are clearly NP-Hard as both involve solving decision problems that are NP-complete.

4 CP approaches

In this section we present three approaches to model and solve the problem. G refers to the input graph, and G' refers to the resulting graph. In each model we separate the constraints into two groups: essential and redundant. Essential constraints are required for the solution to be sound. Redundant constraints are added to reduce the search space.

4.1 Common variables

The models presented have the following common variables:

- We associate a Boolean variable be_e with edge e . $be_e = 1$ means $e \in G'$
- Each pair of vertices (i, j) is associated with a Boolean variable br . $br_{ij} = 1$ means i reaches j in G'
- We have an integer variable ep_e per edge e . This variable represents the penalty associated with the edge, which is dependent on the selection of the edge. More precisely, ep_e is either equal to e_c if $e_c \in G'$ or ∞ if $e_c \notin G'$.
- Each pair of vertices (i, j) is associated with an integer variable pc , which represents the cost of the shortest path from i to j . That is, pc_{ij} is either equal to the shortest path cost from i to j in G' or ∞ if there is no path in G' going from i to j .

4.2 The length-bound reachability constraint (LBRC)

All constraints are essential in this approach. This approach relies on the connection between the reachability of vertex j from a vertex i and the shortest path from i to j . If there is no path from i to j then we define the length of the shortest path to be ∞ .

- The *PRC* constraints are modelled in terms of the pc variables:

$$(i, j, \lambda) \in PRC \Rightarrow pc_{ij} \leq \lambda \quad (1)$$

- The *NRC* constraints are modelled in terms of the pc variables:

$$(i, j, \lambda) \in NRC \Rightarrow pc_{ij} \geq \lambda \quad (2)$$

- The cost of the shortest path to a reachable node is less than ∞ :

$$br_{ij} = 1 \Leftrightarrow pc_{ij} < \infty \quad (3)$$

- The penalty of an edge e is either the cost of the edge, if the edge is selected, or ∞ :

$$be_{ij} = 1 \Leftrightarrow ep_e = e_c \quad (4)$$

- The cost of the shortest path from i to j must be the edge (i, j) or, for some in-neighbour x of j , a shortest path from i to x followed by the edge (x, j) :

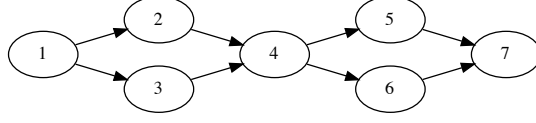
$$pc_{ij} = \min(\{ep_{(i,j)}\} \cup \{pc_{ix} + ep_{(x,j)} | x \in in(j)\}) \quad (5)$$

where $in(j)$ refers to the incoming neighbours of j , and for each pair (i, j) , we have $1 + |in(j)|$ cost variables. This maintain bounds on the costs of the shortest paths, and as edges are denied or enforced, the bounds are updated and propagated.

We define the length-bound reachability constraint (LBRC) as the constraint that keeps variables be , ep , pc , and br consistent in the way described above. More formally,

► **Definition 3.**

$$\begin{aligned} LBRC(be, ep, pc, br) \quad \equiv \quad & (i, j, \lambda) \in PRC \Rightarrow pc_{ij} < \lambda \quad \wedge \\ & (i, j, \lambda) \in NRC \Rightarrow pc_{ij} \geq \lambda \quad \wedge \\ & br_{ij} = 1 \Leftrightarrow pc_{ij} < \infty \quad \wedge \\ & be_{ij} = 1 \Leftrightarrow ep_e = e_c \quad \wedge \\ & pc_{ij} = \min(\{ep_{(i,j)}\} \cup \{pc_{ix} + ep_{(x,j)} | x \in V\}) \end{aligned}$$



■ **Figure 1** A simple directed graph with vertex 4 dominating vertex 7 on paths from vertex 1.

4.3 The length-bound reachability constraint with Transitive closure (LBRC+TC)

Consider the graph in Figure 1, and assume that we have:

$$br_{17} = 0 \wedge br_{14} = 1 \wedge br_{47} = 1$$

These reachability constraints are unsatisfiable. However we cannot detect that unsatisfiability by pure propagation with LBRC. Notice that:

$$\begin{aligned} br_{14} = 1 &\Rightarrow pc_{14} < \infty \\ &\Rightarrow (ep_{12} + ep_{24} < \infty) \vee (ep_{13} + ep_{34} < \infty) \end{aligned}$$

Similarly, we have that:

$$\begin{aligned} br_{47} = 1 &\Rightarrow pc_{47} < \infty \\ &\Rightarrow (ep_{45} + ep_{57} < \infty) \vee (ep_{46} + ep_{67} < \infty) \end{aligned}$$

And we also have that:

$$\begin{aligned} br_{17} = 0 &\Rightarrow pc_{17} = \infty \\ &\Rightarrow pc_{15} + ep_{57} = \infty \\ &\Rightarrow pc_{16} + ep_{67} = \infty \end{aligned}$$

However, we cannot go any further since both pc_{15} and pe_{57} can be set to ∞ (i.e., we have a disjunction). We have the same situation with pc_{16} and pe_{67} . The propagators behind these sum constraints will just wait until one of the variables become less than ∞ to set the other variable to ∞ , or until one of the variables becomes ∞ to declare entailment.

In order to address this lack of propagation, in addition to the essential constraints of LBRC we add redundant constraints implementing the transitive closure of the output graph:

$$br_{ij} = 1 \wedge br_{jk} = 1 \Rightarrow br_{ik} = 1 \quad (6)$$

Taking into account these redundant constraints, we have that $br_{17} = 0$ implies $br_{14} = 0 \vee br_{47} = 0$, which is in direct contradiction with $br_{14} = 1 \wedge br_{47} = 1$.

4.4 The length-bound reachability constraint with Dominators (LBRC+Dom)

Consider again the graph in Figure 1. Suppose now that we have:

$$br_{17} = 1 \wedge br_{14} = 0$$

These two constraints are clearly unsatisfiable given the structure of the graph. However, we cannot detect that unsatisfiability just by pure propagation with LBRC+TC. To do that we need to take into account that all paths from vertex 1 to vertex 7 go through vertex 4.

In this approach we use a constraint from [20], which relies on the notion of dominators:

► **Definition 4.** Given a directed graph $G = (V, E)$, and vertices $i, j, k \in V$, j is a dominator of k with respect to i if all paths from i to k in G go through j

For Figure 1 we see that vertex 4 is a dominator of vertex 7 with respect to vertex 1.

► **Definition 5.** The $DomReach(be, dom, br)$ constraint holds iff br represents the transitive closure of G' , the graph represented by be , and dom is a 3D array representing the dominators of G' , i.e., $dom_{ijk} = 1$ iff j is a dominator of k with respect to i in G' .

We implemented the $DomReach$ constraint following the same ideas of [20], with two main differences. First, we omit the pruning rules associated with the relation between the graph and its transitive closure, as we achieve this through the implementation of Equation 5. Second, we maintain dominators from all sources. In [20], the focus is on computing a single path with mandatory nodes, but in our case reachability constraints could involve all possible sources, which justify maintaining dominators from all sources.

We replace Constraint 6 in LBRC+TC with the following constraints:

- One $DomReach$ constraint to enforce the transitive closure and the dominator relation:

$$DomReach(be, dom, br) \quad (7)$$

- For all i, j, k :

$$(dom_{ijk} = 1 \wedge br_{ik} = 1) \Rightarrow (br_{ij} = 1 \wedge br_{jk} = 1) \quad (8)$$

With these redundant constraints, we have that $br_{17} = 1$ and $dom_{147} = 1$ implies $br_{14} = 1$, which contradicts $br_{14} = 0$.

5 Dreachable approach

We can also model unbounded positive reachability constraints using *Dreachable* [4].

► **Definition 6.** The $Dreachable(G, s, G^*)$ constraint holds iff G^* is a subgraph of G such that all vertices and edges of G^* are reachable from s in G^* .

- The (unbounded) PRC constraints are modelled in terms of the br variables:

$$(i, j, _) \in PRC \Rightarrow br_{ij} = 1 \quad (9)$$

- The (unbounded) NRC constraints are modelled in terms of the br variables:

$$(i, j, _) \in NRC \Rightarrow br_{ij} = 0 \quad (10)$$

- One *Dreachable* constraint per source in PRC is posted:

$$\forall i \in sources(PRC) : Dreachable(G, i, G^i) \quad (11)$$

where G^i is equal to the projection of G' (the output graph) on the vertices that are reachable from i , i.e., $G'[\{j | br_{ij} = 1\}]$.

- The transitive closure is enforced:

$$br_{ij} = 1 \wedge br_{jk} = 1 \Rightarrow br_{ik} = 1 \quad (12)$$

Note that in this model the transitive closure constraints are essential to ensure that the negative reachability constraints are respected.

In [4] it is stated that dominators are used in *Dreachable* to deal with cases like the one in Section 4.4, so this approach should achieve the same level as pruning of LBRC+Dom.

6 Empirical Evaluation

We performed our experiments on machines with Intel(R) Xeon(R) CPU with 2.40GHz running on Ubuntu 18.04. The version of Minizinc [17] used in the experiments is 2.5.3, which comes with Gecode [13] 6.3.0 and Chuffed [2] 0.10.4. LBRC, LBRC+TC and LBRC+Dom were implemented directly in Gecode 6.2.0. However, LBRC was also implemented in Minizinc to be able to compare the same model in both Chuffed and Gecode.

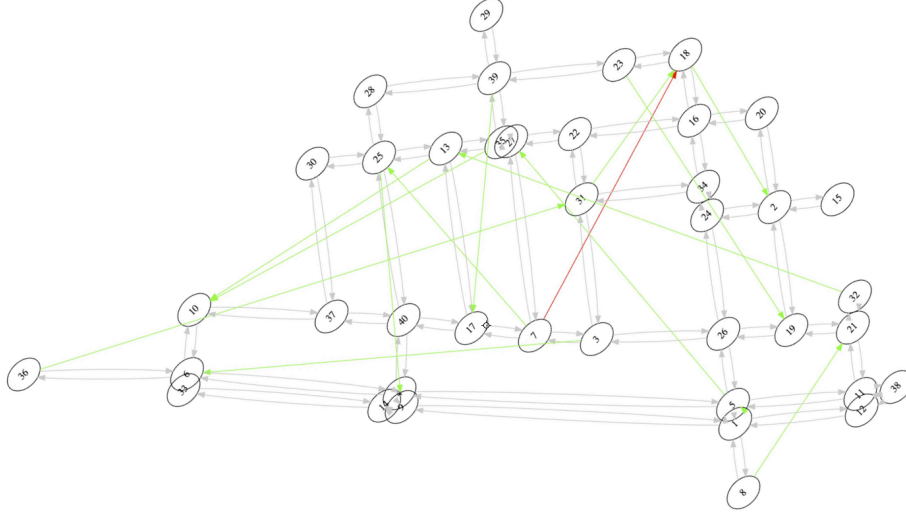


Figure 2 An instance of 40 vertices, 14 positive reachability constraints (coloured in green) and 1 negative reachability constraint (coloured in red).

6.1 Instances

The graphs considered in the evaluation are planar graphs extracted from a real-world road network generated using the GIS-F2E tool [14]. The generated road network has 137626 nodes and 194996 (undirected) links. From this network we randomly select subgraphs by choosing a random node and running Breadth First Search from that node, stopping the search when reaching the specified number of nodes for the subgraph. The approaches evaluated are based on directed graphs so the undirected graphs are converted to directed graphs by adding symmetric edges. One instance is shown in Figure 2. In what follows we use \mathcal{G} to refer to the directed version of the generated road network.

The instances are characterised in terms of the following features:

- *size*: the number of nodes of the graph. The set of sizes considered is $\{20, 24, 28, 32, 36, 40\}$. We randomly select a subgraph of *size* vertices from \mathcal{G} . For each size, we generate 10 graphs for the experiments in Figures 3, 4 and 5, and 100 for the other experiments.
- *Cs*: the percentage of selected constraints. For each graph we randomly select $Cs\%$ of the possible pairs. Each pair denotes a positive or negative reachability constraint.
- *(pos, neg)*: the ratio of positive and negative reachability constraints. For each set of selected reachability constraints, $pos\%$ are labelled as positive and $neg\%$ as negative.
- *(pb, nb)*: the bounds on the positive and negative reachability constraints. Let $maxp$ be the maximum of the lengths of the shortest paths between the positive reachability pairs. All positive reachability constraints are subject to an upper bound of $maxp \times (1 + pb/100)$. $maxn$ is the maximum of the lengths of the shortest paths between the negative reachability pairs, and all negative reachability constraints have a lower bound of $maxn \times (1 + nb/100)$.

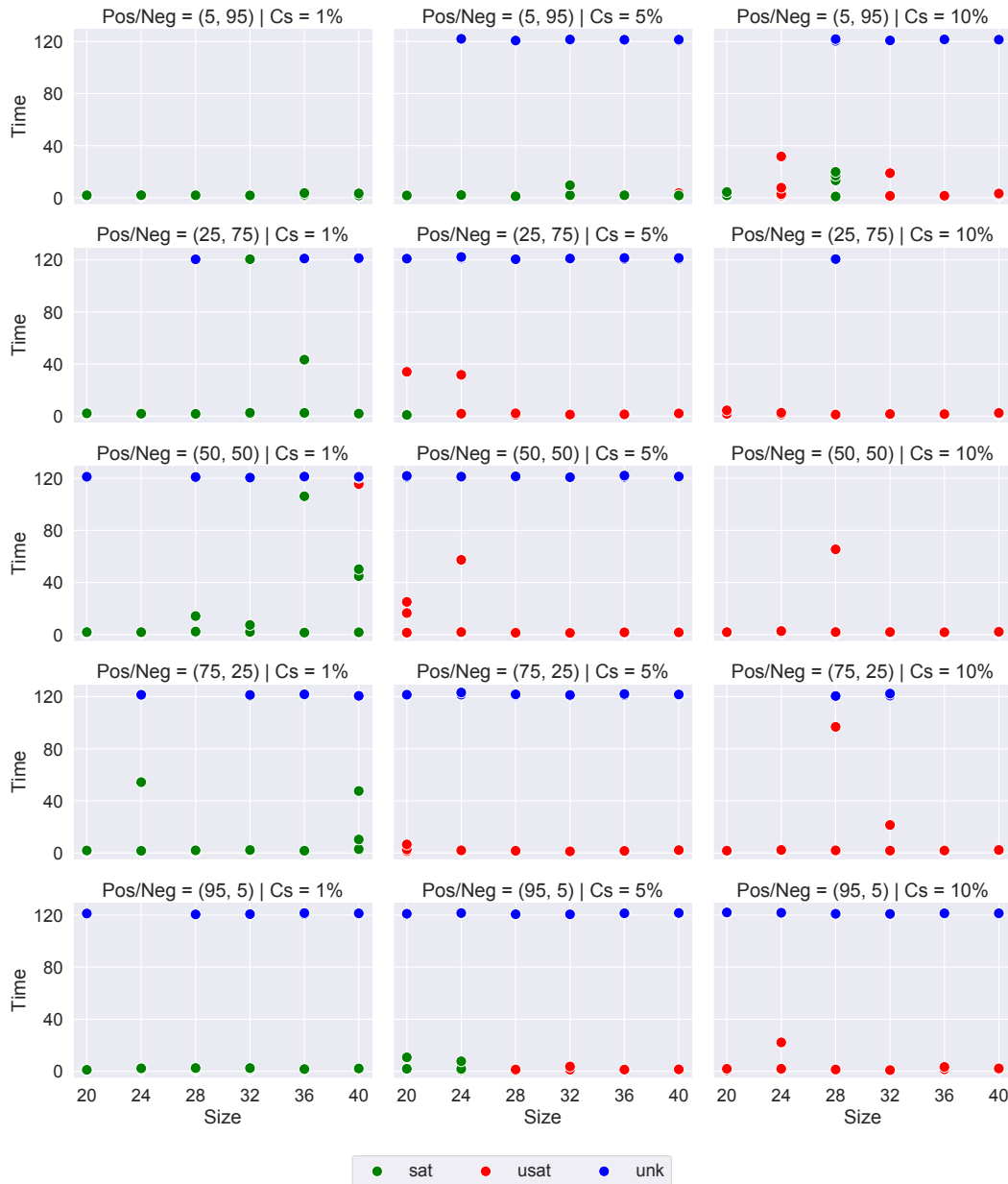


Figure 3 Performance of LBRC in Gecode. Instances are classified as satisfiable (sat), unsatisfiable (usat), and unknown (unk).

For example, consider the case where $size = 40$, $Cs = 1$, and $(pos, neg) = (50, 50)$, with no length bounds. There are $40 \times 39 = 1560$ possible pairs. We randomly choose 1% of the pairs, which amounts to 16 (after rounding up). Of these 16 pairs, 8 (50%) are labelled as positive reachability constraints, and 8 are labelled as negative reachability constraints.

The first set of experiments, whose results are shown in Figures 3 and 4, correspond to using LBRC on all the instances with both Gecode and Chuffed via Minizinc. We were interested in real-time solutions for our application, therefore we set the objective to the minimisation of path lengths (Obj 3), and the timeout to 120 seconds. These experiments let us assess the role that the features described play in the difficulty of solving the instances. A high Cs value usually led to trivially unsatisfiable instances, so we focused on small values.

46:10 Positive and Negative Length-Bound Reachability Constraints

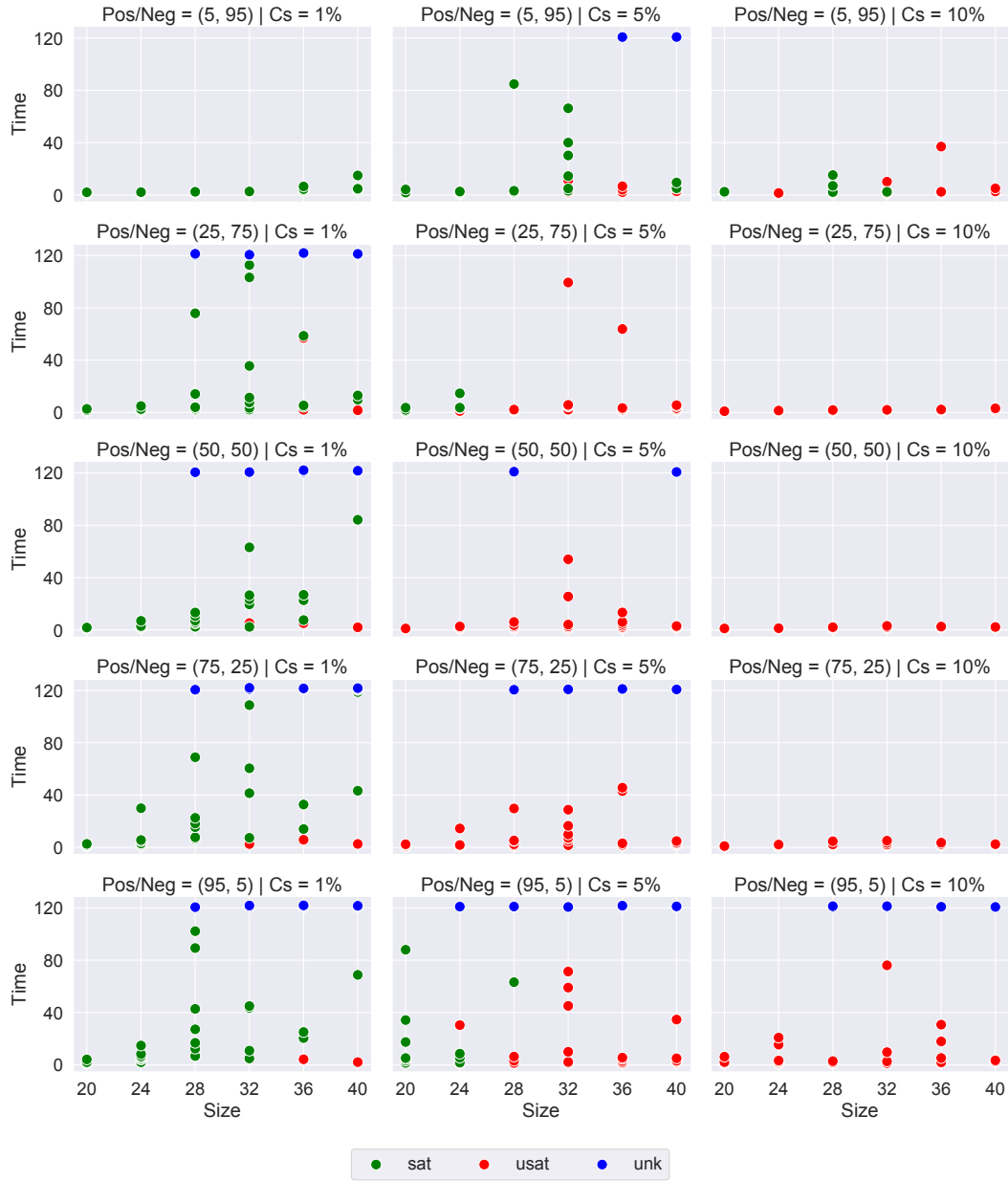


Figure 4 Performance of LBRC in Chuffed. Instances are classified as satisfiable (sat), unsatisfiable (usat), and unknown (unk).

The behaviour of Gecode was extreme: in most cases either it solved the instances very quickly or did not solve them at all. In particular, unsatisfiable instances are challenging for Gecode. Chuffed was better at dealing with the unsatisfiable instances, but its performance was inferior when dealing with satisfiable cases.

We note that a small number of negative reachability constraints are enough to create challenging instances. For example, when $size = 40$, $Cs = 1$, and $(pos, neg) = (95, 5)$, there is only one negative reachability constraint, but most of the instances in this class remained unsolved. We focus on this case in further experiments.

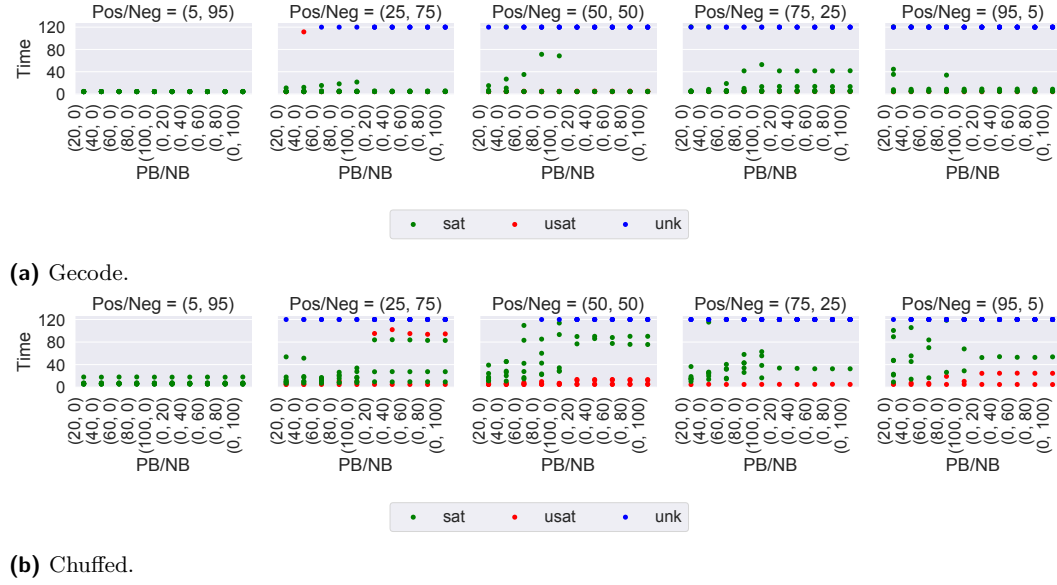


Figure 5 Performance of LBRC on length-bound instances. Instances are classified as satisfiable (sat), unsatisfiable (usat), and unknown (unk).

In Figure 5 we show the impact of the path-length bounds on the runtime for the instances of $size = 40$, $Cs = 1$, and $(pos, neg) \in \{(5, 95), (25, 75), (50, 50), (75, 25), (95, 5)\}$. For both pb and nb we consider values in $\{0, 20, 40, 60, 80, 100\}$, where $pb = 0$ means that the positive constraint is unbounded, and $nb = 0$ means that all paths are denied. While there is not much change when varying nb , we observe that the instances tend to get harder when increasing pb .

6.2 Performance of the different LBRC versions

In this section we consider the different versions of the length-bound reachability constraint: LBRC, LBRC+TC, and LBRC+Dom. We created 100 graphs of 40 vertices, and for each graph we generated an instance of this class (i.e., where $Cs = 1$, and $(pos, neg) = (95, 5)$) without length bounds. Figure 2 shows one of these instances. As the purpose is to assess the additional pruning obtained by the use of explicit transitive closure and dominators, we focus on the decision problem. The decision variables are the be variables as the determination of these variables fully determines the other variables. For each of the approaches we are considering two variable orderings: setting the variable to its minimum value in the domain first (*min*), and setting the variable to its maximum value in the domain first (*max*). Notice that *min* corresponds to excluding the corresponding edge from the set of edges of the output graph and *max* to adding it to the set. In what follows we refer to the approaches obtained when considering the variable orderings as LBRCmin, LBRCmax, LBRCtcMin, LBRCtcMax, LBRCdomMin and LBRCdomMax.

The results of the tests are shown in Figure 6, where Figure 6a refers to the running time, Figure 6b to the number of failures, and Figure 6c to the number of instances that the approach could not solve. The first thing to remark is the gain in pruning when we use LBRC+TC and LBRC+Dom, which positively affects the running time. We also observe that LBRC and LBRC+TC are more sensitive to the variable ordering than LBRC+Dom. Better results are observed with *max* when using LBRC+TC and LBRC+Dom. We believe

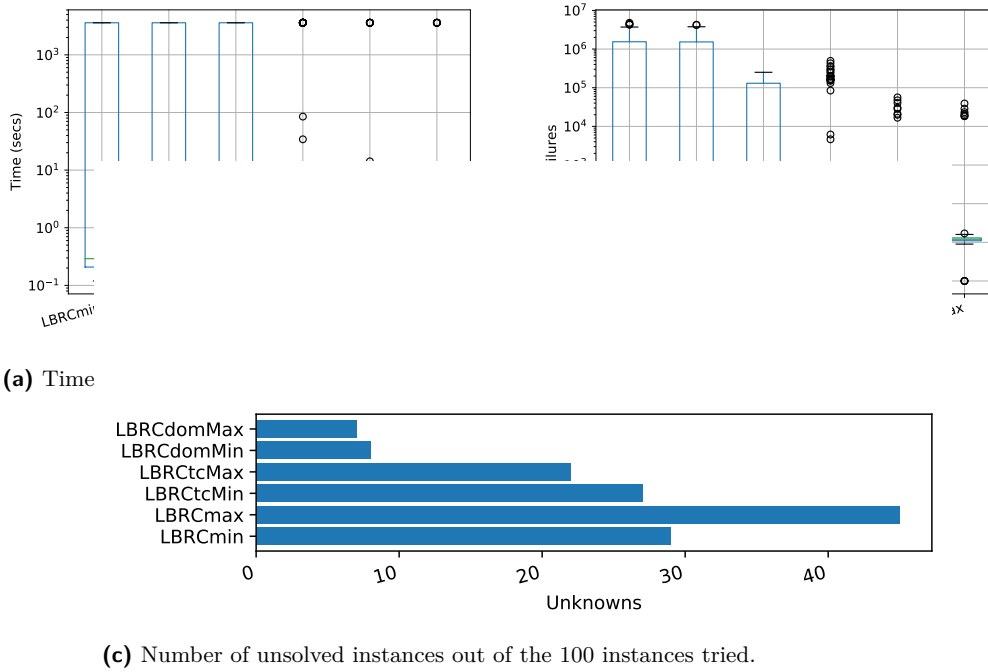
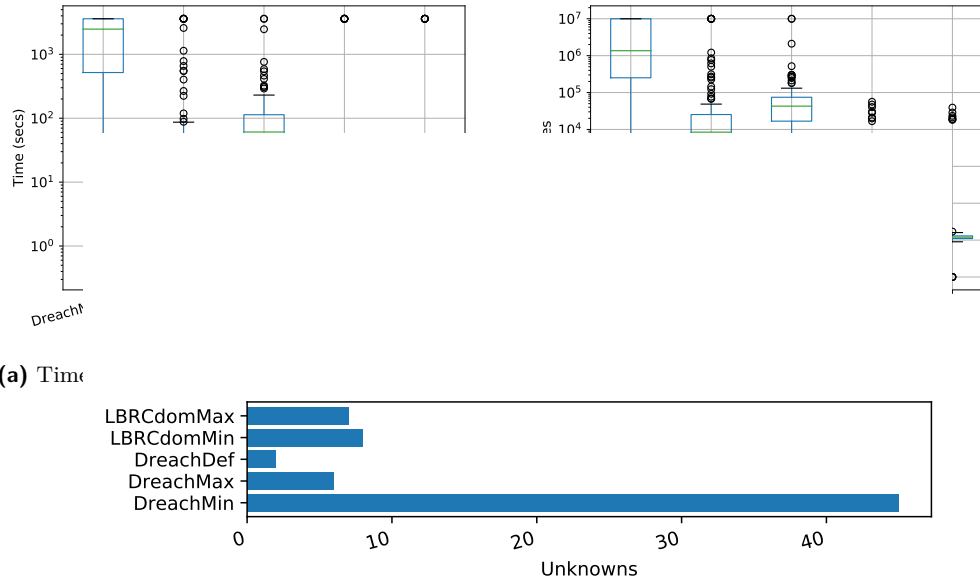


Figure 6 Comparing different versions of the length-bound reachability constraint. The box plots in Figures 6a and 6b show median, inter-quartile range (IQR), bounds of $\pm 1.5 \cdot \text{IQR}$ beyond the box, and outliers, using the `DataFrame.boxplot` function of Pandas[18].

this is because there is more opportunity for the transitive closure rule to play a role as we have more edges. As LBRC does not have the transitive closure pruning, adding edges first actually leads to poorer performance as there are more chances of getting trapped in unsatisfiable cases that are easily detectable by the transitive closure rule. When comparing the performance of the approaches using *min* we see that LBRCtcMin is almost as good as LBRCmin and LBRCdomMin performs much better than both of them, in particular if we look at the number of unsolved instances (see Figure 6c). The advantage of reasoning about dominators is clear, since vertices may become dominators when we take the decision of removing an edge. As discussed in the next section, *min* is useful when optimising the sum of the cost of the edges of the output graph, so it is important to perform well with *min*. Similarly, *max* brings us closer to the optimal solution when minimising the sum of the costs of the paths since the more edges the more chances to connect vertices through the shortest paths. On the easy instances, LBRC perform better than the other two approaches since the overhead of the transitive closure and dominator pruning is not justified. As LBRC+Dom is the best version of the length-bound reachability constraint, we restrict attention to this version for the remaining experiments.

6.3 LBRC+Dom vs Dreachable

We now compare our LBRC+Dom approach against the Dreachable approach of Section 5.



(c) Number of unsolved instances out of the 100 instances tried.

Figure 7 LBRC+Dom vs Dreachable - Decision benchmarks on positive and negative reachability constraints.

6.3.1 Positive and Negative Reachability Constraints

We revisit the instances from Section 6.2. We consider both the decision problem (see Figure 7) and the minimisation of the sum of the costs of the edges of the output graph (see Figure 8). In these experiments we also considered the default search strategy for Chuffed (DreachDef), which is not documented, but led to better results.

The results of the decision problem show that our LBRC+Dom approach is able to find solution faster because we are failing much less. Even though [4] states that dominators are used in the implementation of the Dreachable constraints, it is not clear that is happening in the version provided by Minizinc. Still it is important to mention that DreachMax did manage to close slightly more instances than both LBRCdomMin and LBRCdomMax (see Figure 7c).

The results of the optimisation problem follow the same trend observed in the decision case with respect to the number of failures. However, both approaches ended timing out in most of the cases. Despite that, LBRC+Dom approach managed to get better costs in general. This was mostly due to the good performance observed when using *min*, which tends to minimise the sum of the costs of the edges by selecting fewer edges. It is important to note that in Chuffed we are using both restarts and nogood learning, while in Gecode we have disabled those options. We expect to improve our results even further when considering these options in Gecode.

6.3.2 Positive Reachability Constraints

As mentioned before, if there is no negative reachability constraint, satisfying a set of positive reachability constraints is straightforward. However, if there is an upper bound on the sum of the costs of the selected edges, the problem is NP-complete [1]. This makes the corresponding optimisation problem, i.e., satisfying the set of positive reachability constraints while minimising the sum of the costs of the selected edges, challenging.

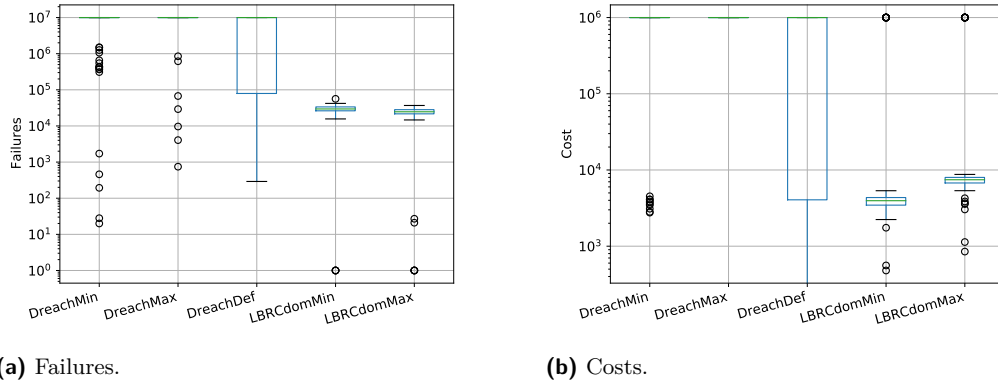


Figure 8 LBRC+Dom vs Dreachable – Optimisation benchmarks on positive and negative reachability constraints.

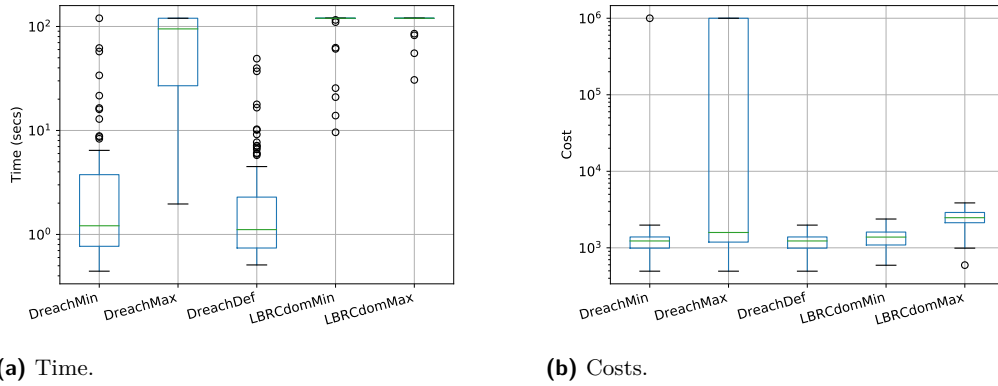


Figure 9 LBRC+Dom vs Dreachable – Optimisation benchmarks on positive reachability constraints.

We have created a new set of instances following the same procedure as above, except instead of randomly selecting a set of pairs, we follow the approach of [1] and randomly select a subset of vertices that are to be fully connected, i.e., for each pair of vertices in this set there should be a path in both directions. Figure 9 shows the results for 100 graphs of 20 vertices. In each case we randomly selected 8 vertices to be fully connected.

The LBRC+Dom approach is outperformed by the Dreachable approach in these instances. Not only does it prove optimality for most of the instances, but it has significantly lower runtime. The main issue with the LBRC+Dom approach has to do with proving optimality. As it can be observed in Figure 9b, the costs obtained by the LBRC approach are very close to the ones obtained by the Dreachable approach but it spends most of the time trying to prove optimality.

7 Conclusions and Future Work

Many practical applications impose positive and negative constraints on reachability, further enhanced with upper and lower bound on the minimum cost paths between pairs of nodes. We have shown that the interaction between positive and negative reachability constraints

leads to complex problems. We have proposed three approaches to modelling these problem as a global constraint on graph variables, which differ on the level of pruning achieved, and demonstrated empirically that the additional pruning plays a key role in solving the problems. We have also studied the dependency between the level of pruning and the search strategy and concluded that the convenience of the search strategy depends on the level of pruning. We have compared our best approach with an existing state of the art approach and shown that when both positive and negative reachability constraints are present, our best approach, incorporating propagation on the transitive closure and dominators, allows significantly lower runtimes, and significantly lower costs for time-limited solving. On the other hand, for problems with only positive reachability constraints, the existing Dreachable constraint is significantly faster.

We believe the improvement offered by the new constraint can be increased by incorporating nogood learning techniques and restarts. Our primary focus in this paper has been on the interaction of positive and negative reachability constraints. Future work will focus on pruning rules to get tighter bounds for the cost, to make the new constraint more competitive in cases where the complexity is driven by the bound on the cost.

References

- 1 Christian Bessiere, Emmanuel Hebrard, George Katsirelos, and Toby Walsh. Reasoning about connectivity constraints. 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 2568–2574. AAAI Press, 2015. URL: <http://ijcai.org/Abstract/15/364>.
- 2 Geoffrey Chu. *Improving combinatorial optimization*. PhD thesis, University of Melbourne, Australia, 2011. URL: <http://hdl.handle.net/11343/36679>.
- 3 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- 4 Diego de Uña. *Discrete optimization over graph problems*. PhD thesis, University of Melbourne, Parkville, Victoria, Australia, 2018. URL: <http://hdl.handle.net/11343/217321>.
- 5 Diego de Uña, Graeme Gange, Peter Schachte, and Peter J. Stuckey. A bounded path propagator on directed graphs. 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 189–206. Springer, 2016. doi:10.1007/978-3-319-44953-1_13.
- 6 Bistra Dilkina and Carla P. Gomes. Solving connected subgraph problems in wildlife conservation. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, volume 6140 of *Lecture Notes in Computer Science*, pages 102–116. Springer, 2010. doi:10.1007/978-3-642-13520-0_14.
- 7 Grégoire Doms. *The CP(Graph) computation domain in constraint programming*. PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2006. URL: <http://hdl.handle.net/2078.1/107275>.
- 8 Jean-Guillaume Fages and Xavier Lorca. Revisiting the tree constraint. In Jimmy Ho-Man Lee, editor, *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 271–285. Springer, 2011. doi:10.1007/978-3-642-23786-7_22.
- 9 Department for Communities and Local Government. *Fire safety risk assessment: offices and shops*. The Stationery Office, UK, 2006.

- 10 Markus Friedrich. Functional structuring of road networks. *Transportation Research Procedia*, 25:568–581, 2017. World Conference on Transport Research - WCTR 2016 Shanghai. 10-15 July 2016. doi:10.1016/j.trpro.2017.05.439.
- 11 M.L. Garcia. *Design and Evaluation of Physical Protection Systems*. Elsevier Science, 2007.
- 12 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- 13 Gecode Team. Gecode: Generic constraint development environment, 2019. Available from <http://www.gecode.org>.
- 14 Alireza Karduni, Amirhassan Kermanshah, and Sybil Derrible. A protocol to convert spatial polyline data to network formats and applications to world urban road networks. *Scientific Data*, 3(1):160046, 2016. doi:10.1038/sdata.2016.46.
- 15 Petr Kolman and Ondrej Pangrác. On the complexity of paths avoiding forbidden pairs. *Discret. Appl. Math.*, 157(13):2871–2876, 2009. doi:10.1016/j.dam.2009.03.018.
- 16 Seán Óg Murphy, Liam O’Toole, Luis Quesada, Kenneth N. Brown, and Cormac J. Sreenan. Ambient access control for smart spaces: dynamic guidance and zone configuration. In *The 12th International Conference on Ambient Systems, Networks and Technologies (ANT), March 23-26, 2021, Warsaw, Poland*, pages 330–337, 2021.
- 17 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.
- 18 The pandas development team. pandas-dev/pandas: Pandas, 2020. doi:10.5281/zenodo.3509134.
- 19 Liliana Pasquale, Carlo Ghezzi, Edoardo Pasi, Christos Tsigkanos, Menouer Boubekeur, Blanca Florentino-Liaño, Tarik Hadzic, and Bashar Nuseibeh. Topology-aware access control of smart spaces. *Computer*, 50(7):54–63, 2017.
- 20 Luis Quesada. *Solving Constrained Graph Problems using Reachability Constraints based on Transitive Closure and Dominators*. PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2006.
- 21 Luis Quesada, Peter Van Roy, Yves Deville, and Raphaël Collet. Using dominators for solving constrained path problems. In Pascal Van Hentenryck, editor, *Practical Aspects of Declarative Languages, 8th International Symposium, PADL 2006, Charleston, SC, USA, January 9-10, 2006, Proceedings*, volume 3819 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 2006. doi:10.1007/11603023_6.
- 22 Meinolf Sellmann. Cost-based filtering for shorter path constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 694–708. Springer, 2003. doi:10.1007/978-3-540-45193-8_47.

Evaluating the Hardness of SAT Instances Using Evolutionary Optimization Algorithms

Alexander Semenov ✉

ITMO University, St. Petersburg, Russia

Daniil Chivilikhin ✉

ITMO University, St. Petersburg, Russia

Artem Pavlenko ✉

ITMO University, St. Petersburg, Russia

JetBrains Research, St. Petersburg, Russia

Ilya Otpuschennikov ✉

ISDCT SB RAS, Irkutsk, Russia

Vladimir Ulyantsev ✉

ITMO University, St. Petersburg, Russia

Alexey Ignatiev ✉

Monash University, Melbourne, Australia

Abstract

Propositional satisfiability (SAT) solvers are deemed to be among the most efficient reasoners, which have been successfully used in a wide range of practical applications. As this contrasts the well-known NP-completeness of SAT, a number of attempts have been made in the recent past to assess the hardness of propositional formulas in conjunctive normal form (CNF). The present paper proposes a CNF formula hardness measure which is close in conceptual meaning to the one based on Backdoor set notion: in both cases some subset B of variables in a CNF formula is used to define the hardness of the formula w.r.t. this set. In contrast to the backdoor measure, the new measure does not demand the polynomial decidability of CNF formulas obtained when substituting assignments of variables from B to the original formula. To estimate this measure the paper suggests an adaptive (ε, δ) -approximation probabilistic algorithm. The problem of looking for the subset of variables which provides the minimal hardness value is reduced to optimization of a pseudo-Boolean black-box function. We apply evolutionary algorithms to this problem and demonstrate applicability of proposed notions and techniques to tests from several families of unsatisfiable CNF formulas.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Hardware → Theorem proving and SAT solving; Theory of computation → Optimization with randomized search heuristics; Mathematics of computing → Combinatorial optimization

Keywords and phrases SAT solving, Boolean formula hardness, Backdoors, Evolutionary algorithms

Digital Object Identifier 10.4230/LIPIcs.CP.2021.47

Supplementary Material *Software (Source Code)*: <https://github.com/ctlab/EvoGuess>

Funding This work was supported by the Ministry of Science and Higher Education of Russian Federation, research project no. 075-03-2020-139/2 (goszadanie no. 2019-1339). Ilya Otpuschennikov's research was funded by Ministry of Science and Higher Education of Russian Federation, project with no. of state registration: 121041300065-9. Artem Pavlenko was supported by JetBrains Research.

1 Introduction

Modern Boolean Satisfiability Problem (SAT) solving algorithms are de-facto a standard computational instrument used in many application domains including symbolic verification, software testing, bioinformatics, combinatorics, and cryptanalysis [10]. SAT solvers work with Boolean formulas, most often written in Conjunctive Normal Form (CNF). If determining



© Alexander Semenov, Daniil Chivilikhin, Artem Pavlenko, Ilya Otpuschennikov, Vladimir Ulyantsev, and Alexey Ignatiev;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 47; pp. 47:1–47:18



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

satisfiability of a CNF formula takes a SAT solver more than a preallocated amount of time, a natural question to ask is how *hard* this formula is for this specific solver? Hereinafter, it is convenient for us to follow the notation of [2] and use the concept of *hardness* of a SAT instance.

For some SAT solving algorithms and some families of formulas their hardness can be estimated analytically [1, 6, 7, 13, 15, 31, 55, 56]. However, to our best knowledge, the general case of the problem of estimating the hardness of a formula w.r.t. to practical SAT solving algorithms is yet to be resolved. The main reason for this is that a state-of-the-art SAT solver is a complicated piece of software, whose behavior depends on a vast number of various parameters [33, 34]. Smallest changes of parameter values may drastically affect the observable characteristics of the SAT solving process, e.g., the number of unit propagations or backtracks. This phenomenon is known as *heavy-tailed behavior* [30]. If a SAT solver demonstrates such behavior for a concrete CNF formula, then estimating how hard the formula is for this solver is hardly feasible with the existing methods, or such estimates will be extremely inaccurate. In light of the myriads of practical applications of modern SAT solvers, it is of unquestionable importance to propose a universal hardness measure for arbitrary CNF formulas that could be used in practice for any SAT solver.

Prior work proposed a few hardness measures of Boolean formulas w.r.t. specific SAT solving algorithms. One of the best-studied approaches estimates parameters of the search tree generated by the algorithm. This class of measures includes *space complexity of tree-like resolution* [27, 40], *width of formula* [27], and *space of formula* [2]. For some families of CNF formulas, e.g. pigeonhole principle formulas [18], such measures may be estimated analytically. However and as far as we know, there is no computationally efficient way of estimating any of said measures for an arbitrary CNF formula.

Another approach to estimating formula hardness builds on the concept of a *strong backdoor set* (SBS) introduced in [59]. An SBS is a subset of the set of variables of a CNF formula such that any assignment of the variables from this subset makes the whole formula polynomially decidable. Clearly, the set of all variables in the formula comprises a trivial SBS. If for formula C there exists a non-trivial SBS B w.r.t. some polynomial-time algorithm P , e.g. unit propagation [24], then the hardness of this formula w.r.t. B and P may be estimated as $\text{poly}(|C|) \cdot 2^{|B|}$. Thus, the notion of SBS gives us a way of estimating the hardness of a formula based on two main components: the strong backdoor set itself and the polynomial-time algorithm used for solving weakened SAT instances. Sadly, for an arbitrary Boolean formula there is no guarantee that a relatively small SBS exists. To check if a given set B of variables in formula C is an SBS, one has to run the algorithm P on all (in the worst case) $2^{|B|}$ CNF instances derived from C by substituting all possible assignments to the variables of B . The algorithm for solving SAT using backdoors described in [59] enumerates all subsets of the set of variables of a target CNF formula by iteratively increasing their size.

In this paper, we propose a novel hardness measure of an arbitrary CNF formula w.r.t. an arbitrary deterministic complete SAT solving algorithm, which may be estimated by applying standard methods of black-box optimization. Conceptually, the suggested hardness measure is in some sense similar to the aforementioned SBS-based hardness measure. For an arbitrary CNF formula over the set X of variables the proposed approach also uses two components: 1) a set $B \in 2^X$, and 2) an arbitrary complete but, most importantly, *not necessarily* polynomial SAT solving algorithm A .

The proposed decomposition hardness (or *d-hardness*) is defined for an arbitrary CNF formula C . More specifically, we first introduce measure $\mu_{B,A}(C)$ expressing the hardness of formula C w.r.t. a concrete set $B \in 2^X$ and a concrete deterministic complete SAT solving

algorithm A . Second, the d-hardness of C is defined as the minimum of $\mu_{B,A}(C)$ over all possible sets B . To estimate $\mu_{B,A}(C)$ in practice, we propose an adaptive probabilistic (ε, δ) -approximation algorithm. This algorithm uses ideas close to the ones from [36], and is based on the Monte Carlo method. But in contrast to many similar approaches, the proposed algorithm can adaptively tune the random sample size to achieve the required accuracy of $\mu_{B,A}(C)$ estimation. By estimating $\mu_{B,A}(C)$ with this algorithm we can reduce the problem of evaluating d-hardness of an arbitrary formula C to optimization of a stochastic pseudo-Boolean fitness function [22]. The latter problem is solved with approaches traditionally used in black-box optimization: namely, evolutionary algorithms [11, 41].

To illustrate the usefulness of the d-hardness concept suppose that we have some extremely hard CNF formula C . Consider the two following approaches. First, launch a SAT solver on C and wait as long as needed to decide satisfiability of C . There is no guarantee that the process will finish in any reasonable amount of time. Second, run algorithms that assess the d-hardness proposed in this article. After a fixed amount of time, say, 12 hours, we will get some set $B \in 2^X$ and a corresponding d-hardness estimate. By means of suggested methods one can compare different CNF formulas in the sense of their d-hardness.

Concrete contributions of this paper are the following.

1. We propose a new measure of hardness for a CNF formula w.r.t. an arbitrary complete deterministic SAT solving algorithm, and prove its theoretical soundness.
2. We develop an adaptive (ε, δ) -approximation algorithm for estimating this measure.
3. We conduct an experimental evaluation that demonstrates practical applicability of the proposed measure.

2 Preliminaries

Recall that Boolean variables have values from $\{0, 1\}$. A Boolean variable x and its negation $\neg x$ are called *literals*. Literals x and $\neg x$ are called *complementary*. A *clause* is a disjunction of literals, which does not include complementary ones. A Boolean formula in CNF is a conjunction of different clauses. Let C be an arbitrary CNF formula and X , $|X| = k$ be the set of variables encountered in C . An arbitrary total function $\alpha: X \rightarrow \{0, 1\}$ defines an *assignment* of variables from X . For an arbitrary assignment α the *interpretation* of formula C on α and the *substitution* of α to C are defined in a standard way, see e.g. [16]. Thus, a Boolean function $f_C: \{0, 1\}^k \rightarrow \{0, 1\}$ is defined. Assignment $\alpha \in \{0, 1\}^k: f_C(\alpha) = 1$ is called a *satisfying assignment* for C . If a satisfying assignment exists for C , formula C is called *satisfiable*. Otherwise, C is called *unsatisfiable*.

As in many other works on proof complexity and hardness of Boolean formulas, formulas are assumed to be in conjunctive normal form (CNF) and unsatisfiable, see e.g. [2, 18, 55], etc. It is justified by the fact that for the majority of satisfiable instances (especially with a large number of satisfying assignments) it is possible that the algorithm will get “lucky” and come across a short satisfiability certificate. This is not possible with unsatisfiable instances.

Let C be an unsatisfiable CNF formula over the set of variables X . For an arbitrary set $B \subseteq X$, denote the set of all possible assignments to variables of B as $\{0, 1\}^{|B|}$. Following [59], for an arbitrary $\beta \in \{0, 1\}^{|B|}$ denote $C[\beta/B]$ the CNF formula derived from C by substitution of the assignment β of variables B and consequent simplification of the resulting formula.

► **Definition 1** (Williams et al. [59]). *Set $B \subseteq X$ is called a strong backdoor set (SBS) for C w.r.t. a polynomial-time algorithm P if for any $\beta \in \{0, 1\}^{|B|}$ the CNF formula $C[\beta/B]$ is reported by P to be unsatisfiable.*

The article [2] studied a number of approaches to estimating hardness of Boolean formulas in CNF, and the main attention was paid to several similar tree-like metrics. However, for us the particular value are the conclusions made in [2] about the possibility to assess the hardness of a CNF formula using SBS. The following definition suggests itself as a consequence of the analysis of results from [2]. In fact, it uses SBS to evaluate the hardness of an arbitrary CNF formula and reduces this problem to an optimization problem.

► **Definition 2 (b-hardness).** *Let C be an arbitrary unsatisfiable CNF formula and B be an arbitrary SBS for C w.r.t. polynomial-time algorithm P . Denote the total runtime of P on CNF formulas $C[\beta/B]$ for all $\beta \in \{0, 1\}^{|B|}$ by $\mu_{B,P}(C)$. The backdoor-hardness (or b-hardness) of C w.r.t. P is specified as $\mu_P(C) = \min_{B \in 2^X} \mu_{B,P}(C)$, where the minimum is taken among all possible SBSes for C w.r.t. P .*

In [59], an algorithm for solving SAT using SBS is described: it enumerates sets $B \in 2^X$ by gradually increasing their cardinality. If for CNF formula C there exists a small-sized SBS, this algorithm may be quite efficient. Its complexity for an arbitrary C in the assumption that an SBS B exists such that $|B| < k/2$ is

$$O \left(p(|C|) \cdot \left(\frac{2k}{\sqrt{|B|}} \right)^{|B|} \right), \quad (1)$$

where $k = |X|$, $p(\cdot)$ is some polynomial and $|C|$ is the length of the binary encoding of C .

3 d-hardness: Decomposition Hardness of CNF Formula

There are two evident barriers for practical application of the b-hardness notion. First, to prove that an arbitrary $B \in 2^X$ is an SBS we have to construct (in the worst case) CNF formulas $C[\beta/B]$ for all $\beta \in \{0, 1\}^{|B|}$. Second, the algorithm of [59] enumerates sets B of increasing cardinality ($|B| = 1, 2, \dots$). Taking into account (1) we can conclude that if, e.g., the minimal backdoor B has cardinality $|B| = 20$ and $k = 100$, finding B with the aforementioned enumeration algorithm is unrealistic. A similar issue arises for the tree-like metric of hardness described in [2], where for formula refutation a variant of Beame-Pitassi algorithm [5] is used.

In this section we introduce a new hardness measure for CNF formulas. When formulating the main concept we pursue the next two goals: 1) to avoid the barriers referred above, and 2) to suggest a measure that can be used for any complete SAT solving algorithm, considering it as a black-box function. Let us begin from the following definition.

► **Definition 3.** *For an arbitrary CNF formula C over the set of variables X consider any set B , $B \in 2^X$, and let A be an arbitrary deterministic complete SAT solving algorithm. Define the hardness of C w.r.t. B and A as $\mu_{B,A}(C) = \sum_{\beta \in \{0,1\}^{|B|}} t_A(C[\beta/B])$, where $t_A(C[\beta/B])$ is the running time of A on CNF formula $C[\beta/B]$.*

The value $t_A(C[\beta/B])$ may be expressed in any appropriate units. For example, if A is a solver based on Conflict-Driven Clause Learning (CDCL) [42], $t_A(C[\beta/B])$ may be defined as the number of unit clause propagations made by A in the process of proving the unsatisfiability of $C[\beta/B]$. Let us emphasize, that unlike P from Definition 2, in the general case A is not a polynomial-time algorithm. The following definition arises by analogy with the concept of b-hardness.

► **Definition 4 (d-hardness).** *The decomposition hardness (or d-hardness) $\mu_A(C)$ of CNF formula C w.r.t algorithm A is defined as:*

$$\mu_A(C) = \min_{B \in 2^X} \mu_{B,A}(C).$$

The main question in the context of these definitions is as follows: is there a practical way to estimate the values $\mu_{B,A}(C)$ and $\mu_A(C)$? Below we give a positive answer to this question harnessing the idea from [50]: expressing $\mu_{B,A}(C)$ via a special random variable with finite expected value and variance.

Let C be an arbitrary CNF formula over the set of variables X and A be an arbitrary deterministic complete SAT solving algorithm. Consider an arbitrary $B \in 2^X$ and specify a uniform distribution on $\{0,1\}^{|B|}$. Define a random variable ξ_B in the following way: for any $\beta \in \{0,1\}^{|B|}$ the value of ξ_B equals to the running time of algorithm A on CNF formula $C[\beta/B]$. Since algorithm A is complete, the random variable ξ_B has spectrum $S(\xi_B) = \{\xi_1, \dots, \xi_M\}$, where $\xi_i: 0 < \xi_i < \infty$, $i \in \{1, \dots, M\}$, and ξ_B has the following probabilistic distribution:

$$P(\xi_B) = \left\{ \frac{s_1}{2^{|B|}}, \dots, \frac{s_M}{2^{|B|}} \right\},$$

where by s_i , $i \in \{1, \dots, M\}$ we denote the number of such $\beta \in \{0,1\}^{|B|}$ that ξ_B has the value ξ_i . From the above, random variable ξ_B has an expected value $E[\xi_B]$: $0 < E[\xi_B] < \infty$. It is not hard to verify the correctness of the following expressions:

$$\sum_{\beta \in \{0,1\}^{|B|}} t_A(C[\beta/B]) = \sum_{i=1}^M \xi_i \cdot s_i = 2^{|B|} \cdot \sum_{i=1}^M \xi_i \cdot \frac{s_i}{2^{|B|}}.$$

From the above, we can conclude that

$$\mu_{B,A}(C) = 2^{|B|} \cdot E[\xi_B]. \quad (2)$$

The equation (2) is quite important because it expresses $\mu_{B,A}(C)$ via finite expected value of some random variable and, hence, allows estimating the value using the Monte Carlo method [43]. In more detail, our nearest goal is to construct such an evaluation $\tilde{\mu}_{B,A}(C)$ of the value $\mu_{B,A}(C)$ that for any fixed $\varepsilon > 0$, $\delta > 0$ the following condition holds:

$$\Pr[(1 - \varepsilon) \cdot \mu_{B,A}(C) \leq \tilde{\mu}_{B,A}(C) \leq (1 + \varepsilon) \cdot \mu_{B,A}(C)] \geq 1 - \delta. \quad (3)$$

Parameters ε and $1 - \delta$ from (3) in a number of similar cases are named *tolerance* and *confidence level*, respectively.

Now, fix some natural number N . Given C , B , and A , let us carry out N independent observations of random variable ξ_B introduced above. We may consider these N observations as one observation of N independent random variables with the same probability distribution (remind, that we assume A to be deterministic). Denote these random variables ξ^1, \dots, ξ^N . Define $\tilde{\mu}_{B,A}(C)$ as:

$$\tilde{\mu}_{B,A}(C) = \frac{2^{|B|}}{N} \cdot \sum_{j=1}^N \xi^j. \quad (4)$$

The sense of the fact that will be established below is close to one of the so-called zero-one estimator theorem from [36], but in our case ξ_B is not a Bernoulli variable and we cannot avoid the presence of $Var(\xi_B)$ in the resulting lower bound for N .

► **Theorem 1.** *Let C be an arbitrary CNF formula over variables X , A be a deterministic complete SAT solving algorithm, and B be an arbitrary subset of X . Then for $\tilde{\mu}_{B,A}(C)$ specified by (4) and for any $\varepsilon > 0$, $\delta > 0$, the condition (3) holds for any $N > \frac{Var(\xi_B)}{\varepsilon^2 \cdot \delta \cdot E^2[\xi_B]}$.*

Proof. Due to the assumptions on A , the random variable ξ_B has a finite expected value $E[\xi_B] > 0$ and finite variance $Var(\xi_B)$. If $Var(\xi_B) = 0$ then $S(\xi_B) = \{a\}$, where a is some constant: $a > 0$. In this case the claim of the theorem is trivially satisfied. Below let us assume that $Var(\xi_B) > 0$. Next we use the Chebyshev's inequality [28]:

$$\Pr \left[|\zeta - E[\zeta]| \leq k \cdot \sqrt{Var(\zeta)} \right] \geq 1 - \frac{1}{k^2} \quad (5)$$

which holds for any $k > 0$ and any arbitrary random variable ζ such that $Var(\zeta) > 0$. Fix an arbitrary $\varepsilon > 0$ and select k such that $k \cdot \sqrt{Var(\zeta)} = \varepsilon \cdot E[\zeta]$. With this in mind, transform (5) to the following form:

$$\Pr [|\zeta - E[\zeta]| \leq \varepsilon \cdot E[\zeta]] \geq 1 - \frac{Var(\zeta)}{\varepsilon^2 \cdot E^2[\zeta]}. \quad (6)$$

Due to considering N independent observations of ξ_B as a single observation of N independent random variables with the same distribution the following holds: $E[\xi_1] = \dots = E[\xi_N] = E[\xi_B]$, $Var(\xi_1) = \dots = Var(\xi_N) = Var(\xi_B)$. Consider the random variable $\zeta = \sum_{j=1}^N \xi_j$. If we apply inequality (6) to it we get (taking into account elementary transformations):

$$\Pr \left[(1 - \varepsilon) \cdot E[\xi_B] \leq \frac{1}{N} \cdot \sum_{j=1}^N \xi_j \leq (1 + \varepsilon) \cdot E[\xi_B] \right] \geq 1 - \frac{Var(\xi_B)}{\varepsilon^2 \cdot N \cdot E^2[\xi_B]}. \quad (7)$$

With respect to (2) and (4), the last inequality may be rewritten as:

$$\Pr [(1 - \varepsilon) \cdot \mu_{B,A}(C) \leq \tilde{\mu}_{B,A}(C) \leq (1 + \varepsilon) \cdot \mu_{B,A}(C)] \geq 1 - \frac{Var(\xi_B)}{\varepsilon^2 \cdot N \cdot E^2[\xi_B]}. \quad (8)$$

The validity of Theorem 1 directly follows from (8). \blacktriangleleft

4 Estimation of d-Hardness via Evolutionary Optimization Algorithms

As follows from the results of the previous section, for exact calculation of d-hardness of an arbitrary CNF formula C it is required to find the set B with the minimum value of $\mu_{B,A}(C)$ over all $B \in 2^X$. For any B , instead of trying out all vectors $\beta \in \{0, 1\}^{|B|}$ as is necessary when we work with the b-hardness concept, we may compute the estimation $\tilde{\mu}_{B,A}(C)$ using the following Monte Carlo scheme:

- let us carry out N independent observations of random variable ξ_B : ξ^1, \dots, ξ^N ;
- calculate the value $\tilde{\mu}_{B,A}(C)$ specified by (4).

Due to Theorem 1, $\tilde{\mu}_{B,A}(C)$ is an (ε, δ) -approximation of $\mu_{B,A}(C)$ for a proper value of N .

4.1 (ε, δ) -approximation algorithm for d-hardness estimation

In theory, since $E[\xi_B]$ and $Var(\xi_B)$ are finite, we can estimate $\mu_{B,A}(C)$ with any accuracy specified beforehand. However, it may be not achievable for real cases: for example, when $Var(\xi_B)$ is too large. Therefore, in experiments when selecting N to achieve the required values of ε and δ we have to replace $E[\xi_B]$ and $Var(\xi_B)$ with their statistical counterparts. This practice is generally accepted in mathematical statistics. In the experimental part we will give a number of examples when the estimates obtained in this way are accurate enough.

Following e.g. [58] we use for estimating $E[\xi_B]$ the sample mean $\bar{\xi}_B$, constructed for a concrete random sample ξ^1, \dots, ξ^N : $\bar{\xi}_B = \frac{1}{N} \cdot \sum_{j=1}^N \xi^j$. The unbiased sample variance is used

to estimate $Var(\xi_B)$: $s^2(\xi_B) = \frac{1}{N-1} \cdot \sum_{j=1}^N (\xi^j - \bar{\xi}_B)^2$. Taking into account Theorem 1, for some fixed ε and δ , we select any such N that the following condition holds:

$$N > \frac{s^2(\xi_B)}{\varepsilon^2 \cdot \delta \cdot (\bar{\xi}_B)^2}. \quad (9)$$

More concretely, we use the following variant of this approach. At the starting point we choose some relatively small N (say, $N = 100$), construct a random sample and calculate $\bar{\xi}_B$ and $s^2(\xi_B)$. Using fixed values of ε and δ (say, $\varepsilon = 0.1$, $\delta = 0.05$) we check if condition (9) is satisfied. If not, we augment our current random sample by N new observations of ξ_B , thus doubling the random sample size; after this we recalculate $\bar{\xi}_B$ and $s^2(\xi_B)$. These steps are repeated until condition (9) is satisfied.

Note that in the general case we cannot efficiently calculate the value of ξ_B (and, accordingly, $\tilde{\mu}_{B,A}(C)$): for example, for B of a small cardinality this problem may be comparable in complexity with solving SAT for the initial CNF formula C . However, most importantly, there always exists such a set B that for any $\beta \in \{0, 1\}^{|B|}$ the corresponding value of ξ_B is calculated efficiently, e.g. in the case when $B = X$. Another example in this context is when B is some Strong Unit Propagation Backdoor Set (SUPBS): a type of backdoor in which the unit propagation rule is used as the polynomial algorithm P [59].

The next important point is that unlike the algorithm from [59] or the Beam-Pitassi algorithm, we apply computational schemes used in metaheuristic optimization [41] to find a set B with a good value of $\tilde{\mu}_{B,A}(C)$. In such schemes, the objective function (*fitness function*) is calculated efficiently at some starting point, and then attempts are made to consistently improve the values of this function in other points of the search space w.r.t. some general search strategy, e.g. local search [14] or evolutionary algorithms [41].

So, in the context of all the concepts introduced above, let $B_0 = \{x_1^0, \dots, x_n^0\}$, $B_0 \subseteq X$ be an initial subset for which $\tilde{\mu}_{B_0,A}(C)$ can be calculated efficiently (e.g. $B_0 = X$ or B_0 is some SUPBS). We will look for B with a good value of $\tilde{\mu}_{B,A}(C)$ as some $B \in 2^{B_0}$. Define B using a Boolean vector $\lambda_B \in \{0, 1\}^n$, assuming that $\lambda_i = 1$ if $x_i^0 \in B$ and $\lambda_i = 0$ if $x_i^0 \notin B$, $\lambda_B = (\lambda_1, \dots, \lambda_n)$. Fix N and consider the multivalued function

$$F_{A,C,N}: \{0, 1\}^n \rightarrow \mathbb{R}_+ \quad (10)$$

defined as follows: for vector $\lambda_B \in \{0, 1\}^n$ we build the set B , then we generate (in accordance with a uniform distribution on $\{0, 1\}^{|B|}$) vectors $\beta_j \in \{0, 1\}^{|B|}$, $j \in \{1, \dots, N\}$ and, using these vectors as a random sample, construct corresponding values of ξ_B : ξ^1, \dots, ξ^N . Then the value of function (10) for λ_B is $\frac{2^{|B|}}{N} \sum_{j=1}^N \xi^j$. Note that in the general case for different random samples the values of (10) can differ, thus this function is multivalued.

4.2 Used evolutionary optimization algorithms

In the experimental part of the article we use evolutionary algorithms for optimizing function (10): in more detail, we apply an algorithm from the family of $(1 + 1)$ Fast Evolutionary Algorithms, $(1 + 1)$ FEA [23] with parameter β , and one special modification of a genetic algorithm. Below we give a brief description of these algorithms.

First, consider the ordinary $(1 + 1)$ Evolutionary Algorithm (EA) [44]. It uses the simplest implementation of the concept of random mutation: one random mutation of an arbitrary $\alpha \in \{0, 1\}^n$ is implemented by a series of n independent Bernoulli trials with success probability $p = 1/n$. If $i \in \{1, \dots, n\}$ is the index of a successful trial, then the i -th bit in α is flipped. The $(1 + 1)$ EA has an extremely high worst-case complexity [25], but demonstrates good results in many practical cases. As mentioned in [57], this is mostly because on average $(1 + 1)$ EA behaves similarly to the Hill Climbing algorithm [49] (for a single random mutation, the expected value of the number of flipped bits equals one), but with a non-zero probability can move from α to any point in $\{0, 1\}^n$.

There are ways of reducing the worst-case estimation of $(1 + 1)$ EA if we imply the complexity measure proposed in [25]. One of these ways is changing the mutation rate in the original $(1 + 1)$ EA. The $(1 + 1)$ FEA $_\beta$ described in [23] is a good example. The core of

this algorithm is the so-called heavy-tailed mutation operator: it flips bits of the considered Boolean vector with probability Λ/n (instead of $1/n$ in standard $(1+1)$ EA), where Λ is the value of a random variable with Power-law distribution $D_{n/2}^\beta$ with parameter β [23]. The worst-case estimation of this algorithm is $O(n^\beta \cdot 2^n)$ instead of n^n for the original $(1+1)$ EA. In computational experiments we used the $(1+1)$ FEA $_\beta$ with parameter $\beta = 3$, because it is the minimal integer value of this parameter for which the expected value of the number of flipped bits tends to some constant with the increase of n : according to [23], this constant is approximately 1.3685.

We also experimented with generating a new vector λ_B on the basis of several existing vectors, using a special variant of a genetic algorithm which was used in [47]. Several vectors λ_B with already calculated values of the considered objective function (10) form a *population* in terms of the genetic algorithm [41]. In one iteration, the new population (offspring) is formed from the current one.

Denote the current population as P_{cur} and the new population as P_{new} , $|P_{\text{cur}}| = |P_{\text{new}}| = R$ for some fixed R . Let $P_{\text{cur}} = \{\lambda_{B_1}, \dots, \lambda_{B_R}\}$. P_{cur} is associated with a distribution $D_{\text{cur}} = \{p_1, \dots, p_R\}$, where

$$p_i = \frac{1/F_{A,C,N}(\lambda_{B_i})}{\sum_{j=1}^R (1/F_{A,C,N}(\lambda_{B_j}))}, i \in \{1, \dots, R\}.$$

To form the new population P_{new} , we first select G individuals from P_{cur} w.r.t. the distribution D_{cur} , and apply the standard two-point crossover [41]. Second, we select H individuals from P_{cur} with respect to the distribution D_{cur} without changes. Finally, we apply to each $G + H$ selected individuals the standard $(1+1)$ random mutation, flipping each bit with probability $1/n$. We ensure $G + H = R$ and compute the value of the objective function for new individuals in P_{new} . Then, we choose R best individuals from $P_{\text{cur}} \cup P_{\text{new}}$, and the resulting set becomes P_{cur} for the next iteration. In the experiments, we used $R = 8$ and $G = 4$.

5 Experimental Evaluation

Here we demonstrate that the proposed approach allows practically estimating the d-hardness of unsatisfiable CNF formulas with sufficiently high precision. As concrete examples, we consider equivalence checking encodings and crafted tests. For the value of $t_A(C[\beta/B])$ we select the number of unit propagations made by algorithm A while solving CNF formula $C[\beta/B]$. This choice, in contrast with using solving time, together with fixing the random seed of the solver, facilitates reproducibility of our results. We also show that sometimes our approach discovers sets B that may be used to solve SAT formulas in parallel with super-linear speedup.

5.1 Benchmarks

We consider two classes of CNF formulas or tests. The first class is comprised of so-called crafted tests. These are synthetic tests, constructed with the aim to generate formulas that are as hard as possible with as few variables as possible.

Quite a few generators of such tests are available. In this work we used the **s_{gen}** generator [54] version 6. Only unsatisfiable instances were generated using **s_{gen}**, instances are denoted **s_{gen}**^{seed}_{#variables}, describing the number of variables in the CNF formula and the random seed used to generate it, e.g. **s_{gen}**¹⁰¹₁₅₀. Search for the set B with the minimal value of function (10) was done on the entire set of variables of the CNF formula.

We also considered a class of tests related to equivalence checking [39]. Consider two Boolean circuits S_1 and S_2 over any complete basis, e.g. $\{\neg, \wedge\}$. We assume that each circuit has n inputs and m outputs. Thus, circuits S_1 and S_2 define functions

$$f_1: \{0, 1\}^n \rightarrow \{0, 1\}^m, f_2: \{0, 1\}^n \rightarrow \{0, 1\}^m$$

respectively. We need to prove that $f_1 \cong f_2$ (pointwise equality), in this case the circuits S_1 and S_2 are equivalent ($S_1 \cong S_2$). It is known [38] that this problem can be efficiently (in time linear of the number of elements in S_1 and S_2) reduced to SAT for a CNF formula C : $S_1 \cong S_2$ if and only if C is unsatisfiable. CNF formula C is constructed from circuits S_1 and S_2 using Tseitin transformations [55]. Bits of vectors from $\{0, 1\}^n$ are encoded with variables forming the set $X^{\text{in}} = \{x_1, \dots, x_n\}$, associated with inputs of S_1 and S_2 .

The CNF formula constructed in this way exhibits the following important property. For a Boolean variable x and an arbitrary $\alpha \in \{0, 1\}$ let us denote by $l_\alpha(x)$ the literal $\neg x$ if $\alpha = 0$ and literal x if $\alpha = 1$. Consider an arbitrary $\alpha = (\alpha_1, \dots, \alpha_n)$, $\alpha_i \in \{0, 1\}$, $i \in \{1, \dots, n\}$ and the following CNF formula:

$$l_{\alpha_1}(x_1) \wedge \dots \wedge l_{\alpha_n}(x_n) \wedge C. \quad (11)$$

It is known (see e.g. [8]) that (un)satisfiability of formula C can be determined by solely applying exhaustive unit propagation to CNF formulas (11) obtained across all possible assignments $\alpha \in \{0, 1\}^{|X^{\text{in}}|}$. In other words, set X^{in} is a SUPBS for C . Then from the above it follows that we can search for B with a good value of $\mu_{B,A}(C)$ among the subsets of X^{in} . For this purpose we will launch a methaheuristic search minimizing the function (10) on the Boolean hypercube $\{0, 1\}^{|X^{\text{in}}|}$.

We applied the described approach to equivalence checking of circuits S_1, S_2 representing two different algorithms which perform sorting of any d l -bit natural numbers. We considered the following sorting algorithms: bubble sorting, selection sorting [20], and pancake sorting [29]. Corresponding SAT encodings can be constructed using any software applied in symbolic verification, e.g. CBMC [17]; in this work we use Transalg [46, 51], which better suits our purposes. We conducted a substantial amount of experiments where equivalence of such circuits was checked. Below we present a few of these results. The SAT instances are denoted by $\text{BvS}_{l,d}$, $\text{BvP}_{l,d}$, and $\text{PvS}_{l,d}$ for Bubble vs Selection, Bubble vs Pancake, and Pancake vs Selection, respectively.

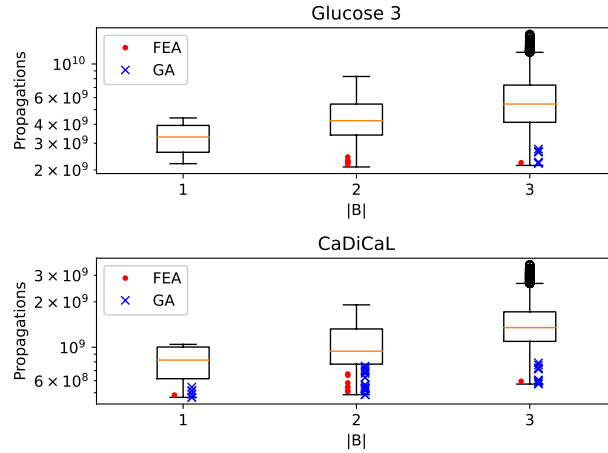
5.2 Experimental setup and implementation details

The proposed approach has been implemented in Python, using PySAT [35] for SAT solving with backend solvers Glucose 3 [3] and CaDiCaL [9], referred to as *g3* and *cd* respectively. The implementation of black-box optimization makes use of distributed computation. Experiments were run on a computing cluster using up to 5 nodes, each node includes two 18-core Intel Xeon E5-2695 2.1 GHz processors and 128 GB of RAM. Each experiment consisted of estimation optimization phase (looking for a set B with minimal estimation value $\tilde{\mu}_{B,A}(C)$) and estimation checking phase (exact calculation of $\mu_{B,A}(C)$). We used the evolutionary algorithms described above for traversing across the search space. We denote the (1+1) FEA₃ algorithm as “FEA”, and the Genetic Algorithm from [47] as “GA”. For calculating the value $\tilde{\mu}_{B,A}(C)$ the adaptive probabilistic (ε, δ) -algorithm presented above was applied. For each B such that $|B| \leq 9$ we directly calculated $\mu_{B,A}(C)$ instead of its estimation $\tilde{\mu}_{B,A}(C)$. In each case the optimization process of function (10) was run with a time limit of 12 hours, using confidence level $1 - \delta = 0.95$. Depending on the concrete CNF formula, we used different values of N ranging from 500 to 40000.

The goal of estimation checking was to assess the efficiency of the decomposition using the found set B . To achieve this we solved instances $C[\beta/B]$ for all $\beta \in \{0, 1\}^{|B|}$ for several described benchmarks (in cases when $|B| \leq 17$), and thus calculated the exact value $\mu_{B,A}(C)$.

5.3 Main experimental results on d-hardness estimation

As mentioned above, we have performed a substantial amount of experiments on different SAT formulas. Here we only report on experiments with tests whose dimensionality allows explicitly checking the precision of resulting d-hardness estimations by exact calculation of $\mu_{B,A}(C)$. In the experiments with formula $\text{PvS}_{4,7}$ we found with our algorithm sets B consisting of three and fewer variables. In order to evaluate the quality of the corresponding decompositions we traversed through all possible sets B of sizes 1, 2, and 3. The corresponding problems are relatively simple, however, to solve them all we used about 3 days of runtime of a single cluster node (36 cores of Intel Xeon E5-2695) in total. Note that finding a set via solving an optimization problem for function (10) took up to 12 hours. The results are presented in Fig. 1 in the form of boxplot diagrams (whiskers span is 1.5 of interquartile range). The lower bound (in the number of propagations) of the diagrams corresponds to the best (smallest) value of function $\mu_{B,A}(C)$ over all possible B : $|B| \in \{1, 2, 3\}$. For several sets with the best values of $\mu_{B,A}(C)$ found by the proposed approach, these values are represented in the diagram: red dots correspond to sets found by FEA and blue crosses to the ones found by GA. Note that in every case our algorithms managed to find a set B for which the value of $\mu_{B,A}(C)$ is between the zeroth and first quartiles of the distribution depicted by the diagram. This proves that our algorithms can find good sets B .



■ **Figure 1** Boxplots for $\mu_{B,A}(C)$ of all sets B for CNF formula $\text{PvS}_{4,7}$, $|B| \leq 3$ and solvers **g3** (top) and **cd** (bottom), and examples of found estimations: our approach allows finding sets B allowing near-optimal hardness estimations.

Table 1 shows experimental results for several SAT instances. For each instance, SAT solver, and evolutionary algorithm, the table shows the cardinality of the found set B , the value $\mu_{B,A}(C)$, and the *decomposition rate* $r_{B,A}(C)$ calculated as $\mu_{B,A}(C)/t_A(C)$. Note that in most cases $r_{B,A}(C)$ is smaller than one, and thus, in these cases the corresponding slicing of formula C using the found set B yields a super-linear speedup when weakened formulas are solved in parallel. Also note that for equivalence checking tests $\text{PvS}_{4,7}$, $\text{BvS}_{4,7}$, $\text{BvP}_{4,7}$ search was done over SUPBSes consisting of $4 \times 7 = 28$ variables. For **sngen** search was done

■ **Table 1** d-hardness estimations for different CNF formulas: most of the found sets B have decomposition rate $r_{B,A}(C) = \mu_{B,A}(C)/t_A(C) < 1$, making it possible to solve the $2^{|B|}$ weakened CNF formulas in parallel with super-linear speedup.

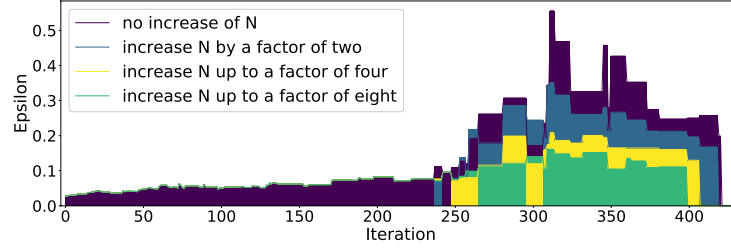
Instance	$ X $	Solver	A	Algorithm	$ B $	$\mu_{B,A}/10^3$	$r_{B,A}$
PvS _{4,7}	3244	g3	FEA	GA	3	2,190,213	0.792
					4	2,250,504	0.814
					5	3,319,314	1.201
					6	3,333,915	1.206
					3	595,695	1.043
sgen ₁₅₀ ¹⁰⁰¹	150	g3	FEA	GA	5	101,371	0.424
					6	244,191	0.763
					6	114,821	0.480
					7	247,947	0.775
sgen ₁₅₀ ¹⁰¹	150	g3	FEA	GA	8	122,796	0.438
					7	131,557	0.470
sgen ₁₅₀ ²⁰⁰	150	g3	GA	FEA	7	151,275	0.569
					6	229,705	0.541
BvS _{4,7}	2134	g3	GA	FEA	3	460,944	1.140
					3	449,325	1.112
BvP _{4,7}	2060	g3	FEA	GA	3	726,080	1.049
					3	771,521	1.115

over the entire set X . Overall, we see that FEA performs slightly better than GA in terms of resulting $r_{B,A}(C)$ values. We can partially explain this by the fact that the GA uses more computational resources in one iteration in comparison with FEA.

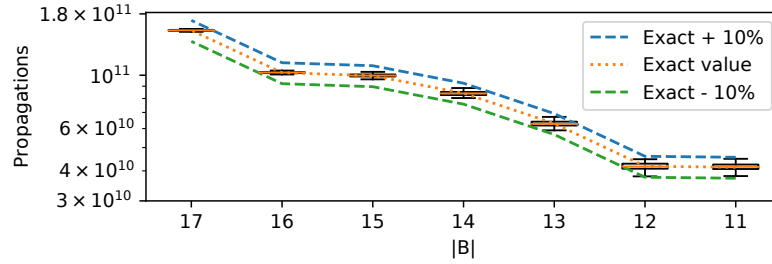
We also performed experiments on searching for non-trivial SUPBSes in the sense of [59] among subsets of X^{in} for the PvS_{4,7} example. Essentially, we implemented a variant of the algorithm from [59], enumerating subsets of X^{in} ($|X^{\text{in}}| = 28$) of gradually increasing cardinality. If for some $B \in 2^{X^{\text{in}}}$ the algorithm found such an assignment $\beta \in \{0, 1\}^{|B|}$ that the application of the unit propagation rule to $C[\beta/B]$ was not enough to decide the satisfiability of $C[\beta/B]$, we concluded that B is not a SUPBS, and switched to the next candidate set B . As a result of these experiments, we have confirmed that for PvS_{4,7} there is no such SUPBS B that $B \subset X^{\text{in}}$ (except X^{in} itself).

In all experiments we used the technique of dynamic adaptation of sample size described in Section 4.1. The plots in Fig. 2 show the dependence of ε on the iteration number for the instance sgen₁₅₀¹⁰⁰¹: the purple plot does not adapt N (the initial value of N is 5000), while the blue, yellow, and green plots may increase N by up to a factor of two, four, and eight respectively. One may notice that the described strategy allows keeping ε below 0.1 most of the time, until finally the set B becomes small enough for switching to direct computation of $\mu_{B,A}(C)$, thus reducing ε to zero.

We also studied the accuracy of our estimation $\tilde{\mu}_{B,A}(C)$ with respect to its exact value $\mu_{B,A}(C)$. For this purpose we considered several intermediate sets B found by our approach for the PvS_{4,7} formula and SAT solver g3. For each B we first calculated $\mu_{B,A}(C)$ by solving all $2^{|B|}$ weakened CNF formulas. Second, we calculated $\tilde{\mu}_{B,A}(C)$ using a sample size $N = \frac{1}{100}2^{|B|}$, and repeated this calculation 100 times with different random samples. The result is a distribution of values of $\tilde{\mu}_{B,A}(C)$. In Fig. 3 we depict these distributions



■ **Figure 2** Dependence of ε from iteration number for $\mathbf{sgen}_{150}^{1001}$ and $\mathbf{g3}$: when N may be increased up to a factor of eight, the value of ε is below 0.1 most of the time.



■ **Figure 3** Accuracy of $\tilde{\mu}_{B,A}(C)$ for $\mathbf{PvS}_{4,7}$ and $\mathbf{g3}$: distributions of estimation values $\tilde{\mu}_{B,A}(C)$ remain within 10% of the exact value $\mu_{B,A}(C)$.

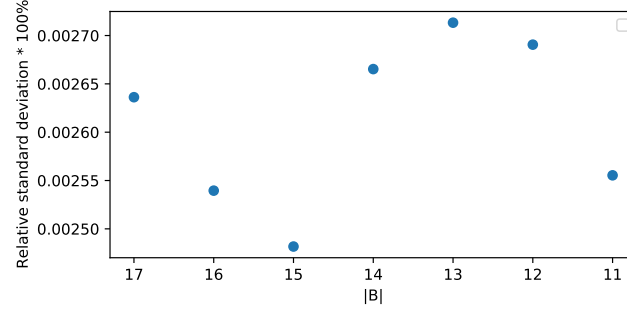
by boxplots for sets with $|B| \in \{17, 16, \dots, 11\}$, in the order they were discovered by the evolutionary algorithm. Fig. 3 also shows with the dotted line the exact value $\mu_{B,A}(C)$ for each backdoor, and the $\pm 10\%$ interval around the exact value with dashed lines. As we can see, the distributions of $\tilde{\mu}_{B,A}(C)$ values remain within 10% of the exact value $\mu_{B,A}(C)$. Also, most importantly, the median value of $\tilde{\mu}_{B,A}(C)$ for each set B is almost exactly equal to the exact value $\mu_{B,A}(C)$ (the dotted line goes through horizontal lines in boxplots that depict the medians). This indicates that the approximation is quite accurate: if for set B the value of $\tilde{\mu}_{B,A}(C)$ is calculated once (as done during the optimization process), there is a high chance that the result will be very close to $\mu_{B,A}(C)$.

5.4 Hardness deviation of weakened CNF formulas

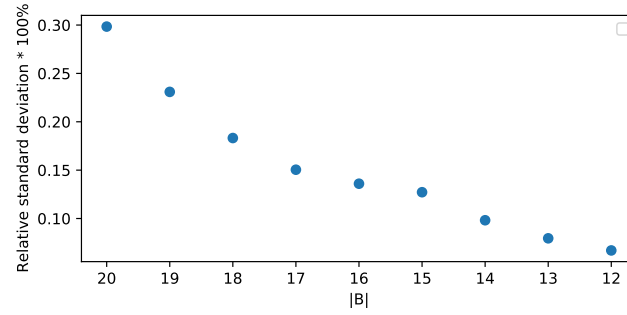
If hardness of weakened formulas differs drastically, one cannot achieve good speedup when solving them in parallel: if, e.g., solving one weakened formula requires, say, 95% of all propagations, then it would not be possible to get even a speedup that is linear in the number of used parallel threads. Thus, in order to use the sets B found by the proposed approach for parallel SAT solving, the corresponding sub-problems (weakened CNF formulas) need to be roughly equally hard. To check if the found sets B have this desired property, we have performed an experimental study regarding the variation of hardness of sub-problems.

More specifically, we measured the relative standard deviation of hardness of all $2^{|B|}$ sub-problems for each set B considered in Fig. 3 for CNF formula $\mathbf{PvS}_{4,7}$ and SAT solver $\mathbf{g3}$, and also for sets B of sizes from 12 to 20 for CNF formula $\mathbf{sgen}_{150}^{200}$ and solver $\mathbf{g3}$. Results are presented in Fig. 4 and Fig. 5 respectively.

As seen from the plots, for $\mathbf{PvS}_{4,7}$ the relative standard deviation of sub-problem hardness does not exceed 0.003%, and for $\mathbf{sgen}_{150}^{200}$ it is within 0.3%. This indicates that for these instances the weakened CNF formulas derived from the corresponding sets B are more or less of equal hardness, so there would be no issues during parallel solving.



■ **Figure 4** Relative standard deviation of sub-problem hardness for several sets B found for $\text{PvS}_{4,7}$ and $g3$.



■ **Figure 5** Relative standard deviation of sub-problem hardness for several sets B found for sgen_{150}^{200} and $g3$.

5.5 Speedup in parallel solving

Here we show some results on solving the original CNF formulas in parallel by means of solving all $2^{|B|}$ weakened formulas derived from the set B generated by our approach. Table 2 shows values of speedup for several CNF formulas and sets B measured for 1..36 parallel threads. The speedup was evaluated as follows. In case of a single thread the speedup is $1/r_{B,A}(C)$, where $r_{B,A}(C)$ is the decomposition rate defined above. In case of $q, q \geq 2$ threads we first accumulated the total number of propagations made by A at each thread. Then we took the maximum value of the number of propagations across all threads and divide $t_A(C)$ by this value to compute the speedup. Thus, in the latter case we take into account the situation, when some threads have finished their work earlier than the others. Note that in the majority of cases, the speedup is indeed super-linear.

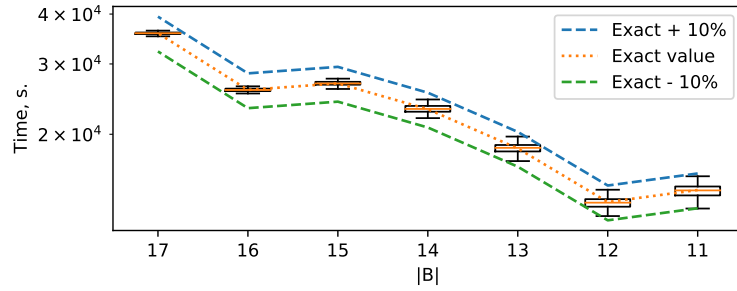
Of course, our approach does not and cannot guarantee that the speedup will be super-linear or even linear: apart from the set B itself, it depends on the properties of the CNF formula, the used strategy of parallel task distribution. However, practical results illustrated in Table 2 give reason to be optimistic.

5.6 Correspondence between the number of unit propagations and solving time

As noted above, in this paper for the value of $t_A(C[\beta/B])$ we select the number of unit propagations made by algorithm A while solving CNF formula $C[\beta/B]$. The reason for choosing this metric instead of just the running time (in seconds) is that the propagations metric is independent of the hardware platform, and the results can be replicated easily.

■ **Table 2** Speedup when using set B to solve weakened CNF formulas on a single core and in parallel (using 2..36 threads).

Instance	$ B $	Solver	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads	36 threads
sgen_{150}^{101}	8	g3	2.3	4.6	8.8	16.8	31.3	37.0	37.0
	13	cd	1.9	3.9	7.7	14.9	29.4	56.9	62.6
sgen_{150}^{200}	8	g3	1.6	3.3	6.2	12.2	22.3	29.9	29.9
sgen_{150}^{200}	8	g3	1.8	3.6	7.1	13.3	25.5	36.2	36.2
	7	g3	2.2	4.4	8.5	15.8	28.0	28.8	28.8
	8	cd	1.3	2.6	5.0	9.6	19.1	22.8	22.8



■ **Figure 6** Accuracy of $\tilde{\mu}_{B,A}(C)$ for $\text{PvS}_{4,7}$ and g3 , where $t_A(C[\beta/B])$ is the running time of the SAT solver in seconds: values of time and unit propagations are sufficiently, though not ideally, correlated.

However, in a practical application we would be interested in sets B that provide a speedup not only in the number of propagations, but also in terms of the running time. Therefore, we replicated results depicted in Fig. 3, measuring $t_A(C)$ and $t_A(C[\beta/B])$ in seconds (for a single thread).

Results are depicted in Fig. 6. Let us compare this plot with Fig. 3. Ideally (if the number of unit propagations exactly correlates with solving time), these plots should be quite the same, except for absolute values of propagations and time. Here, instead, we see that sometimes a decreased value of $\tilde{\mu}_{B,A}(C)$ (and also $\mu_{B,A}(C)$ for that matter) when $t_A(C[\beta/B])$ is measured in propagations corresponds to slightly increased values of $\tilde{\mu}_{B,A}(C)$ and $\mu_{B,A}(C)$ when $t_A(C[\beta/B])$ is measured in seconds: for example, this is the case for pairs $(|B| = 16, |B| = 15)$ and $(|B| = 12, |B| = 11)$. Despite this, the main trends of both plots are the same, indicating that when estimating the decomposition hardness the number of unit propagations can be considered as an adequate deterministic analog of a SAT solver running time.

6 Related Work

There have been a number of attempts to define hardness measures of Boolean formulas. Some of them are purely theoretical, others can be used in practical applications. For the most part, existing works appeal to the peculiarities of specific algorithms and do not consider the SAT solver as a black-box function, as it is done in our approach.

The relationship between the various measures of hardness is demonstrated in [2]. The key motivation for our work was the idea from [2] to determine the hardness of a Boolean formula, starting from the concept of the Backdoor Set introduced in [59]. This measure

(b-hardness) is determined only for Strong Backdoor Sets (SBS) of the function. The value of b-hardness on a particular SBS is equal to the total time required to solve all formulas obtained from the original CNF formula when partitioning it according to this SBS. Also recall that the b-hardness definition implies that weakened formulas are solved in polynomial time.

In our case, unlike [2] and [59], we use an arbitrary set of Boolean variables and an arbitrary (not necessarily polynomial) complete algorithm for solving SAT. In all other aspects our definition of decomposition hardness is similar to the definition of backdoor hardness. Actually, similar ideas have been used to evaluate the effectiveness of SAT Partitionings, mainly as applied to formulas arising in algebraic cryptanalysis: see e.g. [21, 26, 37, 50, 53, 60], etc. However, we emphasize that we are not aware of any works in which these ideas would be used to specify and estimate hardness of CNF formulas in general. Also, none of mentioned papers consider accuracy of obtained estimates or ways of improving this accuracy (such as our (ε, δ) -algorithm).

7 Discussion & Conclusion

Let us emphasize again that for any CNF formula there always exists such a set B that $\tilde{\mu}_{B,A}(C)$ can be calculated efficiently. Thus, for any extremely hard CNF formula we can always obtain some d-hardness estimation. It can be useful in cases when it is necessary to understand whether there is any practical sense in trying to solve the corresponding problem. One can argue that the accuracy of such estimates is questionable (e.g. due to transition from expectation and variance to their statistical counterparts), but our computational results show that they quite often turn to be accurate in practice. Sometimes, obtained preliminary estimates are not precise, but the resulting set B can give a very efficient decomposition (with rate $r_{B,A} < 1$).

As shown above, the cases when $r_{B,A} < 1$ are quite frequent in the studied classes of tests. In such a situation solving all CNF formulas $C[\beta/B]$ is cheaper than solving the original CNF formula without decomposition. Thus, if B has reasonable size, we may use a distributed computational platform to solve all weakened CNF formulas in parallel, and the corresponding speedup will be super-linear.

Recall that in this paper we only addressed hardness estimation for unsatisfiable CNF formulas. In the case of satisfiable formulas our estimation measure does not provide good accuracy guarantees: if a formula has many satisfying assignments, the SAT solving algorithm can get “lucky” or “unlucky”, which would require other estimation measures, e.g. such as the one proposed in [52].

In conclusion, in this paper we have proposed a novel approach to evaluating the hardness of unsatisfiable SAT formulas w.r.t. a deterministic SAT solver. The new hardness measure, d-hardness, is computed w.r.t. a subset B of formula’s variables, and corresponds to the minimal total computation effort needed to solve $2^{|B|}$ weakened CNF formulas across all possible subsets B . To illustrate the practical applicability of the new measure we proposed and developed an adaptive probabilistic (ε, δ) -approximation algorithm based on evolutionary optimization and demonstrated its effectiveness on tests from several families of SAT formulas.

We believe that the concept which lies in the base of the decomposition hardness can be useful in SAT solving strategies aimed at hard SAT instances. In the future, we plan to use the ideas which are close to the ones considered above to estimate the usefulness of cubes in the context of the Cube and Conquer approach [32].

Finally, although the paper argues that decomposition hardness can be effectively estimated with respect to any complete deterministic SAT solving algorithm, the presented experimental study focuses solely on a few SAT solvers based on conflict-driven clause learning (CDCL) [42]. As a result and given that the proof system of CDCL is known to be as strong as general resolution [4, 48], an interesting line of future work will be to extend the proposed ideas to existing algorithms that build on the proof systems strictly stronger than resolution, including cutting planes [19, 45] and dual-rail based MaxSAT [12], among others.

References

- 1 Michael Alekhnovich, Jan Johannsen, Toniann Pitassi, and Alasdair Urquhart. An exponential separation between regular and general resolution. In *STOC*, pages 448–456, 2002.
- 2 Carlos Ansótegui, Maria Luisa Bonet, Jordi Levy, and Felip Manyà. Measuring the hardness of SAT instances. In *AAAI*, pages 222–228, 2008.
- 3 Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving Glucose for incremental SAT solving with assumptions: Application to MUS extraction. In *SAT*, pages 309–317, 2013.
- 4 Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res.*, 22:319–351, 2004.
- 5 Paul Beame and Toniann Pitassi. Simplified and improved resolution lower bounds. In *FOCS*, pages 274–282, 1996.
- 6 Eli Ben-Sasson, Russell Impagliazzo, and Avi Wigderson. Near optimal separation of tree-like and general resolution. *Comb.*, 24(4):585–603, 2004.
- 7 Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow - resolution made simple. *J. ACM*, 48(2):149–169, 2001.
- 8 Christian Bessière, George Katsirelos, Nina Narodytska, and Toby Walsh. Circuit complexity and decompositions of global constraints. In *IJCAI*, pages 412–418, 2009.
- 9 Armin Biere. CaDiCaL at the SAT Race 2019. In *SAT Race*, pages 8–9, 2019.
- 10 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability (Second Edition)*, 2021.
- 11 Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, 2003.
- 12 Maria Luisa Bonet, Sam Buss, Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. Propositional proof systems based on maximum satisfiability. *Artif. Intell.*, 300:pages to appear, 2021.
- 13 Maria Luisa Bonet, Juan Luis Esteban, Nicola Galesi, and Jan Johannsen. Exponential separations between restricted resolution and cutting planes proof systems. In *FOCS*, pages 638–647, 1998.
- 14 Edmund Burke and Graham Kendall. *Search Methodologies*. Springer, 2014.
- 15 Samuel R. Buss and György Turán. Resolution proofs of generalized pigeonhole principles. *Theor. Comput. Sci.*, 62(3):311–317, 1988.
- 16 Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., 1973.
- 17 Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176, 2004.
- 18 Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *J. Symb. Log.*, 44(1):36–50, 1979.
- 19 William J. Cook, Collette R. Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discret. Appl. Math.*, 18(1):25–38, 1987.
- 20 Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- 21 Nicolas T. Courtois and Gregory V. Bard. Algebraic cryptanalysis of the data encryption standard. In *IMACC*, pages 152–169, 2007.

- 22 Guoli Ding, Robert F. Lax, Jianhua Chen, Peter P. Chen, and Brian D. Marx. Transforms of pseudo-boolean random variables. *Discret. Appl. Math.*, 158(1):13–24, 2010.
- 23 Benjamin Doerr, Huu Phuoc Le, Régis Makhlara, and Ta Duy Nguyen. Fast genetic algorithms. In *GECCO*, pages 777–784, 2017.
- 24 William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J. Log. Program.*, 1(3):267–284, 1984.
- 25 Stefan Droste, Thomas Jansen, and Ingo Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theor. Comput. Sci.*, 276(1-2):51–81, 2002.
- 26 Tobias Eibach, Enrico Pilz, and Gunnar Völkel. Attacking bivium using SAT solvers. In *SAT*, pages 63–76, 2008.
- 27 Juan Luis Esteban and Jacobo Torán. Space bounds for resolution. *Inf. Comput.*, 171(1):84–97, 2001.
- 28 William Feller. *An Introduction to probability theory and its applications*, volume 2. John Wiley & Sons, Inc., 2 edition, 1971.
- 29 William H. Gates and Christos H. Papadimitriou. Bounds for sorting by prefix reversal. *Discret. Math.*, 27(1):47–57, 1979.
- 30 Carla P. Gomes and Ashish Sabharwal. Exploiting runtime variation in complete solvers. In *Handbook of Satisfiability (Second Edition)*, pages 463–480. IOS Press, 2021.
- 31 Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39:297–308, 1985.
- 32 Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *HVC*, pages 50–65, 2011.
- 33 Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: An automatic algorithm configuration framework. *J. Artif. Intell. Res.*, 36:267–306, 2009.
- 34 Frank Hutter, Marius Lindauer, Adrian Balint, Sam Bayless, Holger H. Hoos, and Kevin Leyton-Brown. The configurable SAT solver challenge (CSSC). *Artif. Intell.*, 243:1–25, 2017.
- 35 Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.
- 36 Richard M. Karp, Michael Luby, and Neal Madras. Monte-carlo approximation algorithms for enumeration problems. *J. Algorithms*, 10(3):429–448, 1989.
- 37 Stepan Kochemazov and Oleg Zaikin. ALIAS: A modular tool for finding backdoors for SAT. In *SAT*, pages 419–427, 2018.
- 38 Daniel Kroening. Software verification. In *Handbook of Satisfiability (Second Edition)*, pages 791–818, 2021.
- 39 Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *DAC*, pages 263–268, 1997.
- 40 Oliver Kullmann. Upper and lower bounds on the complexity of generalised resolution and generalised constraint satisfaction problems. *Ann. Math. Artif. Intell.*, 40(3-4):303–352, 2004.
- 41 Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013.
- 42 Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability (Second Edition)*, pages 133–182. IOS Press, 2021.
- 43 Nicholas Metropolis and S. Ulam. The Monte Carlo Method. *J. Amer. Statistical Assoc.*, 44(247):335–341, 1949.
- 44 Heinz Mühlenbein. How genetic algorithms really work: Mutation and hillclimbing. In *PPSN*, pages 15–26, 1992.
- 45 Jakob Nordström. On the interplay between proof complexity and SAT solving. *SIGLOG News*, 2(3):19–44, 2015.
- 46 Ilya V. Otpuschennikov, Alexander A. Semenov, Irina Gribanova, Oleg Zaikin, and Stepan Kochemazov. Encoding cryptographic functions to SAT using TRANSALG system. In *ECAI*, pages 1594–1595, 2016.
- 47 Artem Pavlenko, Alexander A. Semenov, and Vladimir Ulyantsev. Evolutionary computation techniques for constructing SAT-based attacks in algebraic cryptanalysis. In *EvoApplications*, pages 237–253, 2019.

- 48 Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011.
- 49 Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.
- 50 Alexander A. Semenov and Oleg Zaikin. Algorithm for finding partitionings of hard variants of boolean satisfiability problem with application to inversion of some cryptographic functions. *SpringerPlus*, 5(1), 2006. Article no. 554.
- 51 Alexander A. Semenov, Ilya V. Otpuschennikov, Irina Gribanova, Oleg Zaikin, and Stepan Kochemazov. Translation of algorithmic descriptions of discrete functions to SAT with applications to cryptanalysis problems. *Log. Methods Comput. Sci.*, 16(1), 2020.
- 52 Alexander A. Semenov, Oleg Zaikin, Ilya V. Otpuschennikov, Stepan Kochemazov, and Alexey Ignatiev. On cryptographic attacks using backdoors for SAT. In *AAAI*, pages 6641–6648, 2018.
- 53 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *SAT*, pages 244–257, 2009.
- 54 Ivor T. A. Spence. Weakening cardinality constraints creates harder satisfiability benchmarks. *ACM J. Exp. Algorithmics*, 20:1.4:1–1.4:14, 2015.
- 55 Grigoriy Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constr. Math. and Math. Logic*, pages 115–125, 1970.
- 56 Alasdair Urquhart. The complexity of propositional proofs. *Bull. Symb. Log.*, 1(4):425–467, 1995.
- 57 Ingo Wegener. Theoretical aspects of evolutionary algorithms. In *ICALP*, pages 64–78, 2001.
- 58 Samuel S. Wilks. *Mathematical statistics*. John Wiley and Sons, 1962.
- 59 Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to typical case complexity. In *IJCAI*, pages 1173–1178, 2003.
- 60 Oleg S. Zaikin and Stepan E. Kochemazov. On black-box optimization in divide-and-conquer SAT solving. *Optimization Methods and Software*, pages 1–25, 2019.

Optimising Training for Service Delivery

Ilankaikone Senthoooran ✉ 

Data Science & AI, Monash University, Clayton, Australia

Pierre Le Bodic ✉ 

Data Science & AI, Monash University, Clayton, Australia

Peter J. Stuckey ✉ 

Data Science & AI, Monash University, Clayton, Australia

Abstract

We study the problem of training a roster of engineers, who are scheduled to respond to service calls that require a set of skills, and where engineers and calls have different locations. Both training an engineer in a skill and sending an engineer to respond a non-local service call incur a cost. Alternatively, a local contractor can be hired. The problem consists in training engineers in skills so that the quality of service (i.e. response time) is maximised and costs are minimised. The problem is hard to solve in practice partly because (1) the value of training an engineer in one skill depends on other training decisions, (2) evaluating training decisions means evaluating the schedules that are now made possible by the new skills, and (3) these schedules must be computed over a long time horizon, otherwise training may not pay off. We show that a monolithic approach to this problem is not practical. Instead, we decompose it into three subproblems, modelled with MiniZinc. This allows us to pick the approach that works best for each subproblem (MIP or CP) and provide good solutions to the problem. Data is provided by a multinational company.

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

Keywords and phrases Scheduling, Task Allocation, Training Optimisation

Digital Object Identifier 10.4230/LIPIcs.CP.2021.48

Acknowledgements We are grateful for our industry partner for this opportunity to work on a challenging real-life workforce planning problem and for the many discussions that have allowed us to conduct this work.

1 Introduction

Large and/or complex machinery and equipment needs regular servicing, so a significant role for companies who maintain such equipment is to schedule engineers to visit customers who have such equipment. Apart from regular servicing, equipment can break down so (emergency) repair visits by engineers also need to be scheduled.

In this work we consider a company that provides services for a wide variety of equipment. Each piece of equipment is complex, and engineers need to be explicitly trained on how to service each piece of equipment. In such circumstances the scheduling problem becomes hard: each engineer is trained to provide services for many, but still a limited subset of, types of machinery. Assigning an engineer to a service call is only possible if they possess all skills (typically few) required by that service call.

In this setting, given an existing roster of engineers and a forecast of service calls over a long horizon, the problem we are tackling consists in strategically deciding which engineers should be trained, and in what skill(s), in order to minimise costs and ensure a short response time. This generates a complex multi-layer decision problem:

- How many engineers do we need for each skill in order to cover demand?



© Ilankaikone Senthoooran, Pierre Le Bodic, and Peter J. Stuckey;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 48; pp. 48:1–48:15

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- Which engineers should be trained with which new skills in order to meet demands?
- And finally, how do we schedule engineers to ensure customers needs are met in a timely manner?

The focus of this paper is on the first two parts of this problem. Of course the efficacy of the first two parts of the problem are only realised if we consider generating actual schedules, in order to see that we are meeting customer demand. The ability to simulate “what-if” scenarios, such as upcoming training opportunities and/or preparing for an anticipated change in demand, at a regular interval (e.g. once a year) allows the service delivery provider to make the best decisions regarding “whom to train” and “on what skill”.

Another aspect of our problem is geographic. The partner we work with is international and has engineers available in different states (in this case, different geographical parts of Australia). Assigning service calls to engineers outside of their home state incurs travel times and costs that can be reduced by training the right engineers in the right skills in the right locations. States have different numbers and types of service activities, each one therefore needs a (possibly empty) tailor-made roster of engineers.

In this paper we give a mathematical model for optimising training for service delivery. We illustrate the model on real-world data provided by our industry partner. The model is broken into three parts to answer the three questions in turn: which skills are required? who should be trained with them? and how does this affect service task scheduling? We show that this separation is able to provide much more scalable, and indeed better solutions, than a monolithic model attempting to answer all three questions at once. We show that the training optimisation can deliver significant savings in comparison to the current assignment processes (with some caveats).

2 The Problem: Training for Service Delivery

From an operational point of view, service calls arise on the fly and engineers must be assigned to these jobs so that the calls can be answered as early as possible. An engineer assigned to a call must possess all the skills required by that job. An engineer is assigned to a job for the duration of the job, and to at most one job at a time. Assigning an engineer to a job outside of their home state incurs an extra fixed travel cost and a per-day living cost. A service call within Australia should not be assigned to an overseas engineer. However, an overseas job can be allocated to any engineer, including overseas engineers.

While we have just described the operational problem as an online scheduling problem, in this paper we consider it to be offline, as what we actually want to determine is how engineers should be trained, which is a strategic decision. In other words, we suppose that all jobs are known in advance. This is a simplification we make 1) because our industrial partner is mostly interested in how to train its workforce, rather than how to schedule its jobs, and 2) because the actual online method used to assign jobs to engineers used by our industry partner is too complex to be replicated, for instance by simulation, and needs to be abstracted to make strategic decisions.

We also consider that training takes no additional time, because for our industrial partner, training a new skill takes a few days and is usually only available at a few times during the year. Hence if from a strategic point of view, we determine that it is valuable for an engineer to learn a given skill, it would outweigh any operational consideration. However, we take into consideration the training cost, which varies based on skill.

3 Modelling and Solving the Problem by Decomposition

Our industrial partner provided historical scheduling data to be used as a projection for future demand. While a straightforward formalisation of this problem into a single model was clearly a possibility, it proved not to be a good one. Indeed, while it is possible to create a model that trains engineers, assigns them to service calls and schedules them into a single monolithic problem that is solved using an off-the-shelf solver, be it a Constraint Programming (CP) or a Mixed-Integer Programming (MIP) solver, this approach would hit two major brick walls. First, as the reader might expect, and as our experiments will show, this is completely impractical due to the combined complexity of the intertwined subproblems. Second, if the model could be solved to optimality, the training decisions that would be taken would overfit the historical data by using the fact that future service requests are completely visible at training time. In reality, future demands are a “fog of war” that cannot be captured by a single scenario. Instead, modern optimisation modelling practice dictates that we would need to obtain or generate more scenarios. It is standard to model uncertainty with chance constraints that need to be satisfied with a certain probability, usually 90 or 95%. See [3] for instance. However, our experiments with a single scenario already fail to produce good solutions in a reasonable time.

To avoid both poor and overfitted solutions, we decompose the problem into three subproblems to reduce the complexity, increase the solving speed, and hide the scheduling decisions from the training decisions. The basic idea is to first forecast future demand using past data, typically over a longer period of time, and determine what skills are in short supply. This is the *capacity planning* subproblem. It is actually a combinatorial problem, not “just” data analysis, as for each skill there may be enough capacity to cover the jobs that require it, but no feasible assignment when taking all skills into account. This serves as an input to the second subproblem, which allocates these new skills to specific engineers, the *skill allocation* subproblem. The second subproblem embeds as a guide a relaxation of the third subproblem, *job scheduling*, which assigns engineers to jobs and schedules them, using their newly-acquired skills, if any. The third subproblem does not decide on training, but it allows us to measure the quality of the solution of the first two subproblems with respect to service quality (i.e. delays to answer calls) and travel costs, which we optimise over in the third subproblem.

To summarise, we solve in sequence the three following subproblems:

- **Capacity planning** identifying skills that are shortage in each state to serve local service calls with the local engineers based on the historical data.
- **Skill allocation** finding whom to train and on what skill to train from the identified set of skills that are in shortage based on the future service calls, allowing travel.
- **Job scheduling** assigning service calls to engineers and scheduling them so as to minimise the overall cost and/or response time.

The subproblems are discussed in detail in the following subsections.

3.1 Capacity Planning

Capacity planning is done for every state individually to identify the skills that are required to serve the local jobs with the local engineers. We look at the past data to identify what skills were in shortage for each state. This is done by matching the available skills with the requirement in a way that minimises the cost to train engineers to meet the shortage. Suppose a state has two engineers, one, say E1, with skills A and B, and other, say E2 with C. There are two jobs: one that requires skill A, and another that requires skill B. Suppose

that both require 5 days. Technically, in this case, both jobs can be performed by the local engineer E1 with skills A and B. However, one job must wait until the other is completed. To minimise the response time, engineer E2 should be trained on either A or B. This decision depends on the training cost of the skills. If A is more expensive than B, E1 will perform the job that requires skill A and E2 will perform the other given training on skill B.

Now, let's say, we have an option to use local contractors to perform the job and the new objective is to minimise the total cost, both training and contractor charges. For the above example, if the cost of the contractor is cheaper than training an engineer, the tool will not suggest any skill training.

In the event where a state has no local engineers or not enough to cater for the demand within the period in consideration, no new skills will be suggested as there are no local engineers to be trained. Naturally, in this case, service calls are made using engineers from other states or by local contractors.

Next, we show the formulation of the capacity planning problem for a state.

3.1.1 Input and Derived Data

Input Index Sets.

- $\mathcal{J} = \{1, \dots, N^{\mathcal{J}}\}$: set of jobs (service calls) that need to be assigned engineers and scheduled.
- $\mathcal{E} = \{1, \dots, N^{\mathcal{E}}\}$: set of engineers / technicians.
- $\mathcal{SK} = \{1, \dots, N^{\mathcal{SK}}\}$: set of skills.
- $\mathcal{S} = \{1, \dots, N^{\mathcal{S}}\}$: set of all states both where jobs need to be served and engineers are located.
- $\mathcal{SK}_e^{eng} \subseteq \mathcal{SK}$: subset of skills that engineer $e \in \mathcal{E}$ has training on
- $\mathcal{E}_s \subseteq \mathcal{E}$: subset of engineers who are located in state $s \in \mathcal{S}$
- $\mathcal{J}_s^{sk} \subseteq \mathcal{J}$: subset of jobs from state $s \in \mathcal{S}$ that require skill $sk \in \mathcal{SK}$

Input Data.

- h^{start} : planning horizon start date
- h^{end} : planning horizon end date
- $wdays$: number of working days within the planning horizon $h^{start} - h^{end}$
- $nsdays$: number of days that becomes available upon training a new skill
- $split \in \mathbb{Z}^+$: number of periods that the planning horizon is split into
- loc_e^{eng} : location (state) of engineer $e \in \mathcal{E}$
- loc_j^{job} : location (state) of job $j \in \mathcal{J}$ needs to be performed
- d_j : duration of job $j \in \mathcal{J}$
- C^{cont} : per day cost to contract a job
- C_{sk}^{train} : cost of training on skill $sk \in \mathcal{SK}$

3.1.2 Decision Variables

- $skshort_{sk}^s \in \mathbb{Z}^+$: number of days that skill $sk \in \mathcal{SK}$ is in shortage in state $s \in \mathcal{S}$ during the planning horizon.
- $sksupp_{sk}^s \in \mathbb{Z}^+$: number of days that skill $sk \in \mathcal{SK}$ is supplied/allocated in state $s \in \mathcal{S}$ during the planning horizon.
- $skreq_{sk}^s \in \{0, 1\}$: 1 if skill $sk \in \mathcal{SK}$ is identified as required skill in state $s \in \mathcal{S}$ during the planning horizon, otherwise 0.

3.1.3 Constraints and Objective Function

Upper bound. on the skill supplied and in shortage is limited to the required amount.

$$skshort_{sk}^s \leq \sum_{j \in \mathcal{J}_s^{sk}} d_j, \quad sk \in \mathcal{SK}, s \in \mathcal{S}. \quad (1)$$

$$sksupp_{sk}^s \leq \sum_{j \in \mathcal{J}_s^{sk}} d_j, \quad sk \in \mathcal{SK}, s \in \mathcal{S}. \quad (2)$$

The amount of skill supplied cannot exceed the number of working days.

$$sksupp_{sk}^s \leq wdays, \quad sk \in \mathcal{SK}, s \in \mathcal{S}. \quad (3)$$

New skill suggestion. Skills that are not required by the jobs within the planning horizon are not suggested.

$$\sum_{j \in \mathcal{J}_s^{sk}} d_j = 0 \rightarrow skreq_{sk}^s = 0, \quad sk \in \mathcal{SK}, s \in \mathcal{S}. \quad (4)$$

Supply and demand. Demand for skill is met by the supply or by contractor (shortage) or by training.

$$\sum_{j \in \mathcal{J}_s^{sk}} d_j \leq sksupp_{sk}^s + skshort_{sk}^s + nsdays \times skreq_{sk}^s, \quad sk \in \mathcal{SK}, s \in \mathcal{S}. \quad (5)$$

Objective function. sum of the cost to contract the jobs that are shortage in skill and to conduct training on suggested new skills.

$$\min \leftarrow obj = C^{cont} \times \sum_{sk \in \mathcal{SK}, s \in \mathcal{S}} skshort_{sk}^s + \sum_{sk \in \mathcal{SK}, s \in \mathcal{S}} C_{sk}^{train} \times skreq_{sk}^s. \quad (6)$$

3.2 Skill Allocation

From the previous step (capacity planning), we know what are the set of skills a state needs to handle the future jobs. So deciding whom to train and on what skill to train them is crucial since the wrong decision might lead to longer response time and higher cost. This step looks at some jobs in hand, preferably for a shorter period, and decides whom to train in the skills found to be in short supply in the previous step for all states simultaneously. Here we allow a job to be allocated to an engineer from a different state than the job's location (state). The objective here is to reduce the cost of training someone and the cost of travel involved in attending jobs that are assigned to an engineer from a different state. The travel includes the return flight cost and the accommodation cost equivalent to the length of the job duration. If a state needs a skill and that state has excess workforce, training someone locally is cheaper than the alternatives, and the algorithm will recommend training a local engineer on that skill. In this case, the local engineer can perform the job that requires the recommended skill.

Next, we present the formulation of the skill allocation problem of a state.

3.2.1 Input and Derived Data

Input Index Sets.

- $\mathcal{SK}_j^{job} \subseteq \mathcal{SK}$: subset of skills that are required to perform job $j \in \mathcal{J}$.
- $\mathcal{E}^{sk} \subseteq \mathcal{E}$: subset of engineers that have training on $sk \in \mathcal{SK}$

Input Data.

- $C_{a,b}^{flight} \in \mathbb{Z}^+, a \neq b$: flight cost of travelling from state $a \in \mathcal{S}$ to state $b \in \mathcal{S}$.
- $C_a^{acco} \in \mathbb{Z}^+$: per day accommodation cost in state $s \in \mathcal{S}$.
- cap^{train} : maximum number of new skills an engineer is allowed to acquire.
- cap^{job} : maximum number of jobs an engineer is allowed to undertake.

Functions. The constraints in our model use the following functions.

- $isOverseasJob(j)$: returns true if job $j \in \mathcal{J}$ needs to be performed overseas, otherwise false.
- $isOverseasEngineer(e)$: returns true if engineer $e \in \mathcal{E}$ is from overseas, otherwise false.

3.2.2 Decision Variables

- $newskills_e \in \mathcal{SK}_e^{eng}$: list of skills suggested/allocated to engineer $e \in \mathcal{E}$.
- $alloc_j \in \mathcal{E}^{sk}, sk \in \mathcal{SK}_j^{job}$: engineer that job $j \in \mathcal{J}$ is assigned to.
- $cost_j^{travel}$: travel cost to perform job $j \in \mathcal{J}$.
- $cost_{sk}^{train}$: cost of training an engineer in skill $sk \in \mathcal{SK}$.

3.2.3 Constraints and Objective Function

Available skills for training are restricted to those identified by the previous solution

$$newskills_e \subseteq \{sk \mid sk \in \mathcal{SK}, skreq_{sk}^s = 1\}, \quad s \in \mathcal{S}, e \in \mathcal{E}_s \quad (7)$$

Limit. on the number of new skill recommendations for an engineer.

$$|newskills_e| \leq cap^{train}, \quad e \in \mathcal{E} \quad (8)$$

A limited number of jobs are assigned to an engineer when deciding the skill recommendation.

$$\sum_{j \in \mathcal{J}} (alloc_j = e) \leq cap^{job}, \quad e \in \mathcal{E} \quad (9)$$

Travel cost. If a job is allocated to an engineer from a different state, apply flight and accommodation cost otherwise set the cost to zero.

$$loc_{alloc_j}^{eng} \neq loc_j^{job} \rightarrow cost_j^{travel} = C_{loc_{alloc_j}^{eng}, loc_j^{job}}^{flight} + C_{loc_j^{job}}^{acco} \times d_j, \quad j \in \mathcal{J} \quad (10)$$

$$loc_{alloc_j}^{eng} = loc_j^{job} \rightarrow cost_j^{travel} = 0, \quad j \in \mathcal{J} \quad (11)$$

Location. An engineer should only be assigned to a job if their existing skill set and the newly recommended skill matches the job skill requirement

$$\mathcal{SK}_j^{job} \subseteq \mathcal{SK}_{alloc_j}^{eng} \cup newskills_{alloc_j}, \quad j \in \mathcal{J} \quad (12)$$

An overseas engineer cannot perform jobs in Australia.

$$\neg isOverseasJob(j) \rightarrow \neg isOverseasEngineer(alloc_j), \quad j \in \mathcal{J} \quad (13)$$

Dominance. To increase the solving efficiency by eliminating the symmetries, we apply a dominance constraint – one that favours skill addition to an engineer with a subset of skills when compared to another engineer’s skills.

$$\mathcal{SK}_{e1}^{eng} \subseteq \mathcal{SK}_{e2}^{eng} \rightarrow |\text{newskills}_{e1}| \geq |\text{newskills}_{e2}|, \quad e1, e2 \in \mathcal{E}, e1 \neq e2. \quad (14)$$

When the maximum number of new skills that an engineer can acquire within the planning period is restricted to one, as is often the case with our partner company, the above constraint is a dominance constraint, it does not remove any optimal solutions. However, when a new engineer can be trained in two or more skills we have no proof that this is a dominance constraint, and instead use it as a rule of thumb to improve solving efficiency.

Objective function. sum of training and travel costs.

$$\min \leftarrow obj = \sum_{e \in \mathcal{E}} \sum_{ns \in \text{newskills}_e} \text{cost}_{ns}^{\text{train}} + \sum_{j \in \mathcal{J}} \text{cost}_j^{\text{travel}}. \quad (15)$$

3.3 Job Scheduling

From the previous step (skill allocation), we know which engineer to train on what skill in order to perform given future jobs. The next step is to assign jobs to engineers while scheduling them. In this step, we assume the engineers have already acquired the training on the recommended skills and have added those skills to their existing skill set. The overall objective is to minimise the total travel cost.

Next, we show the formulation of the job scheduling problem of a state.

3.3.1 Input and Derived Data

Input Index Sets.

- $\mathcal{J}^{sk} \subseteq \mathcal{J}$: subset of jobs that require skill $sk \in \mathcal{SK}$
- \mathcal{C} : set of contractors

Input Data.

- arrivalDate_j : arrival date of job $j \in \mathcal{J}$
- wait_j : preferred wait time of job $j \in \mathcal{J}$
- maxWait : maximum time a job can wait until its been attended since the allowed start date, i.e. $\text{arrivalDate}_j + \text{wait}_j, j \in \mathcal{J}$

3.3.2 Decision Variables

- startDate_j : start date of job $j \in \mathcal{J}$
- $\text{cost}_j^{\text{cont}}$: cost to contract job $j \in \mathcal{J}$.

3.3.3 Constraints and Objective Function

All the constraints listed in step 2, except (a) the constraint on the number of new skills an engineer is allowed to acquire, (b) the dominance constraint and (c) the constraint on the set of possible engineers who can perform a job in the event contractors are permitted, are applied in this step. In addition to these, the following constraints are also enforced:

Non-overlapping. Two jobs that are allocated to the same engineer cannot overlap in time. We use the $\text{disjunctive}(s, d)$ constraint which forces tasks with start times given by array s and durations given by array d not to overlap. We apply this to each engineer e by replacing the duration of tasks that the engineer is not allocated by 0.

$$\text{disjunctive}(\text{startDate}, [\text{if } \text{alloc}_j = e \text{ then } d_j \text{ else } 0 | j \in \mathcal{J}]), \quad e \in \mathcal{E} \quad (16)$$

Limit on overlapping jobs. When contractors are not used, we enforce a redundant constraint to apply a cap on the number of jobs that can overlap, which is equal to the available engineers with the required skill. We use the global $\text{cumulative}(s, d, r, b)$ constraint which forces at each point in time, the total number of tasks (with start times given by array s , durations given by array d and resources required to perform the task given by array r) that overlap that point, does not exceed the limit given by b . We apply this to each skill sk .

$$\text{cumulative}([\text{startDate}_j | j \in \mathcal{J}^{sk}], [d_j | j \in \mathcal{J}^{sk}], [1 | j \in \mathcal{J}^{sk}], \text{card}(\mathcal{E}^{sk})), \quad sk \in \mathcal{SK} \quad (17)$$

Scheduling window. A job must only be scheduled after a set period since its arrival date. Each job depending on its type can have a different wait period.

$$\text{startDate}_j \geq \text{arrivalDate}_j + \text{wait}_j, \quad j \in \mathcal{J} \quad (18)$$

A job must be served within a set period after the wait time.

$$\text{startDate}_j \leq \text{arrivalDate}_j + \text{wait}_j + \text{maxWait}, \quad j \in \mathcal{J} \quad (19)$$

Possible engineers for a job. When contractors are permitted, we allow assigning an engineer with the required skill or a contractor for a job. Here we assume the contractor has the necessary skill and available in every state.

$$\text{alloc}_j \in \mathcal{E}^{sk} \cup \mathcal{C}, sk \in \mathcal{SK}_j^{job}, sk \in \mathcal{SK}, \quad j \in \mathcal{J} \quad (20)$$

Contractor Cost. If a job is allocated to a contractor, apply the associated cost otherwise set the cost to zero.

$$\text{alloc}_j \in \mathcal{C} \rightarrow \text{cost}_j^{\text{cont}} = C^{\text{cont}} \times d_j, \quad j \in \mathcal{J} \quad (21)$$

$$\text{alloc}_j \notin \mathcal{C} \rightarrow \text{cost}_j^{\text{cont}} = 0, \quad j \in \mathcal{J} \quad (22)$$

Objective function. sum of travel and contractor costs.

$$\min \leftarrow \text{obj} = \sum_{j \in \mathcal{J}} \text{cost}_j^{\text{travel}} + \text{cost}_j^{\text{cont}}. \quad (23)$$

3.4 Monolithic Model

The single monolithic problem is essentially modelled by combining all constraints in step 2 and step 3, where the newskills_e are kept as variables so that deciding whom to train on what, allocating jobs to engineers and scheduling are performed together. Equation (7) is omitted so there is no limit on the possible set of new skills that can be trained. The objective is to reduce the sum of training and travelling costs, i.e. Equation (15).

4 Experiment

We have evaluated our algorithm by executing it as a single-thread process on an Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz on actual data provided by an industry partner. The data has approximately 8800 jobs over two years requiring 113 different skills. We are given 53 engineers with a varying skill sets to perform the jobs at hand, and be trained further. The engineers are located in 7 states, including one overseas, while service calls occur in 22 states, including 14 overseas from where 10% of the service calls originate. We split the data into two, the first year (Y1) data is used for capacity planning and the second year (Y2) data is used to perform skill allocation and job scheduling.

We modelled our problem using MiniZinc [15], which allowed us to try different off-the-shelf solvers (CP and MIP) on the same model before deciding on the most suitable one. MiniZinc translates constraints into forms suitable for the chosen solver. For example, in MiniZinc, sets are native. For both CP and MIP solvers, the set constraints, given in Equation (7), are mapped to zero/one representations [2]. To choose an appropriate solver for each step in the decomposed approach and the monolithic model, we ran the preprocessing for each experiment using several solvers (CP and MIP) - Chuffed, Gecode, CBC, and Gurobi. The results were consistent on all occasions and we chose a MIP solver (Gurobi) for steps 1 and 2, and a CP solver (Chuffed) for step 3. For the monolithic model, a CP solver (Chuffed) worked best.

For step 3 and the monolithic model we use a programmed search strategy to find solutions quickly. During scheduling we choose the unscheduled job with the earliest start time and then fix its start time to this earliest possible time and assign an engineer for that job first trying to schedule an engineer who resides in the same location as the job, thus favouring solutions with lower travel costs. We experimented with a number of search strategies and found this to be overall the most robust.

In all the tables provided here, steps 1, 2 and 3 refer to capacity planning, skill allocation and job scheduling stages of the decomposed approach, respectively. Tables 1 shows the results of experiments conducted to compare the performance of the decomposed approach against the monolithic approach for two settings: skill allocation and job scheduling over one-month and two-month periods. For this comparison, we performed the capacity planning using the first-year (Y1) data, and the skill allocation and job scheduling on the second-year (Y2) data. To have a fair comparison, we used the same Y2 data used in the monolithic model for the skill allocation step of the decomposed approach for each period. All the runs had a cutoff time set at 360 seconds except for the monolithic model, which had it set at 3600 seconds. The values shown in the tables indicate the values at the timeout.

The first line of Table 1 shows that determining the skills in short supply is straightforward, we can find an optimal solution in 10 seconds. The results in Table 1 show that while the monolithic model can generate better solutions within its much longer time limit for some of the one month long instances, it scales very poorly, unable to find solutions to one one month problem and any two month problems. Clearly decomposing the problem into 3 parts does restrict the resulting solutions, since an “optimal solution” for the decomposed problem can be bettered by the monolithic model. However, this was when we were looking at a shorter period to decide the skill allocation. From the experimental results presented next, we can see that when skill allocation is performed over an extended period, that is, when looking at the larger problem, the decomposed approach outperforms the monolithic model in terms of solution quality and solving speed. This is true for the all months, including the

cases that were previously bettered by the monolithic model. Overall it is clear that the monolithic model is not practical, even given far more resources and solving a restriction of the problem it cannot compete.

The second experiment, shown in Table 2, tackles a much more practical version of the problem, and is the default problem used in the tool delivered to the industrial partner. Here the decisions of what skills are in short supply and which engineers should be trained on which skills (steps 1 and 2) are both performed over 1 year of data. The table compares three scenarios: when no new skills can be trained, when we can train at most one new skill per engineer, and when we can train at most two new skills per engineer. Clearly given more flexibility of training skills allows us to train more engineers. The second part of the table shows the monthly solutions over the year under the three different scenarios. It considers two different maximum waiting times for jobs. It gives the time to compute the solution, the total wait time over all jobs, the number of job serviced by interstate engineers and the total travel cost, for each month of jobs. For each different wait time it also shows the year sums of the statistics. The headline result is that (perhaps unsurprisingly) enabling new skill training can significantly improve upon the overall costs of providing the service calls. Indeed new skill training is required when we restrict the waiting time, Y2M7 and Y2M8 have no solution with the available engineers and skills. The savings of targeted training are significant reducing overall costs by 30%. Interestingly the more flexible scenario where we can train two skills per person does not always lead to a better overall cost solution. Recall that the training decisions are made on the Y1 data and hence may not be completely reflected in the Y2 data that we actually schedule, and indeed the total number of new skills assigned to engineers only marginally increases. The more flexible scenario does lead to significantly less waiting time for customers though.

Given that in some months we find no viable schedule without using contractors, we extend the step 3 model to allow contractors. This requires a more complex search strategy to obtain good results, but ensures that we find a viable schedule for each month. We applied a search strategy that is similar to the one used in the step 3 model before the extension, which chooses the unscheduled job with the earliest start time and then fix its start time to the earliest possible time and assign an engineer for that job first trying to schedule an engineer who is not a contractor and resides in the same location as the job. The results shown in Table 3 are very similar to those in Table 2 since we try to minimize the use of contractors. The results here do not change the previous conclusions.

5 Related Work

There is extensive literature on the workforce allocation problem [10, 17, 1], including with CP models [12, 14]. A significant part of the literature deals with the problem of assigning *crossed-trained* workers, i.e. trained in multiple skills, to jobs that require a single skill. Although not necessary when each task requires a single skill, cross-training allows the reduction of service delivery delays and increase the utilization of the workforce. A number of papers, among which [4, 5, 6, 13, 18], study the effect of cross-training of the entire workforce as a single varying parameter, but not with decision variables that describe the specific skills that each staff member must learn. This approach is most appropriate when the workforce is large and many staff members have the same skill profile.

■ **Table 1** Performance comparison between decomposed and monolithic approaches. Steps 1,2,3 represents capacity planning, skill allocation, and job scheduling stages, respectively. MM refers to monolithic model. A † indicates a greedy optimal solution for steps 2 and 3 was discovered. The best overall cost solution found of the two approaches is in bold. A – indicates no solution was found within the time limit.

Period	Step	#Jobs	Time	Total Shortage	Total Wait	#New Skills	#Interstate Jobs	Training Cost (x100)	Travel Cost (x100)	Total Cost (x100)
Y1	1	4301	10	759						
1-month, $maxWait = 15$										
Y2M1	2,3	364	8+7		222	8	18	237	1184	†1421
	MM	364	14400		287	8	18	237	1049	1286
Y2M2	2,3	400	8+21		873	7	16	198	1262	†1460
	MM	400	14400		723	7	15	201	1112	1313
Y2M3	2,3	429	10+360		896	6	28	165	1632	1797
	MM	429	14400		690	4	39	107	1883	1990
Y2M4	2,3	393	8+10		414	5	18	140	1389	†1529
	MM	393	14400		522	6	18	140	1127	1267
Y2M5	2,3	449	10+91		816	6	29	165	2017	†2182
	MM	449	14400		750	6	29	165	1825	1990
Y2M6	2,3	380	7+360		863	3	19	84	1889	1973
	MM	380	14400		595	0	31	0	2053	2053
Y2M7	2,3	459	10+360		1353	5	31	140	2480	2620
	MM	459	14400		1202	0	64	0	2878	2878
Y2M8	2,3	370	8+15		638	6	17	171	1487	†1658
	MM		14400		—	—	—	—	—	—
Y2M9	2,3	371	7+10		703	3	15	84	1007	†1091
	MM	371	14400		657	4	15	104	921	1025
Y2M10	2,3	386	8+15		581	6	18	168	1104	†1272
	MM	386	14400		574	7	18	188	1131	1319
Y2M11	2,3	637	14+360		975	8	33	224	1999	2223
	MM	637	14400		1156	0	70	0	2897	2897
Y2M12	2,3	299	6+7		332	4	16	115	1154	†1269
	MM	299	14400		267	4	16	115	1131	1246
2-month, $maxWait = 15$										
Y2M1-2	2,3	812	14+360		2346	9	53	254	3352	3606
	MM		14400		—	—	—	—	—	—
Y2M2-3	2,3	810	21+360		1731	8	43	221	3325	3546
	MM		14400		—	—	—	—	—	—
Y2M3-4	2,3	818	19+360		1475	10	38	274	3373	3647
	MM	818	14400		2429	0	101	0	4534	4534
Y2M4-5	2,3	814	15+360		2015	8	47	218	4631	4849
	MM	814	14400		1651	0	83	0	4815	4815
Y2M5-6	2,3	829	23+360		3079	7	78	196	5471	5667
	MM		14400		—	—	—	—	—	—
Y2M6-7	2,3	814	15+360		3089	9	62	255	4646	4901
	MM		14400		—	—	—	—	—	—
Y2M7-8	2,3	741	16+360		1852	7	41	199	3228	3427
	MM		14400		—	—	—	—	—	—
Y2M8-9	2,3	741	17+360		1572	6	32	168	2030	2198
	MM	741	14400		1628	0	71	0	2832	2832
Y2M9-10	2,3	996	26+360		1755	9	56	260	3507	3767
	MM		14400		—	—	—	—	—	—
Y2M10-11	2,3	637	15+360		975	8	33	224	2010	2234
	MM	637	14400		1138	0	70	0	2897	2897
Y2M11-12	2,3	573	14+33		591	8	22	224	2162	†2386
	MM	573	14400		559	6	27	171	2080	2251

48:12 Optimising Training for Service Delivery

■ **Table 2** Comparison – Effect of allowing 0,1 or 2 skills per engineer on the total cost and changing *maxWait* on the solving time – time out per run 360s. A “—” indicates the problem is unsatisfiable.

			No New Skills			One-Skill Per Person			Two-Skill Per Person					
Period	Step	#Jobs	Time	#New Skills	Training Cost (x100)	Time	#New Skills	Training Cost (x100)	Time	#New Skills (# Engs)	Training Cost (x100)			
Y1	1,2	4301	0	0	0	9+216	19	540	9+467	20 (16)	565			
			Total Wait	#Inter-state Jobs	Travel Cost (x100)	Total Wait	#Inter-state Jobs	Travel Cost (x100)	Total Wait	#Inter-state Jobs	Travel Cost (x100)			
maxWait = 30														
Y2M1	3	364	10	322	43	1740	5	235	13	1047	4	249	12	1038
Y2M2	3	400	21	604	37	1805	10	670	12	1173	12	616	12	1173
Y2M3	3	429	61	1071	49	2229	20	921	19	1305	23	932	19	1305
Y2M4	3	393	28	618	42	1776	9	648	13	1241	11	488	13	1241
Y2M5	3	449	80	881	55	2488	35	933	20	1826	35	833	20	1826
Y2M6	3	380	360	880	30	2146	360	803	14	1753	360	826	14	1753
Y2M7	3	459	360	1716	63	2969	53	1272	21	2015	54	1225	21	2015
Y2M8	3	370	18	826	33	2156	7	627	13	1519	8	568	13	1519
Y2M9	3	371	15	727	26	1181	8	544	10	973	12	6	10	973
Y2M10	3	386	15	634	43	1817	10	418	13	1132	9	427	13	1132
Y2M11	3	637	360	1482	68	2919	294	1295	29	1794	303	1271	26	1695
Y2M12	3	299	7	308	28	1697	3	225	12	1150	3	221	11	1141
Total			10069	517	24923	8591 189 17468			8256 184 17376					
maxWait = 15														
Y2M1	3	364	19	278	43	1750	6	235	13	1047	4	249	12	1038
Y2M2	3	400	18	546	37	1805	15	577	12	1173	14	559	12	1173
Y2M3	3	429	360	731	49	2370	76	681	20	1434	89	772	20	1434
Y2M4	3	393	36	573	42	1776	10	576	13	1241	11	467	13	1241
Y2M5	3	449	75	805	55	2488	41	777	20	1826	40	735	20	1826
Y2M6	3	380	360	565	31	2226	360	761	16	1925	360	755	16	1925
Y2M7	3	459	—	—	—	—	188	1184	21	2026	360	1119	23	2171
Y2M8	3	370	—	—	—	—	14	578	13	1519	13	545	15	1539
Y2M9	3	371	32	740	26	1191	19	648	10	983	32	662	10	983
Y2M10	3	386	85	514	44	1838	9	416	14	1153	8	394	14	1153
Y2M11	3	637	360	1093	71	3115	360	944	31	1938	360	925	29	1844
Y2M12	3	299	5	225	28	1697	3	220	12	1150	4	192	12	1223
Total			6070	426	20256	7597 195 17955			7374 196 18115					

■ **Table 3** Comparison (contractors allowed) – Effect of allowing 0, 1 or 2 skills per engineer on the total cost and changing *maxWait* on the solving time – time out per run 360s.

			No New Skills				One-Skill Per Person				Two-Skill Per Person			
Period	Step	#Jobs	Time	#New Skills	Training Cost (x100)		Time	#New Skills	Training Cost (x100)		Time	#New Skills (# Engs)	Training Cost (x100)	
Y1	1,2	4301	0	0	0		7+523	31	850		7+3297	23 (20)	643	
			Total Wait	#Inter-state, Contract-ed Jobs	Total Cost (x100)		Total Wait	#Inter-state, Contract-ed Jobs	Total Cost (x100)		Total Wait	#Inter-state, Contract-ed Jobs	Total Cost (x100)	
maxWait = 30														
Y2M1	3	364	25	314	36,15	1651	12	177	12,12	1149	360	2059	27,16	2505
Y2M2	3	400	47	592	30,17	1696	29	560	8,14	1080	360	3072	28,20	2407
Y2M3	3	429	92	1054	40,21	2099	57	753	13,17	1194	340	2076	24,23	2973
Y2M4	3	393	44	604	36,14	1683	28	401	8,13	1154	360	2791	33,5	3841
Y2M5	3	449	129	837	47,16	2394	69	791	17,11	1762	360	3895	30,16	4010
Y2M6	3	380	360	861	25,14	2050	360	724	12,10	1675	360	4414	27,24	3667
Y2M7	3	459	360	1635	51,23	2813	152	1518	13,20	1897	202	1629	8,38	2132
Y2M8	3	370	38	754	24,23	2018	24	518	5,22	1387	131	1526	13,23	1814
Y2M9	3	371	30	732	18,17	1079	23	511	3,16	877	360	2669	35,10	2933
Y2M10	3	386	40	623	35,12	1750	17	385	10,6	1096	199	1321	18,15	1595
Y2M11	3	637	360	1487	63,12	2870	200	1117	20,15	1562	30	443	11,13	1916
Y2M12	3	299	14	311	21,8	1650	8	196	8,5	1120	30	443	11,13	1916
Total			9804	426,192	23753		7651	129,171	15953		26338	265,216	31709	
maxWait = 15														
Y2M1	3	364	51	258	36,15	1661	14	177	12,12	1149	8	215	8,12	966
Y2M2	3	400	37	534	30,17	1696	29	523	8,14	1080	23	589	8,14	1080
Y2M3	3	429	360	692	45,13	2341	360	613	14,16	1276	360	722	15,15	1347
Y2M4	3	393	48	579	36,14	1683	39	392	8,13	1154	26	558	8,13	1154
Y2M5	3	449	98	787	47,16	2394	70	714	17,11	1762	67	807	17,11	1762
Y2M6	3	380	360	609	29,7	2179	360	765	15,5	1795	360	718	16,5	1897
Y2M7	3	459	360	1068	53,18	2946	262	1306	13,20	1908	360	1222	21,18	2135
Y2M8	3	370	56	535	23,24	2106	31	506	5,22	1387	76	514	6,22	1469
Y2M9	3	371	44	721	18,18	1084	47	638	3,17	882	27	670	3,17	882
Y2M10	3	386	287	502	36,12	1771	15	320	11,6	1117	13	361	11,6	1117
Y2M11	3	637	360	1140	66,10	3100	360	1028	26,6	1832	360	1042	22,11	1848
Y2M12	3	299	11	237	21,8	1650	7	196	8	1120	6,5	218	7,5	1111
Total			7662	440,172	24611		7178	140,150	16462		7636	142,149	16768	

We focus on papers that propose methods to decide how to both train, allocate jobs and schedule them. De Bruecker et al. [8] specifically review the workforce planning literature that takes skills into account, and in particular (in Section 3.3.3) the papers that allow the workforce to be trained. One important dichotomy is whether the skills are *hierarchical* or *categorical*. With hierarchical skills, there are only skill levels, where a worker at a given skill level can perform all jobs at this or a lower skill level. With categorical skills, there is no comparison between skills.

Huang et al. [11] optimise the service level delivered by a consultancy-type business where projects require certain skills. Since the problem is solved via a discrete event simulator, this allows them to model many different aspects of the problem, such as employees deciding to leave. Within the simulation, the decision to assign staff to projects is made by a Linear Program, hence fractional (simultaneous) assignments are possible. The future horizon is divided into planning periods, during which staff can be trained to ensure that enough capacity of each skill is available for the projects in that period. In this model, each staff can only possess one skill, and training that staff is a “transfer”, i.e. the staff loses the previous skill. Other papers consider that staff can be re-trained, losing their original skills, or transferred between departments [16].

De Bruecker et al. [7] propose a three-step approach to training and scheduling aircraft maintenance teams from one season to the next. Each stage solves a Mixed-Integer Program and feeds into the next step. The first stage consists in scheduling maintenance shifts to ensure the flights can operate on schedule, and assigning workers to the shifts. The second stage refines the set of skills needed by each team of workers in order to reduce the amount of training needed. The third stage attempts to schedule the training needed in the training season.

6 Conclusion

We have demonstrated the potential value of additional training in reducing overall costs for our service delivery problem. Together with the substantial cost reduction, our solutions also provides auxiliary benefits, such as a more highly trained workforce, and considerably less travel, thereby improving staff wellbeing and reducing the company’s carbon footprint.

Furthermore, we have shown that using a single monolithic model to solve the entire problem was not practical, as the solutions it can find in a time similar to our decomposition approach are much worse.

One limitation is our current inability to solve the third subproblem, which schedules jobs and evaluate the quality of our training decisions, for a one-year horizon. We believe that the cost we obtain for the month-by-month approach are a reasonable approximation of the cost that would be obtained for the entire year, but we have not been able to experimentally verify this hypothesis.

Another potential limitation of the current approach is that our third subproblem is set as an offline job scheduling problem. While this integrates well with the first two subproblems, offline scheduling might give a biased measure of the efficiency with which the workforce can deal with jobs that in practice would need to be allocated on the fly. Hence future work includes solving the job scheduling subproblem online, which, since recently, can be modelled in Minizinc using the techniques presented in [9].

References

- 1 Mark Antunes, Vincent Armant, Kenneth N. Brown, Daniel A. Desmond, Guillaume Escamocher, Anne-Marie George, Diarmuid Grimes, Mike O’Keeffe, Yiqing Lin, Barry O’Sullivan, Cemalettin Ozturk, Luis Quesada, Mohamed Siala, Helmut Simonis, and Nic Wilson. Assigning and scheduling service visits in a mixed urban/rural setting. *International Journal on Artificial Intelligence Tools*, 29(03n04):2060007:1–2060007:31, 2020. doi:10.1142/S0218213020600076.
- 2 Gleb Belov, Peter J. Stuckey, Guido Tack, and Mark Wallace. Improved linearization of constraint programming models. In Michel Rueher, editor, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 49–65. Springer, 2016. International Conference on Principles and Practice of Constraint Programming 2016, CP 2016 ; Conference date: 05-09-2016 Through 09-09-2016. doi:10.1007/978-3-319-44953-1_4.
- 3 John R. Birge and François Louveaux. *Introduction to Stochastic programming (2nd edition)*. Springer, New York, NY, 2011.
- 4 Michael J. Brusco. An exact algorithm for a workforce allocation problem with application to an analysis of cross-training policies. *IIE Transactions*, 40(5):495–508, 2008. doi:10.1080/07408170701598124.
- 5 G M Campbell. A two-stage stochastic program for scheduling and allocating cross-trained workers. *Journal of the Operational Research Society*, 62(6):1038–1047, 2011. doi:10.1057/jors.2010.16.
- 6 Gerard M. Campbell. Cross-utilization of workers whose capabilities differ. *Management Science*, 45(5):722–732, 1999. doi:10.1287/mnsc.45.5.722.
- 7 Philippe De Bruecker, Jeroen Beliën, Jorne Van den Bergh, and Erik Demeulemeester. A three-stage mixed integer programming approach for optimizing the skill mix and training schedules for aircraft maintenance. *European Journal of Operational Research*, 267(2):439–452, 2018. doi:10.1016/j.ejor.2017.11.047.
- 8 Philippe De Bruecker, Jorne Van den Bergh, Jeroen Beliën, and Erik Demeulemeester. Workforce planning incorporating skills: State of the art. *European Journal of Operational Research*, 243(1):1–16, 2015. doi:10.1016/j.ejor.2014.10.038.
- 9 Alexander Ek, Maria Garcia de la Banda, Andreas Schutt, Peter J. Stuckey, and Guido Tack. Modelling and solving online optimisation problems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(02):1477–1485, April 2020. doi:10.1609/aaai.v34i02.5506.
- 10 A.T Ernst, H Jiang, M Krishnamoorthy, and D Sier. Staff scheduling and rostering: A review of applications, methods and models. *European Journal of Operational Research*, 153(1):3–27, 2004. Timetabling and Rostering. doi:10.1016/S0377-2217(03)00095-X.
- 11 Huei-Chuen Huang, Loo-Hay Lee, Haiqing Song, and Brian Thomas Eck. Simman—a simulation model for workforce capacity planning. *Computers & Operations Research*, 36(8):2490–2497, 2009. Constraint Programming. doi:10.1016/j.cor.2008.10.003.
- 12 Serdar Kadioglu, Mike Colena, Steven Huberman, and Claire Bagley. Optimizing the cloud service experience using constraint programming. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming*, pages 627–637, Cham, 2015. Springer International Publishing.
- 13 Haitao Li and Keith Womer. Scheduling projects with multi-skilled personnel by a hybrid milp/cp benders decomposition algorithm. *J. Scheduling*, 12:281–298, June 2009. doi:10.1007/s10951-008-0079-3.
- 14 Y. Naveh, Y. Richter, Y. Altshuler, D. L. Gresh, and D. P. Connors. Workforce optimization: Identification and assignment of professional workers using constraint programming. *IBM Journal of Research and Development*, 51(3.4):263–279, 2007. doi:10.1147/rd.513.0263.
- 15 N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 529–543. Springer-Verlag, 2007.

- 16 Haiqing Song and Huei-Chuen Huang. A successive convex approximation method for multistage workforce capacity planning problem with turnover. *European Journal of Operational Research*, 188(1):29–48, 2008. doi:10.1016/j.ejor.2007.04.018.
- 17 Jorne Van den Bergh, Jeroen Beliën, Philippe De Bruecker, Erik Demeulemeester, and Liesje De Boeck. Personnel scheduling: A literature review. *European Journal of Operational Research*, 226(3):367–385, 2013. doi:10.1016/j.ejor.2012.11.029.
- 18 Kum-Khiong Yang, Scott Webster, and Robert A. Ruben. An evaluation of worker cross training and flexible workdays in job shops. *IIE Transactions*, 39(7):735–746, 2007. doi:10.1080/07408170701244687.

Human-Centred Feasibility Restoration

Ilankaikone Senthoooran 

Data Science & AI, Monash University,
Clayton, Australia

Gleb Belov 

Data Science & AI, Monash University,
Clayton, Australia

Kevin Leo 

Data Science & AI, Monash University,
Clayton, Australia

Michael Wybrow 

Human-Centred Computing, Monash University,
Clayton, Australia

Matthias Klapperstueck 

Human-Centred Computing, Monash University,
Clayton, Australia

Tobias Czauderna 

Human-Centred Computing, Monash University,
Clayton, Australia

Mark Wallace 

Data Science & AI, Monash University,
Clayton, Australia

Maria Garcia de la Banda 

Data Science & AI, Monash University,
Clayton, Australia

Abstract

Decision systems for solving real-world combinatorial problems must be able to report infeasibility in such a way that users can understand the reasons behind it, and understand how to modify the problem to restore feasibility. Current methods mainly focus on reporting one or more subsets of the problem constraints that cause infeasibility. Methods that also show users how to restore feasibility tend to be less flexible and/or problem-dependent. We describe a problem-independent approach to feasibility restoration that combines existing techniques from the literature in novel ways to yield meaningful, useful, practical and flexible user support. We evaluate the resulting framework on two real-world applications.

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

Keywords and phrases Combinatorial optimisation, modelling, human-centred, conflict resolution, feasibility restoration, explainable AI, soft constraints

Digital Object Identifier 10.4230/LIPIcs.CP.2021.49

Funding Partly by Australian Research Council grant DP180100151 and Woodside Energy Ltd.

1 Introduction

Finding (high quality) solutions to combinatorial problems is important for our society. This has fuelled research into technologies to model and solve these problems, many of which are now used in decision systems deployed by businesses such as Amazon, Google and HP.

An important, but less researched aspect of these systems is their interaction with human users, particularly when reporting infeasibility created by errors or “what-if” scenarios. While users do need information to restore feasibility, it is not obvious what information is best. Research has mainly focused on finding subsets of the problem constraints responsible for the infeasibility. This has yielded interesting subsets, such as Minimal Unsatisfiable Sets (MUS) and Minimal Correction Subsets (MCS) [20], and enumeration methods to compute them efficiently (e.g., [13, 16, 19, 18, 21]). See [6] for applications of these subsets.

While enumeration methods are a great starting point for explaining infeasibility to users, a straightforward use of these methods is not suitable for real-world systems [10]. We have experienced this repeatedly, most recently in a system that finds high quality 3D layouts for an industrial plant, where better quality solutions can save millions of dollars. We soon realised it is easy for users to create infeasible plants due to incorrect data (e.g., making the



© Ilankaikone Senthoooran, Matthias Klapperstueck, Gleb Belov, Tobias Czauderna, Kevin Leo, Mark Wallace, Michael Wybrow, and Maria Garcia de la Banda;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 49; pp. 49:1–49:18



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

plant too small for its equipment) and/or inconsistent constraints (e.g., setting object A on the ground and also on top of another object B). Since plants contain hundreds of pieces of equipment, multiple inconsistencies are easily introduced. Our attempts to use some of the constraint-independent enumeration methods available [19, 18] to find and resolve these inconsistencies resulted in impractical waiting times and hundreds of MUSes, overwhelming users. Further, these methods did not show how to restore feasibility. Our attempts to use methods that sacrificed generality for speed, and which reported the minimum changes needed to restore feasibility [6, 27], led to users being given a very restricted set of choices.

Our experiences illustrate the need for user support that is *meaningful, useful, practical and flexible*. We say support is meaningful if it expresses the selected constraint subsets in a way that is understandable to users. It is useful if it helps users determine not only what prevents the system from finding a solution, but also how to actually modify the data or constraints to eliminate the inconsistency. It is practical if it is fast enough for users, and it is flexible if it gives them choices regarding how to find and resolve infeasibility.

The few methods that address some of these four needs [17, 8, 4] tend to focus only on one of them and/or are problem-dependent. Our main contributions are (1) to describe a problem-independent approach to feasibility restoration that combines existing qualitative and quantitative techniques in novel ways, yielding meaningful, useful, practical and flexible user support, and (2) to evaluate the trade-offs between practicality and flexibility for the conflict resolution alternatives offered by our approach, as well as their meaningfulness and usefulness, in the context of two real-world applications. In doing this, we also contribute (3) a method to quantify the violation of logical combinations of constraints, and (4) a problem-independent interface for user intervention.

2 Background and Related Work

Explaining infeasibility. Given an unsatisfiable set of constraints C , subset $M \subseteq C$ is a Minimal Unsatisfiable Set (MUS) of C iff M is unsatisfiable and removing any constraint from M makes it satisfiable; and is a Minimal Correction Subset (MCS) of C iff $C \setminus M$ is satisfiable and adding any $c \in M$ to $C \setminus M$ makes it unsatisfiable. Every MCS is a *hitting set* of all MUSes (i.e., has a non-empty intersection with each MUS) and vice-versa. While removing a MUS from C might not make it satisfiable (C may have disjoint MUSes), removing one MCS from C does. Applications often have a subset B of C , called the *background*, that should not appear in the computed subsets. Its complement $C \setminus B$ is the *foreground*. MUSes and MCSes are redefined using B as follows. A *minimal conflict* of C for B is a subset M of the foreground such that $M \cup B$ is unsatisfiable, and for any $M' \subset M$, $M' \cup B$ is satisfiable. A *minimal relaxation* of C for B is a subset M of the foreground such that $(C \setminus M) \cup B$ is satisfiable and for any subset $M' \subset M$, $(C \setminus M') \cup B$ is unsatisfiable. Herein we treat MUS and MCS as synonyms of minimal conflicts and minimal relaxations, respectively.

Finding one MUS. Early techniques to find a MUS are based on linear deletion methods [13], where each constraint in unsatisfiable set C is tentatively removed from it, and is added back to C if its removal yields satisfiability. Once all constraints in the initial C are tested, those in the final C form a MUS. One of the most popular techniques is QuickXplain [16], which reduces the number of satisfiability checks needed by recursively splitting and reducing C to a MUS. These approaches use solvers as satisfiability checkers, without taking into account any properties of C . Other approaches sacrifice such generality for speed by focusing on

particular kinds of constraints, like those in linear programs [26, 12], numerical constraint satisfaction problems [11], and Mixed Integer Programs (MIP) [13], where a MUS is called an Irreducible Inconsistent Subsystem (IIS).

Enumerating MUSes and finding one MCS. Most MUS enumeration techniques gain speed by keeping track of the explored subsets to prune superseded ones. One of the most popular is MARCO [19], which avoids redundant checks of supersets/subsets of known unsatisfiable/satisfiable sets. The aim when enumerating MUSes is often to compute an MCS. Efficient techniques exist to compute MCSes directly (e.g., [20, 9, 21]), rather than as hitting sets of MUSes. MCSes discovered during MUS enumeration can also be used to speed up the enumeration process [1].

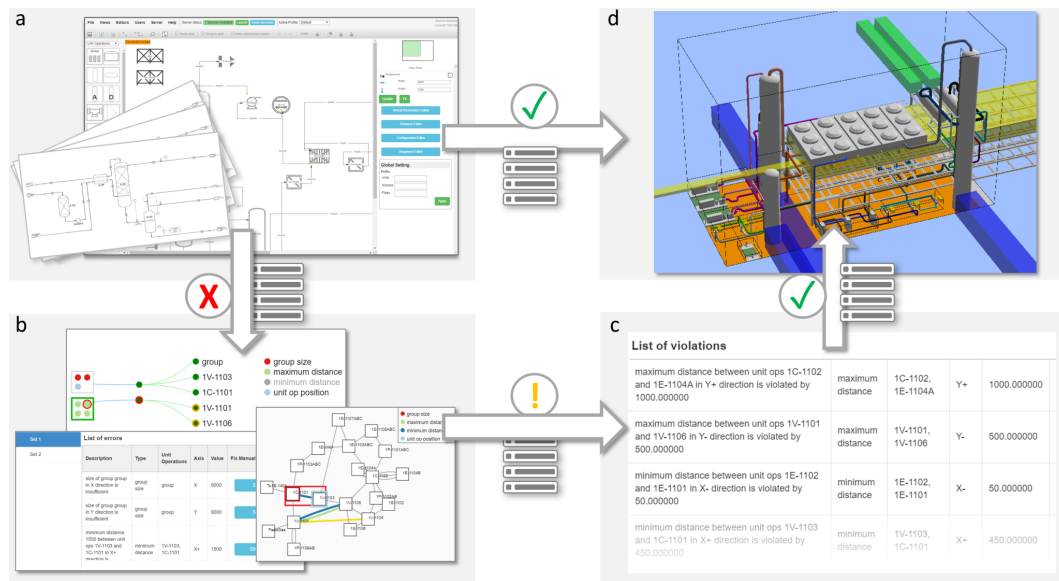
Preferred subsets. The exponential number of MUSes and MCSes, together with the large number of constraints in real-world problems, have yielded methods that allow users to express their preferences [16, 22, 23]. These methods obtain MUSes and MCSes built from less preferred constraints, which can be violated to satisfy the more important ones. In addition, accounting for preferences can help speed up computation, as shown by MiniBrass [25], which generalises several constraint preference schemes by using *partially ordered valuation structures*, and implements them as a soft constraint modelling language.

Human-understandable subsets. The closest work we know of is that of [17], which uses a version of MARCO to iteratively construct minimal conflicts responsible for causing infeasibility, coupled with minimal relaxations which can be used to restore feasibility. Their minimal sets can be expressed in human understandable language and at different levels of abstraction by using a powerset lattice of Boolean variables to represent each constraint in the foreground, which only contains constraints that can be altered by parameters controlled by the users. It is also the only work we know of that tries to eliminate redundancies from the subsets to produce more compact descriptions. In contrast, our method is problem-independent and, as shown later, its novel combination of enumeration and IIS based methods makes it more practical and flexible.

Connecting models to instances. We distinguish between a problem *model*, where the input data is described in terms of parameters, and a particular model *instance*, where the values of all parameters are added to the model. Models are usually defined in a high-level language, such as JUMP [7] or MiniZinc [24], and their instances are compiled into a *flat format*, where loops are unrolled and constraints are transformed into formats suitable for the selected solver. Both the compiler and the solver may introduce new variables and constraints during the flattening/solving process. This makes it difficult to report constraint subsets to the users in a meaningful way. In this paper we use the MiniZinc toolchain, which assigns a unique identifier [18] to each variable and constraint in a flattened instance that links them to the part of the model's source code that generated them.

3 Motivating Example: Plant Layout

The equipment allocation phase of the Plant Layout system of [2] finds the 3D position coordinates and orientations of the specified *equipment* within a given *container space*, that (a) satisfy distance, maintenance and alignment constraints, and (b) minimise the costs of the plant's *footprint*, of the supporting equipment, and of a Manhattan approximation of the connecting pipes. Figure 1(d) shows a possible solution for the plant described in Figure 1(a).



■ **Figure 1** Simplified Plant Layout workflow for a feasible plant ($a \Rightarrow d$) and for feasibility restoration ($a \Rightarrow b \Rightarrow c \Rightarrow d$).

Data. The user interface (UI, Figure 1(a)) provides a predefined palette of equipment templates, where each template belongs to a *class* (e.g., heat exchangers or pumps) and each class has a set of associated constraints. Users can drag and drop equipment from the palette onto the canvas to describe the plant. They can modify the equipment dimensions, alter the positions of the *nozzles* where the pipes attach, and connect equipment via pipes.

Underlying constraints. Each piece of equipment is automatically constrained to (a) be within the container space, (b) not overlap with any other equipment, (c) be positioned in one of four possible orientations, (d) satisfy the min/max distances associated to its class, and (e) satisfy any maintenance access constraints associated to its class (e.g., needs truck access or cannot have equipment below). In addition, some combinations of equipment have extra constraints (e.g., heat exchangers must be symmetrically positioned w.r.t. their connecting vessel). Finally, the model has redundant constraints to speed up solving.

User constraints. The UI allows users to add, remove and modify some of the underlying constraints. In particular, users can (a) modify the min/max distances between equipment classes, (b) add constraints on the relative position/distance of any two objects, (c) add/delete/modify maintenance access for any equipment, (d) ensure objects are positioned within/at a given area/point, (e) provide upper bounds to the container space dimensions, and (f) add group size constraints forcing selected objects to be within a given sized box.

Internal representation. The optimisation model internally treats equipment as boxes, thus ignoring their exact form. It also treats maintenance access as boxes *attached* to equipment in rigid/rotatable form, thus providing access to one or any of its sides (blue, green and yellow boxes in Figure 1(d)). The position of each box is modelled primarily by its front-left-bottom corner coordinates and its orientation. However, many other auxiliary decision variables are used to make it easier to express certain constraints.

Restoring feasibility. It is easy for user constraints to cause infeasibility. For example, the min/max distances or the absolute/relative positioning of equipment, can conflict with the dimensions of the container space, or make it impossible for equipment to be symmetrically positioned. Figure 1(b) shows some of the ways our system helps users restore feasibility, which include displaying the constraints in all MUSes from which users can select an MCS. This MCS is then used to find restoration values for its constraints (Figure 1(c)). Users can then modify these values in the UI and restart the process for a solution (Figure 1(d)).

4 Design Decisions

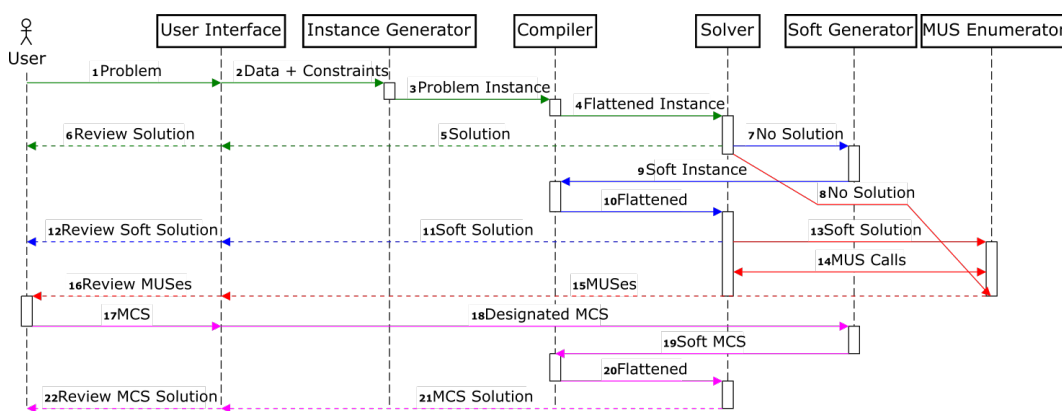
As mentioned before, our aim is to support users not only in understanding the reasons for infeasibility, but also in recovering from it, and do so in a meaningful, useful, practical and flexible way. As with most multi-objective optimisation problems, these objectives often conflict. For example, in order to be useful we would like to find all possible MCS (or minimal relaxation) subsets. However, this can be both impractical and overwhelming to users, thus affecting meaningfulness. The following is a brief summary of our key design decisions.

To be **meaningful** we use MiniZinc’s capability to (a) annotate the model constraints with meaningful names, and (b) connect the variables and constraints seen by the underlying solver (i.e., those in the flattened instance) to those appearing in the problem model. This allows us to present information to the users at the same level of abstraction used by the UI, independently of the underlying technology (see, for example, the constraint names “group size” and “maximum distance” in the infeasibility set reported by Figure 1(b)).

To be **useful** we combine the detection of infeasible sets with a conflict resolution technique where *slack variables* are added to user constraints, transforming them into *soft constraints* [27]. This allows us to tell users not only which constraint subsets are infeasible, but also how the value of their variables can be modified to make them feasible.

To be **practical** we combine time-intensive methods that aim to enumerate all minimal/maximal infeasible/feasible sets, with fast methods that aim to report a single set, possibly not minimal/maximal. By iteratively combining these methods we can provide feedback about infeasibility quickly, while also generating sets that are useful to the users.

To be **flexible** we show users the different ways in which our methods can be executed and allow them to decide what they want reported and how long they can wait for it.



■ **Figure 2** Sequence diagram showing the high-level overview of the method for an application.

5 Method Overview

Figure 2 shows our method as a sequence diagram, where colours correspond to the different restoration methods users can select and the solid/dash outline indicates the right/left direction of the arrow. The first 4 steps (in green) are always followed: the user describes the problem via a UI (step 1), which sends this information to the instance generator (step 2) to generate a MiniZinc instance (step 3) that is flattened by the MiniZinc compiler and sent to the selected solver (step 4). If a (possibly optimal) solution is found, it is sent back to the UI and presented to the user (steps 5 and 6). Otherwise, two paths (blue/red) can be selected. In the blue path the solver engages our soft generator (step 7) to generate a *soft* instance, where the user constraints are relaxed by means of slack variables aimed at quantifying infeasibility (step 9). This soft instance is flattened and sent to the solver (step 10) for a soft solution. Users who want a fast, though potentially incomplete explanation, can get this soft solution (steps 11 and 12) and use the values of the slack variables to modify the problem, and restart the process at step 1. If they want a more complete explanation, the soft solution can be sent to our MUS enumerator (step 13) to speed up its enumeration. In the red path the MUS enumerator is called directly (step 8) and the enumeration is done as usual. In both cases, the enumeration often requires several calls to a solver (step 14). The enumerated MUSes are sent to the UI and presented to the users (steps 15 and 16), who can then review these MUSes (step 16), select their preferred MCS (step 17), and trigger another recovery phase (step 18), where slack variables are only added to the selected MCS constraints (step 19). The resulting soft instance is then flattened (step 20) and solved (step 21). The MCS solution is presented to users in step 22, who can adopt it into the original problem or perform further modifications, before restarting the process.

6 Soft Generator

Our soft generator has two main goals. The first one is to quickly identify either one MCS or the constraints in the MUSes of one MCS. The former greatly reduces the number of constraints users need to modify, but locks them into the constraints of that MCS. The latter allows users to select their preferred MCS from the constraints in the MUSes, but can yield too many constraints. Thus, the former is more meaningful, the latter more flexible. The second goal is to quantify the minimum changes required to restore feasibility [5].

To achieve our goals, we modify the infeasible instance based on [27] in two ways. First, all user constraints are *relaxed* by introducing slack variables, whose values provide the required quantification for our two goals (and must be ≥ 0). Second, the original objective function is replaced by one that always minimises the total slack value (second goal) but can either minimise the number of slacks with a positive value (yielding one MCS), or not (yielding constraints in the MUSes of one MCS). For Plant Layout the violation measured by the slacks corresponds to length units, and the new objective function uses them without any scaling (i.e., the unit of violation is constant, e.g., 1mm) and without any weights, since we do not have *a priori* preferences. Instead, users can control such preferences *a posteriori* (step 17) by switching constraints on and off (see Section MUSes and MCS Visualiser).

Relaxing constraints via slacks. Let $c, x \in \mathbb{R}^n$ be vectors of coefficients and variables, respectively, and $d \in \mathbb{R}$ a constant. Linear *inequality* $c^\top x \leq d$ is relaxed in [27] as $c^\top x \leq d + s_\omega$, where ω is the index of the constraint and, thus, of the slack variable s_ω . Since all slack variables must be ≥ 0 , linear *equality* $c^\top x = d$ is relaxed in [27] using two inequalities: $c^\top x \leq d + s_\omega^+$ and $c^\top x \geq d - s_\omega^-$, which increases the number of constraints.

Combinatorial constraints are not tackled in [27], and we are not aware of any method to quantify their violation. We define a general method by modifying MiniZinc's linearisation mechanism for MIP solvers [3]. Let us show how we do this using logic constraints to illustrate the method. Consider a logic constraint, such as a disjunction of inequalities. MiniZinc linearises logic constraints in three steps. First, each linear component (say $c^\top x \leq d$) is turned into an *indicator constraint*, $b = 1 \rightarrow c^\top x \leq d$, where b is an auxiliary 0/1 variable and \rightarrow logical implication. Second, each indicator constraint is turned into the big- M constraint $c^\top x \leq d + M(1 - b)$, where M is an upper bound for $c^\top x - d$. Finally, the original logic constraint is translated into Boolean arithmetic. Our method modifies step two to relax the indicator constraints by introducing a slack variable into the right-hand side: $b = 1 \rightarrow c^\top x \leq d + s_\omega$. This quantifies the infeasibility.

Replacing the objective function. To identify one MCS quickly, we generate the following lexicographic two-objective function, which first minimises the total number of positive slacks (yielding a minimum-cardinality MCS), and then the total magnitude of slack violation [6] (ensuring the changes needed to restore feasibility via that MCS are minimal):

$$\text{lex_min}(f_1 = \sum_{\omega} b_{\omega}; f_2 = \sum_{\omega} s_{\omega}) \quad (1)$$

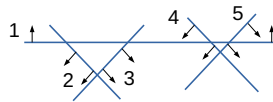
where s_{ω} is the slack variable introduced for constraint ω , and each b_{ω} is a new 0/1 variable defined by the indicator constraint $b_{\omega} = 0 \rightarrow s_{\omega} \leq 0$ (recall $s_{\omega} \geq 0$). Thanks to f_1 , the solution returned minimises the number of constraints that must be violated to get a soft solution to the original problem (those for which the slack variables are positive). This identifies one MCS (in fact a minimum-cardinality MCS), which was our first goal. Thanks to f_2 , the solution also quantifies the minimum total change required for the slack values in the violated constraints, which was our second goal. While the resulting value for f_2 might be higher than that obtained with f_2 alone, neither f_1 nor f_2 alone perform well: f_1 can yield too large slack values, while f_2 is unlikely to yield an MCS.

To identify the constraints in the MUSes of some MCS quickly we generate the following alternative lexicographic two-objective function [27]:

$$\text{lex_min}(f'_1 = \max_{\omega} s_{\omega}; f_2) \quad (2)$$

where ω , s_{ω} , and f_2 are as above. Here, f'_1 minimises the maximum slack value and f_2 the sum of slacks, ensuring each falls below the value returned by f'_1 . Note that this does not group constraints into MUSes, thus speeding up the computation. Again, neither f'_1 nor f_2 alone perform well: f'_1 may leave all slacks positive (including those of non MUS-members), while f_2 can lead to arbitrary MUS-member slacks being zero. Even joining both objectives by a linear combination can fail to produce a single MUS, thus reducing flexibility by reducing the amount of choice. Consider for example, constraints $x \leq 1 + s_1$, $y \leq 1 + s_2$, $Ax + By \geq 3 - s_3$, and $Cx + Dy \geq 3 - s_4$, with user inputs $A = B = C = D = 1$. Minimising $f'_1 + f_2$ produces $s_1 = s_2 = \frac{1}{2}$, $s_3 = s_4 = 0$, while a MUS must have one of the last two constraints. Even objective (2) may miss some constraints of a MUS if they overlap with those of another MUS. This is shown in Figure 3, where lines denote linear constraints 1–5 and arrows denote their feasible directions, yielding three MUSes: $\{1, 2, 3\}$, $\{1, 4, 5\}$, and $\{1, 2, 5\}$. When minimising objective (2), the slacks of constraints 3 and 4 become zero and are thus disregarded.

Note that when using any of the two objectives above, the original objective is not used. This allows us to omit their associated variables and constraints (giving a *satisfaction version* of the model), which can sometimes simplify the model considerably. This is also the case when enumerating MUSes (detailed in the next section).



■ **Figure 3** An example of overlapping MUSes.

7 MUS Enumerator

MUS enumeration often aims at computing an MCS. Automatically doing this can however prevent users from selecting the best MCS for their problem. To increase flexibility, our method graphically displays the MUSes enumerated (see next section) in such a way it is easy for users to identify different MCSes for those MUSes, and select one. We use FindMUS [18], a MUS enumeration tool available for MiniZinc, that extends MARCO to take advantage of the hierarchical structure present in MiniZinc models. User-provided names for the constraints and expressions in the model are included in FindMUS's JSON output, making it easier to integrate with other tools and present more meaningful MUSes to users.

The downside of such flexibility is practicality: MUS enumeration tools are often impractical for real-world systems due to the number of possible foreground constraint combinations. This number can be very large even after removing any redundant constraints added by the system to speed up solving, and after moving to the background (a) data constraints assumed to be correct, such as the dimensions of the equipment in Plant Layout, and (b) underlying constraints that cannot be modified by the user, such as the non-overlap and the group constraints. These constraints are easily marked in the model using MiniZinc annotations, and automatically removed or put in the background by our method. This also increases meaningfulness as it focuses users on the constraints they can change (e.g., the size of a group), and reduces the number of MUSes pointing to the same problem (e.g., one per object in the group not fitting in the allocated space).

To further increase the practicality of MUS enumeration tools we have explored three alternative avenues. The first avenue uses the solution to the soft instance generated by the soft generator to try to speed up the search for MUSes. To achieve this, the MUS enumerator collects in set *Vars* all non-slack variables that occur in at least one constraint with a positive slack value. It then partitions the original set of constraints by defining the foreground as the set of constraints that have at least one variable in *Vars*, and the background as the remaining set of constraints. Intuitively, this focuses the MUSes on the variables that are directly involved in the infeasibility. For Plant Layout, *Vars* corresponds to the objects in the plant and our system uses the annotations in the MiniZinc model to speed up the detection of constraints that involve at least one object in *Vars*.

The second avenue explores the use of the Irreducible Inconsistent Subsystem (IIS) efficiently computed by some MIP solvers – we use Gurobi [14]. To achieve this, the MUS enumerator is modified to start each search for a MUS by asking the solver for an IIS, with the aim of improving performance. Gurobi can report whether the IIS is minimal or not, and we have modified FindMUS to deal with it accordingly, further shrinking non-minimal subsets to a MUS before reporting it. The third avenue combines the two previous ones by allowing the MUS enumerator to take advantage of both soft generation and IIS.

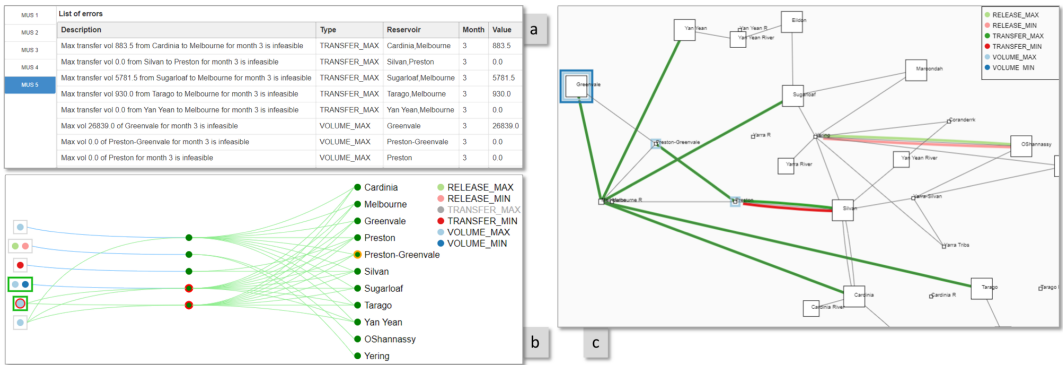


Figure 4 Conflict visualisations for the Water Management problem introduced in Section 10.

8 MUSes and MCS Visualiser

As shown in Figure 2, step 16 enables users to review the MUSes found by the MUS enumerator, and select a correction set (minimal or not, step 17) to execute steps 18–22. To help users during this selection, we developed three problem-independent ways to visualise MUSes, each providing different levels of detail and different perspectives on the conflicts (Figure 1(b) for Plant Layout, and Figure 4 for Water Management, see Section 10).

The most detailed visualisation provides a list of all MUSes found by the algorithm (2 in Figure 1(b) and 5 in Figure 4(a)). Users can select a MUS in the list to see details of its conflicting constraints including description, error type, associated elements and (if the constraint was relaxed) the value of its variables. The user can select one (or more) constraints from the MUSes in the list to form a (possibly not minimal) correction set. The selected constraints are used for step 17 in the workflow shown in Figure 2.

The list visualisation is only meaningful when the number of MUSes and/or constraints in each MUS is small. To increase meaningfulness for large lists, we created a graph representation (Figure 4(b)) that connects the constraints (coloured dots on the left-hand side) with their MUSes (green dots in the middle) and their elements (green dots on the right-hand side). The colour of each constraint dot identifies its type, as described by the legend appearing in the top-right of the figure. Constraints that occur in the same MUSes are grouped in a rectangle and linked to those MUSes, significantly reducing the number of connections and visual clutter.

Users can interact with the graph in two ways. First, if they select a constraint on the graph, the system highlights with a red frame the constraint and all MUSes linked to it (e.g., MUS 2 in Figure 1(b) and MUS 4 and 5 in Figure 4(b)), indicating that all those MUSes will be resolved if the highlighted constraint, or any constraint in that rectangle, is relaxed. The system also highlights with a green frame any rectangle linked to the highlighted MUSes, indicating that selecting constraints in those rectangles is no longer required to resolve the MUSes. This helps users find a suitable MCS (achieved when all MUSes are highlighted), which will be used to relax its constraints and find a solution (steps 18–22 in Figure 2). The second way of interacting with the graph is by deselecting one or more types of constraints in the legend, indicating the user would prefer not to modify them (e.g., constraint “minimum distance” in Figure 1(b) and “TRANSFER_MAX” in Figure 4(b) are greyed out). This causes the system to remove those constraints from the left-hand side. Note that if the user deselects too many constraint types, the remaining constraints might not resolve all existing MUSes. If so, those MUSes are highlighted as a warning to the user.

The third visualisation is a network (Figure 4(c)) that helps users better understand the conflicts by focusing on the conflicting elements and their relationships. Nodes in the network represent constraint elements, while black edges represent relationships among them. The network can be laid out using force-directed layout or fixed locations. Constraint conflicts are visualised on top of this network using the same colour codes as those in the MUS graph visualisation, and drawn either as frames around a node (if the conflict only involves that node) or as coloured edges (if it involves multiple ones).

9 Experimental Evaluation for Plant Layout

This section aims at evaluating the trade-offs between practicality and flexibility for the conflict resolution alternatives offered by our method in the context of a real-world application, as well as their meaningfulness and usefulness.

Benchmarks. We used the Plant Layout’s UI to create four classes of benchmarks: small (S), medium (M), large (L) and extra-large (XL). Small benchmarks have between 2 and 5 boxes and (except for S2) no pipes, medium have 19 boxes and 20 pipes, large have 78 boxes and 66 pipes, and extra-large have 217 boxes and 207 pipes. We then created variations of these four classes by adding constraints that made them infeasible. Those constraints were carefully selected to create a representative single conflict of a specific type. In small benchmarks we evaluated six types of base conflicts using “toy-plants” as a proof of concept to check that these conflicts could be effectively detected using our approach. For the three larger classes, we applied four of these base conflicts: 1) a conflict involving minimum-maximum distance, 2) a conflict involving symmetric placement of equipment, 3) a conflict involving the relative attachment position of a box, and 4) a conflict involving insufficient group size for contained boxes. We also created benchmarks with a combination of conflicts. To reduce the experiment size, we selected four additional conflict combinations of the given base conflicts: 5) symmetry + minimum-maximum distance, 6) symmetry + attachment position, 7) symmetry + group size, and 8) all four base conflicts combined. This resulted in 8 benchmarks for each size class, e.g., M1 for medium size with a minimum-maximum distance conflict, M5 for a medium size with a combined symmetry + minimum-maximum distance conflict, and M8 for a medium size with all four base conflicts combined. This in turn resulted in 30 *infeasible benchmarks*, whose characteristics are shown in Table 1.

The first two columns show the benchmark size and name. The next four show the total number of constraints in the instances generated in step 4 when the benchmark is initially solved (empty if MiniZinc detects infeasibility and aborts), in step 10 when it is relaxed by the soft generator searching for the set of constraints either in an MCS (equation (1) – denoted SGC) or in its MUSes (equation (2) – denoted SGU), respectively, and in step 8 when calling FindMUS directly (denoted FM). The last four columns show the number of conflicts in the benchmark (equal to the number of constraints in a minimum MCS), the number of objects (boxes or pipes) involved in these conflicts, and the type of constraints in conflict (MnD indicates a minimum distance, MxD a maximum distance, Sym a symmetry constraint, AttPos the position of an attached box, GrSz a group size, and CtSp the size of the container space). Note that the last benchmark in the medium, large and extra-large size (M8, L8 and XL8) contains all the conflicts from the first 4 benchmarks in that size.

Setup. All benchmarks were run in 8 different configurations: the three already introduced (SGC, SGU and FM) and the combinations FM+IIS, SGC+FM, SGC+FM+IIS, SGU+FM and SGU+FM+IIS (IIS denotes Gurobi’s IIS). In addition, all configurations involving

■ **Table 1** Characteristics of the 30 infeasible benchmarks created for Plant Layout.

Bnchm.	#Constraints in associated instances				#Conflicts	#Objects in the conflicts		Type of constraints in conflicts
	Initial	SGU	SGC	FM		#boxes	#pipes	
Small	S1	42	113	174	115	1	2	0 MnD, MxD
	S2	141	295	434	306	1	3	2 MnD, MxD, Sym
	S3	99	264	421	250	1	4	0 MnD, MxD, AttPos
	S4	60	150	223	155	1	3	0 MnD, GrSz
	S5	40	107	180	101	1	2	0 MxD
	S6	116	364	575	362	1	5	0 MnD, CtSp
Medium	M1	1,516	4,105	6,506	4,241	1	2	0 MnD, MxD
	M2	1,524	4,145	6,570	4,277	1	3	2 MnD, MxD, Sym
	M3	1,516	4,105	6,506	4,241	1	4	0 MnD, MxD, AttPos
	M4	1,512	4,083	6,472	4,221	1	3	0 MnD, GrSz
	M5	1,528	4,167	6,604	4,297	2	5	2 MnD, MxD, Sym
	M6	1,528	4,167	6,604	4,297	2	7	2 MnD, MxD, Sym, AttPos
	M7	1,524	4,145	6,570	4,277	2	6	2 MnD, MxD, Sym, GrSz
	M8	1,532	4,189	6,638	4,317	4	12	2 MnD, MxD, Sym, AttPos, GrSz
Large	L1	10,653	36,669	60,754	35,687	1	2	0 MnD, MxD
	L2	10,657	36,683	60,780	35,699	1	3	6 MnD, MxD, Sym
	L3	10,653	36,669	60,754	35,687	1	3	0 MnD, MxD, AttPos
	L4	10,653	36,665	60,750	35,683	1	3	0 MnD, GrSz
	L5	10,661	36,705	60,814	35,719	2	5	6 MnD, MxD, Sym
	L6	10,661	36,705	60,814	35,719	2	6	6 MnD, MxD, Sym, AttPos
	L7	10,657	36,683	60,780	35,699	2	6	6 MnD, MxD, Sym, GrSz
	L8	10,665	36,727	60,848	35,739	4	11	6 MnD, MxD, Sym, AttPos, GrSz
XLarge	XL1	95,285	336,243	554,302	332,384	1	2	0 MnD, MxD
	XL2	-	334,157	552,240	329,181	3	5	6 MnD, MxD, Sym
	XL3	95,285	336,243	554,302	332,384	1	2	0 MnD, MxD, AttPos
	XL4	93,854	329,875	543,878	326,030	1	4	0 MnD, GrSz
	XL5	-	336,387	554,482	332,513	4	7	6 MnD, MxD, Sym
	XL6	-	336,387	554,482	332,513	4	7	6 MnD, MxD, Sym, AttPos
	XL7	-	330,019	544,058	326,159	4	9	6 MnD, MxD, Sym, GrSz
	XL8	-	330,063	544,126	326,199	6	13	6 MnD, MxD, Sym, AttPos, GrSz

FM were run in two modes: finding one MUS and finding all. This is because FindMUS supports a fast single MUS extraction mode that might also be useful for users. All runs were performed on an Intel Core i7-8700K (3.70 GHz, 12 cores, 12MB cache) with 32GB memory using MiniZinc 2.5.5, FindMUS, and Gurobi 9.0.1. Each instance was run on 2 cores with 2 instances being run in parallel at a time. Time limits for the soft generator (step 13) and for FindMUS (step 14) were both set at 1/2 hour for small sizes, and 2 hours and 4 hours, respectively, for the other sizes. If reached, SGC and SGU might return unnecessary constraints, while FM might return non-minimal or not all unsatisfiable subsets.

Results. Tables 2 and 3 show the results. Values are underlined if a timeout was reached. Table 2 shows the number and total value of positive slack variables, and number of MUSes. The first two columns show the benchmark size and name. The next four show the number (#sl) and total value (slack) of positive slack variables returned by SGC and SGU. The last six show the number of MUSes (#ms) returned by FM, FM+IIS, SGC+FM, SGC+FM+IIS, SGU+FM, and SGU+FM+IIS, with FM enumerating all MUSes. Table 3 shows the run-times. The first two columns show the benchmark size and name, and the others show the times (minutes:seconds) taken by the 8 configurations in two modes: FM computing one MUS (greyed) and all MUSes.

Discussion. There is no clear winner between SGC and SGU in terms of speed. Both only time out for the XL benchmarks (except for XL5) with symmetry conflicts (whose absolute value constraint slows down solving). When there is no timeout, SGC always returns a minimum MCS, while SGU returns more constraints (those in the associated MUSes), where partition into MUSes is unknown. If SGU returns many constraints (e.g., XL8), they can become overwhelming and thus not meaningful. If it does not, they can also be more

■ **Table 2** Number and total value of positive slack variables, and number of MUSEs.

Bnchm.	SGU			SGC		FM	FM+IIS	SGU+FM	SGU+FM+IIS	SGC+FM	SGC+FM+IIS	
	#	sl	slack	#	sl	ms	#	ms	#	ms	#	ms
Small	S1	2	2,000	1	2,000	1	1	1	1	1	1	1
	S2	4	2,321	1	2,506	5	5	5	5	2	2	2
	S3	4	500	1	500	1	1	1	1	1	1	1
	S4	2	500	1	500	1	1	1	1	1	1	1
	S5	2	1,000	1	1,000	1	1	1	1	1	1	1
	S6	4	3,000	1	3,000	1	1	1	1	1	1	1
Medium	M1	2	2,000	1	2,000	1	1	1	1	1	1	1
	M2	4	2,321	1	2,506	5	5	5	5	2	2	2
	M3	4	500	1	500	1	1	1	1	1	1	1
	M4	2	500	1	500	1	1	1	1	1	1	1
	M5	5	4,000	2	4,506	6	6	6	6	3	3	3
	M6	5	2,821	2	3,006	6	6	6	6	3	3	3
	M7	5	2,821	2	3,006	6	6	6	6	3	3	3
	M8	7	5,000	4	5,506	8	8	8	8	5	5	5
Large	L1	2	50	1	50	4	4	4	4	4	4	4
	L2	9	2,182	1	1,000	24	0	24	0	0	0	2
	L3	3	750	1	500	2	2	2	2	2	2	2
	L4	2	450	1	450	1	1	1	1	1	1	1
	L5	9	2,232	2	1,050	4	4	4	4	9	8	8
	L6	11	2,848	2	1,500	1	1	1	1	1	1	1
	L7	10	2,632	2	1,450	0	0	25	0	0	0	0
	L8	14	3,348	4	2,000	2	2	2	2	2	2	2
XLarge	XL1	2	1,500	1	1,500	1	1	1	1	1	1	1
	XL2	11	40,097	4	22,850	4	4	4	4	4	4	4
	XL3	3	1,950	1	1,500	2	2	2	2	1	1	1
	XL4	3	1,550	1	1,300	3	5	0	3	1	1	1
	XL5	12	41,516	5	24,350	5	5	5	5	5	5	5
	XL6	15	52,661	5	25,056	3	7	5	5	5	5	5
	XL7	12	41,316	5	55,213	4	8	5	5	7	7	7
	XL8	15	44,316	7	33,871	6	5	6	7	7	7	7

■ **Table 3** Run-time results for Plant Layout. Times are given in minutes:seconds.

Bnchm.	SGU		SGC		FM		FM + IIS		SGU+FM		SGU+FM+IIS		SGC+FM		SGC+FM+IIS	
	1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms
Small	S1	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01
	S2	<0:01	<0:01	<0:01	<0:01	<0:01	0:12	<0:01	<0:01	0:02	0:12	<0:01	<0:01	0:02	0:04	0:04
	S3	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	0:02	<0:01	<0:01	<0:01	<0:01	<0:01
	S4	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01
	S5	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01	<0:01
	S6	<0:01	<0:01	<0:01	<0:01	<0:01	0:07	<0:01	<0:01	0:07	0:08	<0:01	<0:01	0:07	0:07	0:07
Medium	M1	<0:01	<0:01	0:03	0:05	0:06	0:08	0:04	0:05	0:08	0:09	0:04	0:05	0:07	0:08	0:08
	M2	<0:01	0:03	0:03	0:16	0:21	1:18	0:05	0:20	0:22	1:26	0:06	0:10	0:24	0:47	0:47
	M3	<0:01	0:02	0:05	0:11	0:19	0:26	0:06	0:10	0:25	0:29	0:07	0:10	0:30	0:33	0:33
	M4	<0:01	0:02	0:02	0:03	0:11	0:12	0:03	0:05	0:15	0:16	0:04	0:05	0:15	0:17	0:17
	M5	<0:01	0:11	0:04	1:43	0:14	2:58	0:05	1:41	0:17	2:52	0:14	1:34	0:26	2:09	2:09
	M6	<0:01	14:39	0:04	1:34	0:18	3:01	0:06	1:25	0:24	2:47	14:40	15:05	15:04	16:39	16:39
	M7	<0:01	0:03	0:03	0:34	0:09	1:32	0:04	0:40	0:11	1:43	0:06	0:27	0:13	1:07	1:07
	M8	<0:01	0:13	0:04	188:11	0:22	194:46	0:05	149:17	0:24	149:36	0:15	127:01	0:36	128:07	128:07
Large	L1	0:33	0:31	1:02	5:37	1:17	5:49	5:59	29:10	3:14	18:37	5:19	28:22	3:03	18:29	18:29
	L2	1:27	70:00	40:00	242:28	5:36	283:42	15:40	241:44	66:14	247:35	79:24	312:51	138:15	327:31	327:31
	L3	0:30	1:36	0:41	6:40	0:59	7:29	7:14	17:10	3:03	14:00	2:51	11:33	3:22	12:46	12:46
	L4	0:28	0:34	0:48	2:15	1:48	3:12	4:03	6:00	3:34	5:37	2:38	3:26	2:30	3:29	3:29
	L5	6:59	3:30	1:00	240:06	1:39	277:59	8:36	275:57	9:15	246:59	121:31	244:26	6:07	280:16	280:16
	L6	3:30	5:54	0:38	294:54	1:01	254:35	4:33	250:05	4:47	284:20	6:38	261:16	7:10	345:39	345:39
	L7	2:16	9:09	1:10	254:31	2:37	245:25	14:45	243:57	4:31	298:03	84:48	262:09	11:28	279:48	279:48
	L8	4:34	1:36	0:41	252:43	1:04	289:09	5:23	253:20	5:47	275:15	2:30	251:14	2:53	305:01	305:01
XLarge	XL1	02:51	11:54	49:26	153:16	19:52	118:35	100:25	127:54	52:48	78:51	84:36	97:36	51:49	63:47	63:47
	XL2	121:25	139:03	02:12	240:27	04:36	250:00	130:20	361:41	127:08	362:49	142:09	164:39	143:03	157:57	157:57
	XL3	106:23	19:22	19:47	251:49	31:51	251:38	195:51	303:19	161:01	278:49	36:20	42:20	89:29	101:34	101:34
	XL4	08:15	13:38	23:41	240:33	12:48	241:44	83:19	248:53	35:15	250:03	19:15	21:20	28:28	29:34	29:34
	XL5	83:46	135:58	38:50	240:57	04:35	244:25	87:00	324:24	89:42	333:00	138:54	378:22	139:06	363:55	363:55
	XL6	123:23	144:03	02:10	240:18	04:18	241:00	135:45	362:48	126:39	364:03	162:16	384:46	148:24	389:58	389:58
	XL7	121:43	243:06	02:22	240:10	04:23	241:16	132:23	362:02	129:04	377:29	246:26	414:04	253:35	420:07	420:07
	XL8	122:01	254:37	18:46	240:31	05:09	240:07	127:28	364:21	127:14	361:48	258:48	495:23	258:23	494:42	494:42

meaningful (may give more context to the failure) and flexible (give more choice). However, if the MCS returned by SGC is meaningful enough, SGC is more useful, as it always provides the minimum number of constraints that must be changed. This already shows the value of having different approaches available.

As expected, enumerating all MUSes using FM is significantly slower than using SGU or SGC. However, if FM does not time out, it is more flexible than SGC (gives a single MCS) and SGU (might miss constraints), and more meaningful, as it provides the full context while partitioning constraints into MUSes (as opposed to SGU). Also, using FM to find a single MUS is usually fast. Users could repeatedly do this to restore instances with several MUSes.

The combination of FM with SGU is not as promising as expected: the possible reduction in flexibility due to only looking at the variables in the constraints returned by SGU only pays off with speed-ups for M3, M8 and XL1. The rest are actually slower. The combination with SGC is better in terms of speed-ups (M2, M3, M8, XL1 to XL4), but might lose MUSes (M2 and M8). SGU and SGC improve FM's usefulness by providing users with slack values.

These results, and our own experience as users, suggest the following strategies for Plant Layout. Less experienced users, those modifying an unfamiliar model, and those with enough time would benefit from using FM on its own and in combination with SGC to get results that are useful (via SGC's slacks), flexible (FM's MUSes) and meaningful (FM+SGC's MUS reduction). Experienced users are likely to find SGU's results useful, fast and (together with their knowledge and experience) meaningful enough to restore feasibility, compared to being flexible with longer run-times and possible timeouts for very large instances.

10 Second Real-World Example: Water Management

Managing a city's water supply requires a complex set of decisions regarding the city's storage/service reservoirs, tunnels and water-transfer pipelines. The work of [15] describes an optimisation system designed to do this by creating a plan that outlines the anticipated operations in the water supply system for years ahead, and identifies for each month the expected water to be sourced, stored, moved from one reservoir to another, or released to the rivers. The resulting operating plan is built to (a) satisfy water demand, environmental and network capacity constraints, (b) minimise the risk of uncontrolled releases from the water harvesting sites, and (c) minimise the cost of transferring water between reservoirs.

Data. The UI provides a predefined water supply network and historical stream-flows, where the user can set (a) the reservoir capacities, (b) planning horizon length and, for each month, (c) the water demands (d) min/max water levels per reservoir and (e) min/max water flow per pipeline. In addition, for the selected planning horizon, users must specify the water levels of each reservoir at the beginning of the period and the anticipated stream-flow derived from past data. This allows users to generate operating plans for different rain inflow scenarios, storage distributions (both spatial and seasonal) and planning horizons.

Underlying constraints. Each reservoir is constrained to maintain its water level within the specified range for each month, and to release water to waterways to meet any specified environmental requirements. Each pipeline (i.e., the water transfer link between reservoirs, from a reservoir to the city, or from a reservoir to the river) cannot transfer more water than its maximum capacity. No redundant constraints were added to the model.

■ **Table 4** Characteristics of the 14 infeasible benchmarks created for Water Management.

Bnchm.		#Constraints in associated instances				#Conflicts	Type of constraints in conflicts
		Initial	SGU	SGC	FM		
Short	S1	12,532	14,994	16,139	16,096	1	MnV, MxV
	S2	12,531	14,993	16,138	16,095	1	MnT, MxT
	S3	12,531	14,993	16,138	16,095	1	MnR, MxR
	S4	12,532	14,994	16,138	16,096	2	MnV, MxV, MnT, MxT
	S5	12,532	14,994	16,138	16,096	2	MnV, MxV, MnR, MxR
	S6	12,531	14,993	16,137	16,095	2	MnT, MxT, MnR, MxR
	S7	12,532	14,994	16,137	16,096	3	All three conflicts
Long	L1	62,506	74,604	80,209	80,166	1	MnV, MxV
	L2	62,505	74,603	80,208	80,165	1	MnT, MxT
	L3	62,505	74,603	80,208	80,165	1	MnR, MxR
	L4	62,506	74,604	80,208	80,166	2	MnV, MxV, MnT, MxT
	L5	62,506	74,604	80,208	80,166	2	MnV, MxV, MnR, MxR
	L6	62,505	74,603	80,207	80,165	2	MnT, MxT, MnR, MxR
	L7	62,506	74,604	80,207	80,166	3	All three conflicts

User constraints. When generating an operating plan, users can modify the network capacity constraints (a, d and e mentioned under “Data” above) to consider events such as the closure of a pipeline or a reduction in the water level at a reservoir during a given period due to maintenance.

Restoring feasibility. The number of network capacity-related parameters exposed to users is high and increases with the planning horizon length. Thus, users can easily cause infeasibility by setting conflicting values. For example, setting a low maximum water level at a reservoir can result in excess water that needs to be transferred out, which then conflicts with the limit set on a pipeline.

10.1 Experimental Evaluation for Water Management

Benchmarks. We built two classes of benchmarks: with short-term operating plans (12-month) and with long-term ones (60-month). All are built on the same distribution network, which has 11 reservoirs, 9 transfer nodes and 46 connections. We then created seven benchmarks for each class by modifying the operational parameters that made them infeasible. Three of the benchmarks have a single base conflict: 1) a conflict involving minimum-maximum reservoir volume, 2) a conflict involving minimum-maximum pipeline limits, and 3) a conflict involving minimum-maximum release limits. The remaining four benchmarks are created from combinations of these base conflicts.

Table 4 shows the characteristics of the resulting 14 *infeasible* benchmarks. Columns 1–7 follow those of Table 1. The last column shows the type of constraints in conflict, where MnV indicates a constraint on the minimum water volume to be maintained in a reservoir, MxV a constraint on the maximum reservoir volume, MnT a minimum transfer volume constraint, MxT a maximum transfer volume constraint, MnR a constraint on the minimum volume released to the rivers, and MxR a constraint on the maximum release volume.

Setup and Results. We used the same setup as for Plant Layout. The results, shown in Tables 5 and 6, follow the same structure as that of Tables 2 and 3.

Discussion. There is no timeout in any of the instances and configurations tried. Interestingly, short-term instances have more slacks/MUSEs than long-term ones. This is because the chosen conflicts have more impact in the short-term benchmarks and can be resolved in many more ways than in the long-term ones.

■ **Table 5** Number and total value of positive slack variables, and number of MUSes.

Bnchm.	SGU		SGC		FM		FM+IIS		SGU+FM		SGU+FM+IIS		SGC+FM		SGC+FM+IIS	
	#	sl	#	sl	#	ms	#	ms	#	ms	#	ms	#	ms	#	ms
Short	S1	3	18,976	1	19,236	7	7	7	7	7	7	7	7	7	7	7
	S2	2	3,317	1	5,267	4	4	4	3	3	3	3	3	3	3	3
	S3	1	240	1	240	1	1	1	1	1	1	1	1	1	1	1
	S4	5	22,293	2	24,503	19	19	19	16	16	16	16	16	16	16	16
	S5	3	19,216	2	19,476	8	8	8	8	8	8	8	8	8	8	8
	S6	3	3,557	2	5,507	5	5	5	4	4	4	4	4	4	4	4
	S7	5	22,533	3	24,743	20	20	20	17	17	17	17	17	17	17	17
Long	L1	1	800	1	800	1	1	1	1	1	1	1	1	1	1	1
	L2	1	620	1	620	1	1	1	1	1	1	1	1	1	1	1
	L3	1	310	1	310	1	1	1	1	1	1	1	1	1	1	1
	L4	2	1,420	2	1,420	2	2	2	2	2	2	2	2	2	2	2
	L5	2	1,110	2	1,110	2	2	2	2	2	2	2	2	2	2	2
	L6	2	930	2	930	2	2	2	2	2	2	2	2	2	2	2
	L7	3	1,730	3	1,730	3	3	3	3	3	3	3	3	3	3	3

■ **Table 6** Run-time results for Water Management. Times are given in minutes:seconds.

Bnchm.	SGU		SGC		FM		FM + IIS		SGU+FM		SGU+FM+IIS		SGC+FM		SGC+FM+IIS	
	1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms	1 ms	all ms
Short	S1	<0:01	<0:01	0:06	0:22	0:03	0:09	0:05	0:27	0:04	0:13	0:05	0:16	0:04	0:09	0:09
	S2	<0:01	<0:01	0:13	0:38	0:03	0:19	0:07	0:28	0:04	0:15	0:07	0:17	0:04	0:09	0:09
	S3	<0:01	<0:01	0:07	0:08	0:03	0:05	0:06	0:08	0:04	0:06	0:06	0:08	0:04	0:06	0:06
	S4	<0:01	<0:01	0:14	1:42	0:05	0:42	0:06	1:10	0:04	0:33	0:08	0:56	0:04	0:23	0:23
	S5	<0:01	<0:01	0:06	0:29	0:04	0:12	0:06	0:26	0:05	0:13	0:06	0:27	0:04	0:13	0:13
	S6	<0:01	<0:01	0:13	0:51	0:03	0:26	0:14	0:35	0:04	0:20	0:08	0:23	0:04	0:12	0:12
	S7	<0:01	<0:01	0:06	2:05	0:04	1:00	0:06	1:25	0:06	0:45	0:06	1:05	0:04	0:31	0:31
Long	L1	0:06	0:05	0:33	0:41	0:17	0:27	0:32	0:42	0:23	0:28	0:32	0:36	0:22	0:26	0:26
	L2	0:06	0:05	0:35	0:43	0:17	0:27	0:38	0:50	0:24	0:34	0:35	0:51	0:22	0:32	0:32
	L3	0:07	0:05	0:30	0:41	0:17	0:26	0:43	0:48	0:25	0:35	0:39	0:44	0:22	0:32	0:32
	L4	0:06	0:05	0:34	1:12	0:18	0:41	0:38	1:15	0:24	0:47	0:35	1:11	0:23	0:46	0:46
	L5	0:06	0:05	0:34	1:16	0:18	0:40	0:35	1:09	0:24	0:46	0:43	1:13	0:23	0:45	0:45
	L6	0:06	0:05	0:30	1:05	0:17	0:39	0:42	1:19	0:24	0:47	0:38	1:15	0:23	0:46	0:46
	L7	0:06	0:05	0:32	1:45	0:17	0:59	0:38	1:48	0:24	1:07	0:41	1:40	0:23	1:06	1:06

SGC is slightly faster than SGU, but this is only noticeable in the long-term benchmarks. SGC always returns a minimum MCS, while SGU returns more constraints in short-term instances but an MCS in long-term ones because the total amount of violations required by an MCS happens to be the minimum. In all cases, SGC produces the same number of constraints as the number of conflicts introduced. Both FM and FM+IIS, enumerate all MUSes, but FM+IIS is faster. Out of the four combined approaches (SGU+IIS, SGU+FM, SGU+FM+IIS, SGC+FM+IIS), SGC+FM+IIS is always fastest, and is even faster than FM+IIS for many short-term benchmarks. All four combinations produce the same number of MUSes for the same benchmark. While they often take longer to enumerate all MUSes than FM and FM+IIS, they produce fewer MUSes in some benchmarks (S2, S4, S6, S7).

Like for Plant Layout, using FM/FM+IIS to enumerate all MUSes is slower than using SGU and SGC. However, it is more flexible and provides the full context; especially with the visualiser, where users can easily see the connections between the conflicting constraints.

Based on the above results and our experience as users, we recommend using FM+IIS or SGC+FM+IIS for this problem, which we have already used to find conflicts for our industry partner very quickly. This might be surprising, as SGC is faster and always points to the correct reservoirs and/or connections. However, all configurations are fast and there are often multiple ways to resolve conflicts in this problem, one of which could be the desired way to fix them, and this could only be found by enumerating all the MUSes.

11 Conclusion

This paper addresses the need for decision systems to provide *meaningful, useful, practical and flexible* conflict resolution techniques to be actually deployed by its target users.

To be meaningful an interface must present application concepts rather than software constructs. While requiring a problem-specific interface, the ability to interact with a solution, change it and get feedback about the infeasibility of the change, is problem-independent. We propose a generic explanation tool that shows which user constraints conflict with each other, describes the conflicts (Figures 1 and 4), and avoids overwhelming users by supporting diagnosis at different levels of detail through the annotation of the underlying constraints, and by letting users determine the amount of information shown (MCS/MUSes).

To be useful the system must also show users how to resolve conflicts. We propose a combination of conflict resolution methods that not only identify the key conflicting decisions, but also reveal the amount of change needed to resolve them. We do this by both reducing the set of infeasible constraints to a minimal set, and finding the smallest total relaxation of the constraints that can restore feasibility.

Theoretical approaches to detecting feasibility do not scale to real-world applications, as finding all minimal unsatisfiable sets (MUSes) is often computationally impractical for them. A key contribution of this paper to practicality and flexibility is the ability for users to control infeasibility detection, as shown in Figure 2, so that useful explanations are returned in the right amount of time (from a few seconds to overnight, depending on the need).

Usability is a slippery and often underestimated concept. The novel features we propose result from a collaborative process with industry users, who were initially sceptical and/or confused about the software, and now want to see the solutions it produces, understand the best trade-offs between the different objectives and learn why some seemingly obvious decisions turn out to be far from optimal.

This work identifies a few future research directions, and gives some initial indication of what can be achieved. For instance, we developed a method to soften logical constraints. This can be extended to other constraint types, e.g., CP global constraints. The presented conflict visualisations are helpful but require user evaluation to prove their usefulness.

References

- 1 Fahiem Bacchus and George Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *International Conference on Computer Aided Verification*, pages 70–86. Springer, 2015. doi:10.1007/978-3-319-21668-3_5.
- 2 Gleb Belov, Tobias Czauderna, Maria Garcia de la Banda, Matthias Klapperstueck, Ilankaikone Senthoooran, Mitch Smith, Michael Wybrow, and Mark Wallace. Process Plant Layout Optimization: Equipment Allocation. In John Hooker, editor, *Principles and Practice of Constraint Programming*, pages 473–489. Springer, 2018. doi:10.1007/978-3-319-98334-9_31.
- 3 Gleb Belov, Peter J. Stuckey, Guido Tack, and Mark Wallace. Improved Linearization of Constraint Programming Models. In Michel Rueher, editor, *Principles and Practice of Constraint Programming*, pages 49–65. Springer, 2016. doi:10.1007/978-3-319-44953-1_4.
- 4 Hadrien Cambazard, Fabien Demazeau, Narendra Jussien, and Philippe David. Interactively solving school timetabling problems using extensions of constraint programming. In *PATAT 2004*, volume 3616 of *LNCS*, pages 190–207, 2004. doi:10.1007/11593577_12.
- 5 John W. Chinneck. *Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods*. Springer, 2008. doi:10.1007/978-0-387-74932-7.

- 6 John W. Chinneck. The maximum feasible subset problem (maxFS) and applications. *INFOR: Information Systems and Operational Research*, 57(4):496–516, 2019. doi:10.1080/03155986.2019.1607715.
- 7 Iain Dunning, Joey Huchette, and Miles Lubin. JuMP: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017. doi:10.1137/15M1020575.
- 8 Andreas Falkner, Alois Haselboeck, Gerfried Krames, Gottfried Schenner, Herwig Schreiner, and Richard Taupe. Solver Requirements for Interactive Configuration. *Journal of Universal Computer Science*, 26(3):343–373, 2020.
- 9 Alexander Felfernig, Monika Schubert, and Christoph Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *AI EDAM*, 26(1):53–62, 2012. doi:10.1017/S0890060411000011.
- 10 Eugene C. Freuder. Explaining Ourselves: Human-Aware Constraint Reasoning. In *Proceedings 31st AAAI*, pages 4858–4862. AAAI, 2017.
- 11 R. M. Gasca, C. Valle, M. T. Gómez-López, and R. Ceballos. NMUS: Structural Analysis for Improving the Derivation of All MUSes in Overconstrained Numeric CSPs. In Daniel Borrajo, Luis Castillo, and Juan Manuel Corchado, editors, *Current Topics in Artificial Intelligence: 12th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2007, Salamanca, Spain, November 12-16, 2007. Selected Papers*, volume 4788 of *LNCS*, pages 160–169. Springer, 2007. doi:10.1007/978-3-540-75271-4_17.
- 12 John Gleeson and Jennifer Ryan. Identifying Minimally Infeasible Subsystems of Inequalities. *INFORMS Journal on Computing*, 2(1):61–63, 1990. doi:10.1287/ijoc.2.1.61.
- 13 Olivier Guieu and John W. Chinneck. Analyzing infeasible mixed-integer and integer linear programs. *INFORMS Journal on Computing*, 11(1):63–77, 1999. doi:10.1287/ijoc.11.1.63.
- 14 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2020. URL: <http://www.gurobi.com>.
- 15 Heerbod Jahanbani, M.D.U.P. Kularathna, Guido Tack, and Ilankaikone Senthooran. Considerations in developing an optimisation modelling tool to support annual operation planning of Melbourne Water Supply System. In Sondoss Elsawah, editor, *MODSIM2019, 23rd International Congress on Modelling and Simulation. Modelling and Simulation Society of Australia and New Zealand, December 2019*, page 592. Springer, 2019.
- 16 Ulrich Junker. QuickXplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI’01 Workshop on Modelling and Solving Problems with Constraints*, 2001.
- 17 Niklas Lauffer and Ufuk Topcu. Human-understandable explanations of infeasibility for resource-constrained scheduling problems. In *Proc. 2nd Workshop on Explainable AI Planning*, pages 44–52, 2019.
- 18 Kevin Leo and Guido Tack. Debugging Unsatisfiable Constraint Models. In *CPAIOR 2017*, pages 77–93, 2017. doi:10.1007/978-3-319-59776-8_7.
- 19 Mark H. Liffiton and Ammar Malik. Enumerating Infeasibility: Finding Multiple MUSes Quickly. In Carla Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 160–175. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-38171-3_11.
- 20 Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008. doi:10.1007/s10817-007-9084-z.
- 21 Joao Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In *Twenty-Third International Joint Conference on Artificial Intelligence*, pages 615–622, 2013.
- 22 Joao Marques-Silva and Alessandro Previti. On Computing Preferred MUSes and MCSes. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 58–74. Springer, 2014. doi:10.1007/978-3-319-09284-3_6.

- 23 Deepak Mehta, Barry O'Sullivan, and Luis Quesada. Extending the notion of preferred explanations for quantified constraint satisfaction problems. In *International Colloquium on Theoretical Aspects of Computing*, pages 309–327. Springer, 2015. doi:10.1007/978-3-319-25150-9_19.
- 24 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.
- 25 Alexander Schiendorfer, Alexander Knapp, Gerrit Anders, and Wolfgang Reif. MiniBrass: Soft constraints for MiniZinc. *Constraints*, 23(4):403–450, 2018. doi:10.1007/s10601-018-9289-2.
- 26 J.N.M. van Loon. Irreducibly inconsistent systems of linear inequalities. *European Journal of Operational Research*, 8(3):283–288, 1981. doi:10.1016/0377-2217(81)90177-6.
- 27 Jian Yang. Infeasibility resolution based on goal programming. *Computers & Operations Research*, 35(5):1483–1493, 2008. doi:10.1016/j.cor.2006.08.006.

SAT-Based Approach for Learning Optimal Decision Trees with Non-Binary Features

Pouya Shati ✉

Department of Computer Science, University of Toronto, Canada

Eldan Cohen ✉

Department of Mechanical and Industrial Engineering, University of Toronto, Canada

Sheila McIlraith ✉

Department of Computer Science, University of Toronto, Canada

Vector Institute, Toronto, Canada

Abstract

Decision trees are a popular classification model in machine learning due to their interpretability and performance. Traditionally, decision-tree classifiers are constructed using greedy heuristic algorithms, however these algorithms do not provide guarantees on the quality of the resultant trees. Instead, a recent line of work has studied the use of exact optimization approaches for constructing optimal decision trees. Most of the recent approaches that employ exact optimization are designed for datasets with binary features. While numeric and categorical features can be transformed to binary features, this transformation can introduce a large number of binary features and may not be efficient in practice. In this work, we present a novel SAT-based encoding for decision trees that supports non-binary features and demonstrate how it can be used to solve two well-studied variants of the optimal decision tree problem. We perform an extensive empirical analysis that shows our approach obtains superior performance and is often an order of magnitude faster than the current state-of-the-art exact techniques on non-binary datasets.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization; Computing methodologies → Machine learning

Keywords and phrases Decision Tree, Classification, Numeric Data, Categorical Data, SAT, MaxSAT

Digital Object Identifier 10.4230/LIPIcs.CP.2021.50

Funding We gratefully acknowledge funding from NSERC, the CIFAR AI Chairs program (Vector Institute), and from Microsoft Research.

1 Introduction

Classification models assign class labels to data observations. Learning classification models from a set of training examples is a key task in supervised machine learning. Decision trees are among the most popular classification models in machine learning as they provide interpretable models and tend to have good performance.

Traditionally, decision-tree classifiers are constructed using greedy heuristic algorithms, such as CART [9], ID3 [21], and C4.5 [22]. However, these algorithms do not provide guarantees on the quality of the resultant trees, which can therefore be unnecessarily large or potentially inaccurate [4]. Alternatively, learning a globally optimal decision-tree classifier was shown to be NP-complete for several optimization criteria [17, 14]. Still, in recent years a variety of exact techniques have been proposed to solve the problem of optimal decision-tree classifiers. One class of techniques focuses on optimizing decision tree size (either the depth of the tree or the number of nodes in the tree) such that all training examples are correctly classified [3, 7, 18]. Another class of techniques, instead, focuses on maximizing the number of correctly classified training examples, while constraining the maximal depth of the decision tree [24, 23, 2, 15, 6]. Recent works found that optimal decision-tree classifiers tend to have



© Pouya Shati, Eldan Cohen, and Sheila McIlraith;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 50; pp. 50:1–50:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

higher out-of-sample accuracy than heuristic approaches [16, 3, 13, 6]. Furthermore, many of the recent approaches for optimal decision trees can be extended to support additional constraints that the resulting trees must satisfy, such as fairness constraints [1].

Many of the recent state-of-the-art approaches are designed for datasets with binary features [3, 24, 18, 15, 23], and some are further limited to binary class labels (i.e., classification with only two classes) [18, 15]. However, in practice, most real-world datasets contain categorical and numeric features. In order to perform classification in datasets with categorical and numeric features, these approaches convert any non-binary feature into a set of binary features using standard techniques (see discussions in [3, 24]). This conversion often introduces a large number of binary features and may lead to poor performance.

In this work, we present and empirically evaluate a novel approach for learning optimal decision-tree classifiers that can directly handle non-binary features without converting them into binary features. Specifically, we make the following contributions:

1. We present a novel SAT encoding of decision trees that directly supports numeric features and use this encoding to solve two well-known optimization problems in classification tasks: (1) finding a minimum-depth decision tree that correctly classifies all training examples; and (2) finding a decision tree of a given depth that maximizes the number of correctly classified training examples.
2. We present an extension of our SAT encoding that directly supports categorical features based on power set branching and show, theoretically and empirically, that the new encoding is more expressive and can lead to decision trees with better solution quality w.r.t. the two studied optimization problems.
3. We perform extensive experimental analysis and show that our encoding significantly outperforms recent state-of-the-art techniques on datasets with non-binary features for each of the studied optimization problems.

2 Technical Background

2.1 Problem Definition

In Section 2.1.1 we formally define the decision trees considered in this work and in Section 2.1.2 we define the two optimization problems we consider for learning optimal decision trees.

2.1.1 Decision Trees

We start by defining the tree structure of a decision tree. Then, we define the depth of the tree based on the deepest leaf node and the special case of complete tree.

► **Definition 1 (Tree Structure).** A tree structure \mathcal{T} is a tuple $(\mathcal{T}_B, \mathcal{T}_L, \delta, p, l, r)$ where \mathcal{T}_B and \mathcal{T}_L are finite sets respectively representing the branching and leaf nodes, $\delta \in \mathcal{T}_B$ is the root node, $p : (\mathcal{T}_B \cup \mathcal{T}_L - \{\delta\}) \rightarrow \mathcal{T}_B$ is the parent function, and $l, r : \mathcal{T}_B \rightarrow (\mathcal{T}_B \cup \mathcal{T}_L)$ are respectively the left and right child functions. A well-formed tree structure is one with the property $\forall x, y : p(y) = x \leftrightarrow (l(x) = y \vee r(x) = y)$.¹

► **Definition 2 (Tree Depth).** Given a tree structure $\mathcal{T} = (\mathcal{T}_B, \mathcal{T}_L, \delta, p, l, r)$, we recursively define the depth of each node $t \in \mathcal{T}_B \cup \mathcal{T}_L$ as $\text{depth}(t) = \text{depth}(p(t)) + 1$ with $\text{depth}(\delta) = 0$. The depth of the tree structure is defined to be the maximum depth among its leaf nodes, $\text{depth}(\mathcal{T}) = \max_{t \in \mathcal{T}_L} \text{depth}(t)$.

¹ Note that well-formedness guarantees the absence of loops in parental relations.

► **Definition 3** (Complete Tree). A tree structure $\mathcal{T} = (\mathcal{T}_B, \mathcal{T}_L, \delta, p, l, r)$ is considered complete if all the leaf nodes have the same depth, i.e., $\forall t_1, t_2 \in \mathcal{T}_L : \text{depth}(t_1) = \text{depth}(t_2)$.

Next, we formally define decision trees and describe how to perform prediction using such decision trees.

► **Definition 4** (Decision Tree). Given a set of features F and integer labels C , a decision tree is a tuple $\mathcal{D} = (\mathcal{T}, \beta, \alpha, \theta)$ where $\mathcal{T} = (\mathcal{T}_B, \mathcal{T}_L, \delta, p, l, r)$ is a tree structure, $\beta : \mathcal{T}_B \rightarrow F$ is the feature selection function, $\alpha : \mathcal{T}_B \rightarrow \text{dom}(F)$ is the threshold selection function, and $\theta : \mathcal{T}_L \rightarrow C$ is the leaf labelling function. A well-formed decision tree satisfies $\forall t \in \mathcal{T}_B : \alpha(t) \in \text{dom}(\beta(t))$. Note that $\text{dom}(j)$ represents the set of possible values for feature $j \in F$ and $\text{dom}(F) = \bigcup_{j \in F} \text{dom}(j)$.

An evaluation of a set of features F is called a data point x . For each $j \in F$, $x[j] \in \text{dom}(j)$ represents the value of feature j at point x . A dataset usually referred to as X , contains a finite set of data points $x_i \in X$ for training or testing purposes.

A decision tree $\mathcal{D} = (\mathcal{T}, \beta, \alpha, \theta)$ predicts the label of a given point x_i by starting from the root and recursively delivering the point to the left or right child until a leaf node is reached. The label of the leaf node is the output. The recursive *predict* function $\Theta(t, x_i)$ on node $t \in \mathcal{T}_B \cup \mathcal{T}_L$ and point $x_i \in X$ is defined as follows:

$$\Theta(t, x_i) = \begin{cases} \theta(t) & \text{if } t \in \mathcal{T}_L \\ \Theta(l(t), x_i) & \text{elseif } x_i[\beta(t)] \leq \alpha(t) \\ \Theta(r(t), x_i) & \text{else} \end{cases}$$

The prediction of decision tree \mathcal{D} for data point $x' \in X$ can be obtained by $\Theta(\delta, x')$ where δ is the root node of the decision tree. We use the simplified notation $\Theta(x')$ to denote $\Theta(\delta, x')$.

Given a labelled dataset, i.e., a dataset in which the correct class label for each data point is provided, we can measure the accuracy of a decision tree on the dataset with respect to the provided labels.

► **Definition 5** (Decision Tree Accuracy). Given a set of features F and integer labels C , a set of training examples $x_i \in X$, and a labelling $\gamma : X \rightarrow C$, the accuracy of decision tree $\mathcal{D} = (\mathcal{T}, \beta, \alpha, \theta)$ on X is the fraction of data points in X that are correctly classified with respect to the labelling γ ,

$$\frac{\sum_{x_i \in X} \mathbb{1}[\Theta(x_i) = \gamma(x_i)]}{|X|},$$

where $\mathbb{1}$ is the indicator function.

2.1.2 Learning Optimal Decision Trees

We consider two problems representing two different optimization criteria for optimal decision tree. Problem 1 consists of finding a decision tree with minimum depth such that all the training examples are correctly classified. This problem is consistent with Avellaneda [3].

► **Problem 1** (Min-Depth Optimal Decision Tree). Given set of features F and integer labels C , a set of training examples $x_i \in X$, and a labelling $\gamma : X \rightarrow C$, output a decision tree $\mathcal{D} = (\mathcal{T}, \beta, \alpha, \theta)$ such that \mathcal{T} is a minimum depth complete tree and \mathcal{D} correctly classifies all training examples, i.e., $\Theta(x_i) = \gamma(x_i) \forall x_i \in X$.

Problem 2 consists of finding a decision tree that maximizes the number of training examples that are correctly classified subject to a constraint on the tree depth. This problem is consistent with the problems considered in a variety of recent works, e.g., in [15, 23, 2].

► **Problem 2** (Max-Accuracy Optimal Decision Tree). *Given a set of features F and integer labels C , a set of training examples $x_i \in X$, a labelling $\gamma : X \rightarrow C$, and a chosen depth d , output a decision tree $\mathcal{D} = (\mathcal{T}, \beta, \alpha, \theta)$ such that \mathcal{T} is a complete tree of the chosen depth, $\text{depth}(\mathcal{T}) = d$, and \mathcal{D} maximizes the number of training examples that are correctly classified, i.e., $\sum_{x_i \in X} \mathbb{1}[\Theta(x_i) = \gamma(x_i)]$ where $\mathbb{1}$ is the indicator function.²*

2.2 SAT and MaxSAT

SAT formulae are represented in Conjunctive Normal Form (CNF) and are defined over a set of Boolean variables. A SAT formula is a conjunction of clauses, each clause is a disjunction of literals, and each literal is either a Boolean variable or its negation. An assignment of the Boolean variables satisfies a clause if at least one of its literals is true. The SAT problem consists of finding an assignment of the variables that satisfies all clauses in a formula [8].

The MaxSAT problem is the optimization variant of the SAT problem and consists of finding an assignment of the variables that maximizes the number of satisfied clauses. Partial MaxSAT [12] is a generalization of the MaxSAT problem where the set of clauses consists of *hard clauses* that must be satisfied and *soft clauses* that can be violated.

3 SAT-based Encoding for Learning Optimal Decision Trees

In this section, we present our SAT-based approach for optimal decision trees. In Section 3.1, we present the core encoding for decision trees that can be used to solve the two optimization problems considered in this work. In Section 3.2, we present a SAT-based approach for solving the min-depth problem (Problem 1) by searching for increasingly deeper decision trees that can correctly classify all training examples. In Section 3.3, we present a partial MaxSAT encoding of the max-accuracy problem (Problem 2).

3.1 Encoding Decision Trees

We propose a SAT encoding of a decision tree, $\mathcal{D} = (\mathcal{T}, \beta, \alpha, \theta)$. Similar to previous works (e.g., [3, 23]), our encoding assumes a tree structure \mathcal{T} (typically a complete tree of some depth d), and decides on the values of β , α , and θ . When the depth of the tree is unknown in advance (e.g., in the min-depth optimal decision tree problem) we can solve a sequence of problems for trees of increasing depth as described in Section 3.2.

3.1.1 Variables

The following binary variables are used to represent the different aspects of a decision tree:

- $[a_{t,j}]$: Represents whether feature j is chosen for the split at branching node t .
- $[s_{i,t}]$: Represents whether point i is directed towards the left child, if it passes through branching node t .
- $[z_{i,t}]$: Represents whether point i ends up at leaf node t .
- $[g_{t,c}]$: Represents whether label c is assigned to leaf node t .

² Note that since $|X|$ is fixed, maximizing the number of correctly classified training examples is identical to maximizing the accuracy in Definition (5).

3.1.2 Clauses

The following set of hard clauses in conjunctive normal form guarantee the validity of the recursion in the modelled decision tree, and can consequently be used as the core encoding for both of the optimization problems we consider.

The clauses in Eq. (1) and Eq. (2) guarantee that exactly one feature is chosen at each branching node $t \in \mathcal{T}_B$.

$$(\neg a_{t,j}, \neg a_{t,j'}) \quad t \in \mathcal{T}_B, j \neq j' \in F \quad (1)$$

$$(\bigvee_{j \in F} a_{t,j}) \quad t \in \mathcal{T}_B \quad (2)$$

For each branching node $t \in \mathcal{T}_B$, we need to make sure that all the data points for which the feature value is less than or equal to the feature threshold are directed left and all the data points for which the feature value is greater than the threshold are directed right. We use $\#_j^i$ to denote the index of the i 'th data point in X when sorted by feature j in ascending order, assuming ties are broken arbitrarily, and define O_j to be the set of all consecutive pairs in this ordering, i.e., $O_j = \{(\#_j^i, \#_j^{i+1}) \mid 1 \leq i \leq |X|-1\}$. Then, the clauses in Eq. (3) guarantee that there are no two points with different feature values where the one with the higher value is directed left while the one with the lower value is directed right. The clauses in Eq. (4), together with the clauses in Eq. (3), guarantee that points with equal values are directed in a similar manner.

$$(\neg a_{t,j}, s_{i,t}, \neg s_{i',t}) \quad t \in \mathcal{T}_B, j \in F, (i, i') \in O_j(X) \quad (3)$$

$$(\neg a_{t,j}, \neg s_{i,t}, s_{i',t}) \quad t \in \mathcal{T}_B, j \in F, (i, i') \in O_j(X), x_i[j] = x_{i'}[j] \quad (4)$$

For each data point x_i , we need to guarantee the validity of its path in the decision tree. We use $A_l(t)$ (resp. $A_r(t)$) to denote all the ancestors of a leaf node $t \in \mathcal{T}_L$ such that t is a descendant of their left (resp. right) branch. The clauses in Eq. (5) and Eq. (6) guarantee that each data point that ends up at a leaf node follows the corresponding path. In contrast, the clauses in Eq. (7) guarantee that each data point that does not end up in leaf node $t \in \mathcal{T}_L$ has at least one deviation from the corresponding path

$$(\neg z_{i,t}, s_{i,t'}) \quad t \in \mathcal{T}_L, x_i \in X, t' \in A_l(t) \quad (5)$$

$$(\neg z_{i,t}, \neg s_{i,t'}) \quad t \in \mathcal{T}_L, x_i \in X, t' \in A_r(t) \quad (6)$$

$$(z_{i,t}, \bigvee_{t' \in A_l(t)} \neg s_{i,t'}, \bigvee_{t' \in A_r(t)} s_{i,t'}) \quad t \in \mathcal{T}_L, x_i \in X \quad (7)$$

The clauses in Eq. (8) guarantee that each leaf node is assigned at most one label. Note that we do not include constraints that prevent leaves with no label. The optimization criteria discussed in Sections 3.2 and 3.3 guarantee that in optimal solutions, all leaves that have corresponding training examples will be assigned a label. Leaf nodes that do not have any corresponding training examples will be assigned an arbitrary label post-optimization in order to maintain a valid decision tree.

$$(\neg g_{t,c}, \neg g_{t,c'}) \quad t \in \mathcal{T}_L, c \neq c' \in C \quad (8)$$

Finally, we add redundant constraints that help prune the search space. For each branching node, the clauses in Eq. (9) guarantee that the data point with the lowest feature value is directed left and the clauses in Eq. (10) guarantee that the data point with the highest feature value is directed right.

$$(\neg a_{t,j}, s_{\#_j^1, t}) \quad t \in \mathcal{T}_B, j \in F \quad (9)$$

$$(\neg a_{t,j}, \neg s_{\#_j^{|X|}, t}) \quad t \in \mathcal{T}_B, j \in F \quad (10)$$

3.1.3 Decoding Decision Trees from Solutions

Assuming a solution to the SAT encoding above, i.e., an assignment of the variables $a_{t,j}$, $s_{i,t}$, $z_{i,t}$, and $g_{t,c}$ that satisfy the clauses in Section 3.1.2, we now describe how to extract the decision tree $\mathcal{D} = (\mathcal{T}, \beta, \alpha, \theta)$. Decoding β is done by setting $\beta(t) = j$ if the variable $a_{t,j}$ is true. Similarly, decoding θ is done by setting $\theta(t) = c$ if the variable $g_{t,c}$ is true. Note that these procedures are valid since we are guaranteed that $a_{t,j}$ and $g_{t,c}$ are unique for each node, i.e., $\forall t \in \mathcal{T}_B : \sum_{j \in F} a_{t,j} = 1$ and $\forall t \in \mathcal{T}_L : \sum_{c \in C} g_{t,c} = 1$.

Since our SAT encoding does not explicitly compute the threshold for each node, to decode α we have to choose a threshold based on the direction of the data points in each branching node. For a branching node $t \in \mathcal{T}_B$ with $\beta(t) = j$, we set $\alpha(t) = x_i[j]$ where $(i, i') \in O_j(X)$ are consecutive data points according to the ordering of feature j such that $s_{i,t}$ is true and $s_{i',t}$ is false. Intuitively, this rule uses the largest value directed left as the feature threshold for the node t .

3.2 Encoding the Min-Depth Optimal Decision Tree Problem

To find a minimum depth decision tree such that all training examples are correctly classified, we add the following clauses to the core decision tree encoding described above:

$$(\neg z_{i,t}, g_{t,\gamma(x_i)}) \quad t \in \mathcal{T}_L, x_i \in X \quad (11)$$

The clauses in Eq. (11) guarantee that the class labels assigned to leaf nodes are consistent with the training set labels of the corresponding data points. If a data point x_i ends in leaf node $t \in \mathcal{T}_L$ then the assigned label of t must match the training label $\gamma(x_i)$.

In order to guarantee that we find the minimum depth decision tree, we follow the technique in [3]. We start by solving the SAT formula for a tree structure \mathcal{T} of depth 1 and in each iteration increase the depth by 1 until a solution is found. This guarantees that when the SAT solver finds a solution, the obtained decision tree has a minimum depth subject to the constraint that all training examples must be classified correctly.

3.3 Encoding the Max-Accuracy Optimal Decision Tree Problem

To find a decision tree of a given depth that maximizes the number of correctly classified training examples, we introduce the following variables to keep track of training examples that are correctly classified:

- $[p_i]$: Represents whether point x_i is correctly classified, i.e., x_i ends up in a leaf node with the label $\gamma(x_i)$.

We use a partial MaxSAT model that includes both hard and soft clauses. We use all the clauses from our core decision tree encoding in Section 3.1 as hard clauses. We also add the hard clauses in Eq. (12) that guarantee p_i is set to true only when x_i ends up in a leaf node whose label is consistent with the training label.

$$(\neg p_i, \neg z_{i,t}, g_{t,\gamma(x_i)}) \quad t \in \mathcal{T}_L, x_i \in X \quad (12)$$

In order to maximize the number of correctly classified training examples, we add a soft clause for each data point, that is satisfied whenever the point is correctly classified.

$$(p_i) \quad x_i \in X \quad (13)$$

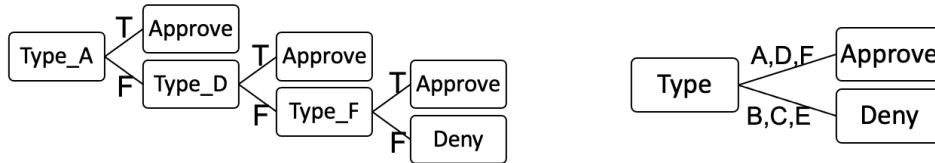
Therefore, the optimal solution for this partial MaxSAT model is a decision tree that maximizes the number of correctly classified training examples.

4 Extending Optimal Decision Trees for Categorical Features

Categorical features are features that take their value from a set of values representing different categories that do not induce a natural ordering.³ For example, in a dataset of financial transactions we can have a feature that describes the type of transaction from a set of known transaction types. In order to deal with a categorical feature with K categories, one can convert the feature to K binary features representing a one-hot encoding of the original categorical feature. However, this can lead to unnecessarily deep decision trees as splits may be required for many categories.

In this section, we show that we can easily extend our approach to generate decision trees that support branching on categorical features directly. We employ *power set branching* in which a selected subset of the categories is assigned to the left branch while the subset that contains the rest of the categories is assigned to the right branch. For the min-depth optimal decision tree problem, such extension can lead to decision trees of smaller depth that can potentially be found earlier. For the max-accuracy optimal decision tree problem, such extension can allow more flexible trees that achieve higher accuracy for the same depth.

► **Example 1.** To demonstrate the potential benefit of power set branching, consider a simplified dataset of transactions that can either be approved or denied. Each transaction has one categorical feature describing the transaction type with the categories $\{A, B, C, D, E, F\}$. Transaction with types A, D , and F are approved, while transactions with types B, C , and E are denied. Figure 1 (left) shows a standard decision tree where the categorical feature was converted to six binary features representing a one-hot encoding of the original feature. Figure 1 (right) shows the extended variant of decision trees where we can branch directly on categorical features by assigning a subset of the categories to each branch. The standard decision tree requires branching, in sequence, on multiple different categories leading to a tree with a minimum depth of 3. In contrast, the extended decision tree that supports branching on categorical features has a depth of 1.



■ **Figure 1** Left: a standard decision tree for the example dataset. Right: a decision tree with power set branching for categorical features.

4.1 Decision Trees with Power Set Branching for Categorical Features

We assume a set of features F that includes categorical features F_C and numeric features F_N such that $F = F_C \cup F_N$. In order to support power set branching, we augment the decision tree with a category subset selector α_C , $\mathcal{D}_C = (\mathcal{T}, \beta, \alpha, \alpha_C, \theta)$. We denote the set of branching nodes associated with numeric features $\Pi^N = \{t \in \mathcal{T}_B \mid \beta(t) \in F_N\}$ and similarly define Π^C the set of branching nodes associated with categorical features. We use α as a

³ Some categorical features induce a natural ordering and can therefore be represented as numeric features. For example a categorical feature with the categories $\{\text{Low}, \text{Medium}, \text{High}\}$ can be transformed to a numeric feature with the values $\{1, 2, 3\}$.

threshold selector for branching nodes with numeric features, $\alpha : \Pi^N \rightarrow \text{dom}(F_N)$, and α_C that selects a subset of categories for branching nodes with categorical feature from the corresponding power set $\alpha_C : \Pi^C \rightarrow 2^{\text{dom}(F_C)}$.⁴

Intuitively, α_C provides the subset of categories for which data points should be directed left. To predict the label of a given point x_i we use the recursive Θ_C as follows:

$$\Theta_C(t, x_i) = \begin{cases} \theta(t) & \text{if } t \in \mathcal{T}_L \\ \Theta_C(l(t), x_i) & \text{elseif } (t \in \Pi^N \wedge x_i[\beta(t)] \leq \alpha(t)) \vee (t \in \Pi^C \wedge x_i[\beta(t)] \in \alpha_C(t)) \\ \Theta_C(r(t), x_i) & \text{else} \end{cases}$$

Decision trees with power set branching are more expressive than decision trees that use a binarized encoding of these categorical features. Specifically, each decision tree that uses binary branching for categorical features can be transformed into a decision tree with power set branching with identical predictions.

► **Proposition 1.** *Given a decision tree $\mathcal{D} = (\mathcal{T}, \beta, \alpha, \theta)$ operating on a set of features $\text{bin}(F)$, there exists a decision tree with power set branching on the same structure $\mathcal{D}'_C = (\mathcal{T}, \beta', \alpha, \alpha'_C, \theta)$ operating on the set of features F such that*

$$\forall x_i : \Theta_C(x_i) = \Theta(\text{bin}(x_i))$$

where $\text{bin}(F)$ is F with its categorical features encoded using binary features in one-hot style and $\text{bin}(x_i)$ is x_i with its values encoded according to the binarized features set $\text{bin}(F)$.

The above proposition is correct since each branching node in the decision tree \mathcal{D} associated with a binary feature that correspond to one category c in the categorical feature $j \in F_C$ can be replaced in \mathcal{D}' with a node that branches directly on the feature j and directs the subset of categories $\{c\}$ to the right and the subset that contains the rest of the categories to the left. Proposition 1 implies the following corollaries on the solution-quality guarantees of power set branching w.r.t each of the optimization problems.

► **Corollary 6.** *Given an instance of Problem 1, the minimum depth found for a decision tree with power set branching is always equal to or less than that of a decision tree with branching based on binarized encoding of categorical features.*

► **Corollary 7.** *Given an instance of Problem 2, the maximum accuracy found for a decision tree with power set branching is always equal to or more than that of a decision tree with branching based on binarized encoding of categorical features.*

4.2 SAT-based Encoding of Decision Trees with Power Set Branching

Interestingly, the proposed extension involves only minor changes to the decision tree encoding in Section 3.1. Eq. (14)–(24) show the modified encoding with the changes highlighted in blue. The clauses in Eq. (16) guarantee that data points where the feature value is less or equal to the feature threshold are directed left and vice versa. As this does not apply to categorical features, we restrict Eq. (16) to numeric features. Instead, we add Eq. (18) that, together with the existing Eq. (17), guarantees that data points with the same category are directed in the same direction. Finally, for numeric features we used redundant constraints

⁴ Similar to α , a well-formedness condition on α_C would dictate that $\forall t \in \Pi^C : \alpha_C(t) \subseteq \text{dom}(\beta(t))$.

that guarantee the lowest feature value is directed left and the highest feature value is directed right. For categorical features we do not have such ordering over categories and instead we simply use the clauses in Eq. (23) to make sure that some arbitrary category is chosen to be directed left (in categorical features, $\#_j$ represents an arbitrary ordering over the category values of feature j) and we modify Eq. (24) such that no specific category is forced to go right.⁵ Note that unlike previous work that focused on categorical features [13], our approach directly encodes optimal decision trees with both numeric and categorical features.

$$(\neg a_{t,j}, \neg a_{t,j'}) \quad t \in \mathcal{T}_B, j \neq j' \in F \quad (14)$$

$$\left(\bigvee_{j \in F} a_{t,j} \right) \quad t \in \mathcal{T}_B \quad (15)$$

$$(\neg a_{t,j}, s_{i,t}, \neg s_{i',t}) \quad t \in \mathcal{T}_B, j \in F_N, (i, i') \in O_j(X) \quad (16)$$

$$(\neg a_{t,j}, \neg s_{i,t}, s_{i',t}) \quad t \in \mathcal{T}_B, j \in F, (i, i') \in O_j(X), x_i[j] = x_{i'}[j] \quad (17)$$

$$(\neg a_{t,j}, s_{i,t}, \neg s_{i',t}) \quad t \in \mathcal{T}_B, j \in F_C, (i, i') \in O_j(X), x_i[j] = x_{i'}[j] \quad (18)$$

$$(\neg z_{i,t}, s_{i,t'}) \quad t \in \mathcal{T}_L, x_i \in X, t' \in A_l(t) \quad (19)$$

$$(\neg z_{i,t}, \neg s_{i,t'}) \quad t \in \mathcal{T}_L, x_i \in X, t' \in A_r(t) \quad (20)$$

$$(z_{i,t}, \bigvee_{t' \in A_l(t)} \neg s_{i,t'}, \bigvee_{t' \in A_r(t)} s_{i,t'}) \quad t \in \mathcal{T}_L, x_i \in X \quad (21)$$

$$(\neg g_{t,c}, \neg g_{t,c'}) \quad t \in \mathcal{T}_L, c \neq c' \in C \quad (22)$$

$$(\neg a_{t,j}, s_{\#_j, t}) \quad t \in \mathcal{T}_B, j \in F \quad (23)$$

$$(\neg a_{t,j}, \neg s_{\#_j, t}) \quad t \in \mathcal{T}_B, j \in F_N \quad (24)$$

Note that in our encoding, it is possible to have degenerate nodes for which all categories are directed left with none of the categories directed right. An optimal solution may have degenerate nodes, however we can easily convert such solutions to optimal solutions without degenerated nodes. As a post-optimization step, we arbitrarily select a subset of categories from the left branch and move them to the right branch. Then, we copy all the subtree of the left branch to the right branch, leading to identical predictions on the training examples and maintaining the optimality of the original solution.

4.2.1 Decoding Decision Trees with Power Set Branching

Decoding a decision tree from a solution to the SAT encoding above follows the same procedure as described in Section 3.1.3, with one key difference. For nodes that are associated with categorical branching, we need to decode α_C , the category subset selector that indicates the subset of categories associated with the left branch. For a categorical branching node $t \in \Pi^C$ that is associated with feature j , i.e., $\beta(t) = j, j \in F_C$, we define $\alpha_C(t)$ to be the set of categories in feature j for which there are data points in X that are directed left,

$$\alpha_C(t) = \{c \mid \exists x_i \in X : x_i[\beta(t)] = c \wedge s_{i,t} = 1\}.$$

⁵ Note that we could add clauses that guarantee that at least one category will go to the right, however it does not provide significant pruning and we therefore opted not to add them.

5 Experiments

In this section, we perform an extensive experimental evaluation of our SAT-based approach for optimal decision trees. For each of the two optimization criteria, we compare our approach to state-of-the-art exact approaches for optimal decision trees that optimize the same criterion based on the runtime and objective value. Previous work on each of the two criteria has already evaluated the quality of decision trees that are optimal with respect to a set of training examples on external validation datasets [15, 3, 2, 24] and also discussed the differences between the two criteria [15, 3]. In this work, we focus on the optimization performance with respect to each of the optimization criteria.

Our algorithm for the min-depth optimal decision tree problem is implemented in C++ based on SAT solver MiniSAT [11]. Our encoding for the max-accuracy optimal decision tree problem is solved using the MaxSAT solver Loandra [5]. The experiments were run on two 12-core Intel E5-2697v2 CPUs and 128G of ram.

5.1 Baselines

We compare our approach to recent state-of-the-art baselines for each optimization problem. For min-depth optimal decision trees, we compare our approach to Avellaneda’s SAT-based approach [3] that uses the MiniSAT solver [11]. For max-accuracy optimal decision trees, we compare our approach to Hu et al.’s MaxSAT encoding [15] solved by Loandra [5], and to Verhaeghe et al.’s constraint programming (CP) model [23] solved by Oscar [20]. Note that other approaches, such as OCT [6], BinOCT [24], DL8 [19], DL8.5 [2], have been previously compared to one or more of the baselines we consider in their original publications [15, 3, 23].

Binary Encoding of Non-Binary Features

Most recent state-of-the-art approaches, including the selected baselines, support datasets that contain only binary features. Therefore, these approaches convert any non-binary feature into a set of binary features in a pre-processing step prior to optimization. To binarize the datasets, we follow the procedure in Avellaneda [3] in which each non-binary feature that can have v possible different values is represented by v binary features, one for each possible value. Furthermore, if the feature is ordered, e.g., numeric features, then each binary feature represents the operator \leq as described in the following example from Avellaneda [3]:

► **Example 2.** *Consider a set of training examples where each example has a single integer feature f , $\{(1), (3), (4), (5)\}$. Then, we can transform f into three binary features $\tilde{f}_0, \tilde{f}_1, \tilde{f}_2$. If the feature \tilde{f}_0 is true it means that the value of f in the example is greater than 1. If the feature \tilde{f}_1 is true it means that the value of f in the example is greater than 3, etc. Therefore, the transformed dataset will be $\{(0,0,0), (1,0,0), (1,1,0), (1,1,1)\}$.*

Note that the above binary feature encoding supports the decision trees described in Definition 4 since selecting a binary feature for a branching node implies a threshold that values smaller or equal to the threshold are directed into one branch, while values greater than the threshold are directed into the other branch.

5.2 Datasets

To evaluate the performance of our approach, we run experiments on 15 datasets with different characteristics, obtained from the UCI repository [10]. Table 1 reports the size of each dataset $|X|$, the number of class labels $|C|$, the number of numeric features $|F_N|$, the number of

■ **Table 1** Description of the datasets used in our experiments.

Type	Name	$ X $	$ F_N $	$ F_B $	$ F_C $	\tilde{f}	$ C $
N	Banknote	1372	4	0	0	5016	2
	Breast Cancer	116	9	0	0	891	2
	Cryotherapy	90	5	1	0	93	2
	Immunotherapy	91	6	1	0	166	2
	Ionosphere	351	32	2	0	8114	2
	Iris	150	4	0	0	119	3
	User Knowledge	258	5	0	0	431	4
	Vertebral Column	310	6	0	0	1741	2
	Wine	178	13	0	0	1263	3
C	Credit Approval [†]	653	6	4	5	1130	2
	Promoter	106	0	0	57	228	2
	Soybean Large [†]	266	5	16	14	76	15
	Protease Cleavage	746	0	0	8	160	2
	Protease Cleavage(/4)	186	0	0	8	156	2
B	Car [‡]	1728	6	0	0	15	2
	Monk2	169	4	2	0	11	2

[†] Records with missing values were removed.

[‡] Classes were merged following Avellaneda [3].

binary features⁶ $|F_B|$, and the number of categorical features $|F_C|$. It also reports the number of binary features after the transformation of all the numeric and categorical features into binary features which is required for the baseline methods.

Consistent with our focus on non-binary features, we selected datasets with mostly numeric features (type N) or categorical features (type C). We also included the datasets Monk2 and Car that we consider binary (type B) since they have either binary features or numeric features that can be convert to binary with a small number of additional variables.

The datasets Credit Approval, Promoter, Soybean Large, and Protease Cleavage include a significant number of categorical features and will be used to evaluate our extension for optimal decision trees with categorical features. We also created a smaller version of Protease Cleavage that includes only 25% of the records (by selecting each fourth row).

5.3 Results

We start by evaluating our SAT-based formulation in Section 3 on numerical and binary datasets. In Section 5.4, we present the results for the min-depth optimal decision tree problem and in Section 5.5, we present the results for the max-accuracy optimal decision tree problem. In Section 5.6, we present results for the categorical branching extension in Section 4. Finally, in Section 5.7 we analyze the memory consumption.

⁶ In our encoding binary features are numeric features that take one of two possible values, however we list them separately in Table 1 as they are supported by the baseline methods without transformation.

■ **Table 2** Experimental results for min-depth optimal decision tree problem.

Dataset	Min Depth	Time (s)	
		Ours	SAT [3]
Banknote	4	5.82	T/O [4]
Breast Cancer	4	6.59	T/O [4]
Cryotherapy	4	0.08	0.24
Immunotherapy	4	0.18	1.3
Ionosphere	?	T/O [4]	T/O [3]
Iris	4	0.04	0.17
User Knowledge	5	1.31	59.44
Vertebral Column	5	87.35	T/O [5]
Wine	3	0.11	14.75
Car	8	T/O [8]	89.1
Monk2	6	2.73	0.28

5.4 Results for the Min-Depth Optimal Decision Tree Problem

We compare our encoding for min-depth optimal decision trees to Avellaneda’s SAT encoding [3] on the binary and numeric datasets. Following Avellaneda’s analysis, we set the time limit to 30 minutes. Table 2 shows the time to optimal solution for each of the approaches, as well as the tree depth of the optimal solution. As both approaches follow the procedure of solving SAT formulae for increasingly deeper trees until a solution is found, the first solution found is guaranteed to be optimal, i.e., of minimum depth. In case an approach does not find an optimal solution in the time limit, we report “T/O [d]” where d indicates the tree depth of the SAT formula being solved at the moment of time out. We highlight in bold results for which the alternative approach required at least twice as much run time or timed out.

The results in Table 2 show that our approach performs at least as well, and in most cases significantly better on datasets with numeric features. In particular, it finds optimal solutions for Banknote, Breast Cancer and Vertebral Column, for which the baseline timed out. In Ionosphere we find that both methods timed out, however our approach has managed to prove that a solution does not exist for a depth of 3, while the baseline only proved that a solution does not exist for a depth of 2. As expected, in the two binary datasets (Car and Monk2), we find that the baseline outperforms our method. In particular, it manages to find an optimal solution to Car, while our approach timed out.

5.5 Results for the Max-Accuracy Optimal Decision Tree Problem

Next, we compare our encoding for max-accuracy optimal decision trees to Hu et al.’s MaxSAT encoding [15] and Verhaeghe et al.’s CP encoding [23] on the binary and numeric datasets. Following Hu et al., we set the time limit to 15 minutes and run experiments for three different depth values, $\{2, 3, 4\}$. Table 3 shows the time required to find an optimal solution for each approach or “T/O” if an optimal solution was not found in the time limit. It also reports the cost of the solutions, i.e., the number of training examples that are *not* correctly classified, for each approach. If an optimal solution was found and proved, then the cost indicates the optimal cost. Otherwise, we report the cost of the best found solution by each approach. Note that the datasets Iris, User Knowledge, and Wine could only be solved by our approach as the two baselines only support classification problems with two class labels.

■ **Table 3** Experimental results for max-accuracy optimal decision tree problem.

Dataset	Depth	Solution Cost			Time (s)		
		Ours	MaxSAT [15]	CP [23]	Ours	MaxSAT [15]	CP [23]
Banknote	2	100	176	100	16.83	T/O	512.21
	3	23	550	100	105.79	T/O	T/O
	4	0	88	100	18.98	T/O	T/O
Breast Cancer	2	19	24	19	5.07	T/O	22.19
	3	9	25	12	242.16	T/O	T/O
	4	0	18	11	20.79	T/O	T/O
Cryotherapy	2	5	5	5	0.57	3.68	4.21
	3	1	1	1	0.73	17.57	27.39
	4	0	0	0	0.75	24.14	7.61
Immunotherapy	2	8	8	8	0.99	10.53	5.22
	3	4	4	4	3.81	T/O	146.45
	4	0	1	0	1.27	T/O	18.53
Ionosphere	2	29	41	29	155.06	T/O	T/O
	3	21	186	29	T/O	T/O	T/O
	4	10	76	28	T/O	T/O	T/O
Iris	2	6	–	–	0.6	–	–
	3	1	–	–	0.77	–	–
	4	0	–	–	0.82	–	–
User Knowledge	2	35	–	–	1.94	–	–
	3	10	–	–	3.29	–	–
	4	1	–	–	3.86	–	–
Vertebral Column	2	45	46	45	15.79	T/O	67.91
	3	32	44	42	T/O	T/O	T/O
	4	15	39	42	T/O	T/O	T/O
Wine	2	6	–	–	1.25	–	–
	3	0	–	–	1.62	–	–
Car	2	250	250	250	12.67	9.2	2.16
	3	182	182	182	T/O	T/O	5.99
	4	122	122	122	T/O	T/O	14.09
Monk2	2	57	57	57	2.74	4.38	1.38
	3	42	42	42	T/O	826.31	3.6
	4	32	31	31	T/O	T/O	8.12

The results in Table 3 show that for all numeric datasets our approach either finds solutions faster than the baselines or that all approaches time out. Furthermore, for all numeric datasets, our approach finds solutions that are at least equal, and in many cases significantly better than the baselines. As expected, for binary datasets (Car and Monk2), our approach underperforms relative to the baselines and timed out for depths 3 and 4. Still, it finds the optimal solution for all depths in Car and for depth 2 and 3 in Monk2. For the three datasets that could not be solved by the baselines, our approach found optimal solutions in seconds. In Wine, a tree of depth 3 correctly classifies all training examples.

■ **Table 4** Results for min-depth decision trees on datasets with categorical features.

Dataset	Min Depth			Time (s)		
	Ours-PS	Ours	SAT [3]	Ours-PS	Ours	SAT [3]
Credit Approval	?	?	?	T/O [6]	T/O [5]	T/O [5]
Protease Cleavage	?	?	?	T/O [5]	T/O [7]	T/O [6]
Protease Cleavage(/4)	4	?	?	0.69	T/O [7]	T/O [6]
Promoter	4	4	4	224.22	498.3	87.69
Soybean Large	?	?	?	T/O [6]	T/O [6]	T/O [6]

5.6 Results on Categorical Datasets

In this section, we present results on datasets with categorical features. We compare our power set branching for categorical features (Ours-PS) against our encoding without power set branching (Ours) and the baselines. As the optimal solution for the different approaches may be different, we do not highlight in bold the lowest run time. For example, our power set branching can fail to find an optimal solution in the time limit, while still obtaining a lower-cost solution compared to an optimal solution of a binary branching approach.

Table 4 reports the results for the min-depth optimal decision tree problem. Note that we report the min depth for each of the approaches as the power set approach can have a lower optimal solution. We find that most categorical datasets could not be solved by any of the methods in the time limit. However, in Protease Cleavage(/4), our encoding that is based on power set branching (Ours-PS) is able to find a solution decision tree of depth 4, while binary branching fails to find a solution after proving that no solution exists for a depth of 6. This demonstrates the expressiveness of our power set branching for categorical features.

Table 5 shows the results for the max-accuracy optimal decision tree problem. We find that in all of the cases, our power set approach obtains the lowest cost solution and that in almost all cases, these solutions are strictly lower compared to all other approaches. Note that we encountered an error when running the code for the CP approach [23] on the Credit Approval dataset with a depth of 4.

5.7 Results on Memory Consumption

To compare the memory consumption of the different approaches, we recorded the peak memory consumption of each approach for each of the datasets. Due to limited space, we only report aggregated results. Table 6 reports the mean and maximum values for the different approaches for each optimization problem and dataset type. In all cases, our approach has the lowest mean and maximum values. In categorical datasets, our power set encoding obtains the second lowest values. In the worst case, our approach required approximately 1GB of memory for the standard encoding and 1.9GB for the power set encoding. In comparison, Avellaneda [3] and Hu et al. [15] required more than 10GB in the worst case. For Verhaeghe et al. [23], we find that the maximum values are approximately within a factor of two from ours, however the mean values are still significantly higher than ours.

6 Conclusion

We present a novel SAT-based encoding for optimal decision trees that can directly encode non-binary features, namely numeric and categorical features. We study two variants of optimal decision trees based on different optimization criteria and present extensive empirical

■ **Table 5** Results for max-accuracy decision trees on datasets with categorical features.

Dataset	Depth	Cost				Time (s)			
		Ours-PS	Ours	[15]	[23]	Ours-PS	Ours	[15]	[23]
Credit Approval	2	84	84	84	84	106.26	151.45	T/O	33.06
	3	76	76	82	81	T/O	T/O	T/O	T/O
	4	60	65	74	[E]	T/O	T/O	T/O	[E]
Protease Cleavage	2	98	186	186	186	T/O	325.35	56.11	4.82
	3	101	133	149	133	T/O	T/O	T/O	29.68
	4	39	98	136	100	T/O	T/O	T/O	T/O
Protease Cleavage(/4)	2	13	49	49	49	18.91	28.58	16.5	4.36
	3	1	29	29	29	7.37	575.93	T/O	22.7
	4	0	37	40	17	1.27	T/O	T/O	T/O
Promoter	2	12	13	13	13	316.71	44.53	316.02	4.35
	3	3	3	4	3	T/O	324.77	T/O	63.22
	4	0	0	0	0	7.83	57.72	81.08	129.15
Soybean Large	2	152	159	–	–	16.22	99.93	–	–
	3	104	112	–	–	T/O	T/O	–	–
	4	35	45	–	–	T/O	T/O	–	–

■ **Table 6** Analysis of peak memory consumption (MB) for the different approaches.

		Min-Depth			Max-Accuracy			
		Ours-PS	Ours	[3]	Ours-PS	Ours	[15]	[23]
N	Mean	N/A	142.02	4,527.80	N/A	236.34	1,646.73	1,176.36
	Max	N/A	1,003.86	16,684.36	N/A	1,299.09	10,994.48	2,381.50
C	Mean	904.73	489.42	3,086.98	412.67	222.61	728.83	1,332.50
	Max	1,865.21	984.20	10,075.11	1,451.05	705.93	3,838.78	2,180.96

analysis that shows our approach outperforms recent state-of-the-art methods on datasets with non-binary features in terms of optimization quality. Furthermore, we show that our extension for categorical features can lead to higher quality solutions.

We believe our work can be extended in a number of ways. Extending our formulations with additional constraints, such as minimum-support constraints or pruning constraints, or alternative optimization criteria is an interesting direction for future work. Investigating the impact of different tree structures and the use of our approach as part of an ensemble method are also interesting research directions. Finally, our approach for dealing with non-binary features can be extended to other classification models such as decision sets [25].

References

- 1 Sina Aghaei, Mohammad Javad Azizi, and Phebe Vayanos. Learning optimal and fair decision trees for non-discriminative decision-making. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 1418–1426, 2019.
- 2 Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Learning optimal decision trees using caching branch-and-bound search. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 3146–3153, 2020.

- 3 Florent Avellaneda. Efficient inference of optimal decision trees. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 3195–3202, 2020.
- 4 Kristin P Bennett. Global tree optimization: A non-greedy decision tree algorithm. *Journal of Computing Science and Statistics*, 26:156–160, 1994.
- 5 Jeremias Berg, Emir Demirović, and Peter J Stuckey. Core-boosted linear search for incomplete MaxSAT. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, pages 39–56. Springer, 2019.
- 6 Dimitris Bertsimas and Jack Dunn. Optimal classification trees. *Machine Learning*, 106(7):1039–1082, 2017.
- 7 Christian Bessiere, Emmanuel Hebrard, and Barry O’Sullivan. Minimising decision tree size as combinatorial optimisation. In *International Conference on Principles and Practice of Constraint Programming (CP)*, pages 173–187. Springer, 2009.
- 8 Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.
- 9 Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. *Classification and regression trees*. Wadsworth & Brooks/Cole Advanced Books & Software, 1984.
- 10 Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL: <http://archive.ics.uci.edu/ml>.
- 11 Niklas Eén and Niklas Sörensson. Minisat SAT solver, 2003. URL: <http://minisat.se/Main.html>.
- 12 Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 252–265. Springer, 2006.
- 13 Oktay Günlük, Jayant Kalagnanam, Minhan Li, Matt Menickelly, and Katya Scheinberg. Optimal decision trees for categorical data via integer programming. *Journal of Global Optimization*, pages 1–28, 2021.
- 14 Thomas Hancock, Tao Jiang, Ming Li, and John Tromp. Lower bounds on learning decision lists and trees. *Information and Computation*, 126(2):114–122, 1996.
- 15 Hao Hu, Mohamed Siala, Emmanuel Hébrard, and Marie-José Huguet. Learning optimal decision trees with MaxSAT and its integration in AdaBoost. In *International Joint Conference on Artificial Intelligence and Pacific Rim International Conference on Artificial Intelligence (IJCAI-PRICAI)*, 2020.
- 16 Alexey Ignatiev, Joao Marques-Silva, Nina Narodytska, and Peter J Stuckey. Reasoning-based learning of interpretable ML models. In *International Joint Conference on Artificial Intelligence (IJCAI)*, page in press, 2021.
- 17 Hyafil Laurent and Ronald L Rivest. Constructing optimal binary decision trees is np-complete. *Information processing letters*, 5(1):15–17, 1976.
- 18 Nina Narodytska, Alexey Ignatiev, Filipe Pereira, Joao Marques-Silva, and IS RAS. Learning optimal decision trees with SAT. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1362–1368, 2018.
- 19 Siegfried Nijssen and Elisa Fromont. Optimal constraint-based decision tree induction from itemset lattices. *Data Mining and Knowledge Discovery*, 21(1):9–51, 2010.
- 20 OscaR Team. OscaR: Scala in OR, 2012. URL: <https://bitbucket.org/oscarlib/oscar>.
- 21 J Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.
- 22 J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.
- 23 Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning optimal decision trees using constraint programming. *Constraints*, 25(3):226–250, 2020.
- 24 Sicco Verwer and Yingqian Zhang. Learning optimal classification trees using a binary linear program formulation. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 1625–1632, 2019.
- 25 Jinqiang Yu, Alexey Ignatiev, Peter J Stuckey, and Pierre Le Bodic. Computing optimal decision sets with sat. In *International Conference on Principles and Practice of Constraint Programming (CP)*, pages 952–970. Springer, 2020.

Pseudo-Boolean Optimization by Implicit Hitting Sets

Pavel Smirnov ✉

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

Jeremias Berg ✉ 

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

Matti Järvisalo ✉ 

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

Abstract

Recent developments in applying and extending Boolean satisfiability (SAT) based techniques have resulted in new types of approaches to pseudo-Boolean optimization (PBO), complementary to the more classical integer programming techniques. In this paper, we develop the first approach to pseudo-Boolean optimization based on instantiating the so-called implicit hitting set (IHS) approach, motivated by the success of IHS implementations for maximum satisfiability (MaxSAT). In particular, we harness recent advances in native reasoning techniques for pseudo-Boolean constraints, which enable efficiently identifying inconsistent assignments over subsets of objective function variables (i.e. unsatisfiable cores in the context of PBO), as a basis for developing a native IHS approach to PBO, and study the impact of various search techniques applicable in the context of IHS for PBO. Through an extensive empirical evaluation, we show that the IHS approach to PBO can outperform other currently available PBO solvers, and also provides a complementary approach to PBO when compared to classical integer programming techniques.

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

Keywords and phrases constraint optimization, pseudo-Boolean optimization, implicit hitting sets

Digital Object Identifier 10.4230/LIPIcs.CP.2021.51

Supplementary Material *Software (Source Code and Experiment Data)*: <https://bitbucket.org/coreo-group/pbo-ihs-solver/>

archived at `swb:1:dir:cb17c95ad7c0b25976387d98840a459491191df2`

Funding Work financially supported by Academy of Finland under grants 322869 and 342145.

Acknowledgements The authors thank Paul Saikko for his initial implementation work on PBO-IHS.

1 Introduction

Declarative approaches are central in efficiently solving various types of NP-hard real-world optimization problems. Indeed various constraint optimization paradigms have been developed, ranging from mixed integer linear programming (MIP) [32] to finite-domain constraint optimization [34] and Boolean satisfiability (SAT) based maximum satisfiability (MaxSAT) [3] and its extensions to e.g. optimization modulo theories and MaxSMT [11, 41]. Each of the paradigms offer distinct features in terms of the declarative language used and the underlying algorithmic approach, ranging from branch-and-cut in MIP to the unsatisfiability-based search through iterative applications of SAT solvers in MaxSAT.

Pseudo-Boolean (PB) constraints [36] constitute an interesting constraint language for modelling and solving optimization problems. Also known as 0-1 linear constraints, stated as linear inequalities with integer coefficients over binary variables, pseudo-Boolean constraints constitute a central fragment of integer programming. However, PB constraints can also



© Pavel Smirnov, Jeremias Berg, and Matti Järvisalo;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 51; pp. 51:1–51:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

be viewed as natural generalizations of conjunctive normal form clausal constraints [5, 36]. Taking this view, effective specialized decision procedures have been developed for PB by lifting search techniques from the realm of SAT solving, boosted with additional inference techniques which lift the theoretical efficiency of PB solvers beyond that of standard SAT solvers [24, 10, 42, 12]. For a recent overview of such conflict-driven pseudo-Boolean solving, we refer the reader to [9]. Recent work on extending these techniques from decision to optimization problems by harnessing search techniques from both core-guided MaxSAT solving [21] and linear programming [20] have been shown to hold promise as an alternative approach to pseudo-Boolean optimization (PBO) complementing the more classical MIP solving techniques [33].

Building on these recent developments, in this work we develop an alternative approach to PBO drawing from both advances in PB solving and IP solving. In particular, motivated by the success of the so-called implicit hitting set (IHS) approach to MaxSAT [16, 17, 18, 37] as a current state-of-the-art MaxSAT solving approach alongside the core-guided approach, we develop a first instantiation of an IHS PBO solver. While the general IHS solving framework has been shown to be applicable in a range of settings [18, 19, 28, 39, 27, 25, 38], we are not aware of earlier work studying the applicability of IHS in the context of PBO. For realizing a competitive IHS PBO solver, we harness recent advances in native reasoning techniques for pseudo-Boolean constraints, which enable efficiently identifying inconsistent assignments over subsets of objective function variables [20], i.e., unsatisfiable cores in the context of PB. As the other major component, we employ integer programming and linear programming for hitting set computations over iteratively accumulated unsatisfiable cores as well as for integrating bounds-based inference techniques [14, 2]. We provide results from an extensive empirical evaluation of our implementation of the IHS approach to PBO, comparing its performance with a range of earlier developed specialized solvers for PBO as well as a commercial MIP solver, and evaluate the impact of the various search techniques of the empirical performance of the IHS PBO solver. It turns out that, overall, our IHS PBO solver outperforms earlier advances in specialized PBO solving, and shows complementary performance depending on the problem domains considered with respect to both other specialized PBO solvers and a commercial MIP solver.

2 Preliminaries

A binary variable x has the domain $\{0, 1\}$. A literal l over a variable x is either x or $\bar{x} \equiv (1-x)$. A pseudo-Boolean (PB) constraint C is a 0-1 integer linear inequality $\sum_i a_i l_i \geq B$ over literals l_i . The set of variables appearing in C is $\text{VAR}(C)$. We assume w.l.o.g. that all PB constraints are in normalized form, i.e., that each variable appearing in it is distinct and that the coefficients a_i and bound B are non-negative integers. We use $l = 0$ as shorthand for the constraints $l \geq 0$ and $-l \geq 0$ (rewritten in normal form). An assignment $\tau: \text{VAR}(C) \rightarrow \{0, 1\}$ is extended to literals by $\tau(\bar{l}) = 1 - \tau(l)$. An assignment τ satisfies C ($\tau(C) = 1$) if $\sum_i a_i \tau(l_i) \geq B$. When convenient we treat an assignment τ over a set X of variables as a set of literals $\tau = \{x \mid x \in X \wedge \tau(x) = 1\} \cup \{\bar{x} \mid x \in X \wedge \tau(x) = 0\}$.

A PB formula $F = \{C_1, \dots, C_n\}$ is a set of PB constraints. We denote by $\text{VAR}(F)$ the set of variables appearing in the constraints of F . An assignment $\tau: \text{VAR}(F) \rightarrow \{0, 1\}$ is a solution to F if it satisfies all constraints in F . We use $\tau(F) = 1$ to denote that τ is a solution to F ; $\tau(F) = 0$ denotes that τ is not a solution to F .

An instance \mathcal{F} of the pseudo-Boolean optimization problem (PBO) consists of a PB formula $\text{CONSTRAINTS}(\mathcal{F})$ and an objective function $O^{\mathcal{F}} \equiv \sum_i w_i l_i$ where each l_i is a literal over a variable $x_i \in \text{VAR}(\text{CONSTRAINTS}(\mathcal{F}))$ and w_i its non-negative integer weight. When

clear from context, we use \mathcal{F} and $\text{CONSTRAINTS}(\mathcal{F})$ interchangeably and drop the superscript from $O^{\mathcal{F}}$. We will sometimes abuse notation and treat O as either a set of literals or a set of weight-literal tuples, i.e., write $l \in O$ and $(w, l) \in O$ to obtain either literals or weight-literal pairs from O . The set of variables appearing in O is $\text{VAR}(O)$. The value of O under an assignment $\tau: \text{VAR}(O) \rightarrow \{0, 1\}$ is $O(\tau) = \sum_i w_i \tau(l_i)$. A solution τ to \mathcal{F} is optimal if it minimizes $O(\tau)$ over all solutions to \mathcal{F} . The PBO problem consists of finding an optimal solution to a given PBO instance.

The approach to computing optimal solutions of PBO instances presented in this work makes use of so called *core constraints* and *hitting sets*.

► **Definition 1.** A constraint $C = \sum_i a_i l_i \geq B$ is a core constraint of \mathcal{F} if: i) $\text{VAR}(C) \subset \text{VAR}(O^{\mathcal{F}})$ and ii) $(\tau(\mathcal{F}) = 1) \rightarrow (\tau(C) = 1)$ holds for all solutions to \mathcal{F} .

In words, a core constraint of an instance \mathcal{F} is a constraint over the variables in the objective function that is satisfied by any solution to \mathcal{F} .

► **Example 2.** Let $0 < r < n$ be two integers and consider the instance $\mathcal{F}^{n,r}$ with the constraints $\{\sum_{i=1}^n b_i \geq r\}$ and objective function $O \equiv \sum_{i=1}^n b_i$. Now $\text{VAR}(\mathcal{F}) = \{b_1, \dots, b_n\}$ and any assignment τ that assigns at least r variables in $\text{VAR}(\mathcal{F})$ to 1 is a solution to \mathcal{F} . The assignment τ^o that sets $\tau^o(b_j) = 1$ for $j = 1 \dots r$ and $\tau^o(b_k) = 0$ for $k = r+1 \dots n$ is an optimal solution to $\mathcal{F}^{n,r}$. The cost of τ^o (and thus the cost of $\mathcal{F}^{n,r}$) is $O(\tau^o) = O(\mathcal{F}^{n,r}) = r$. The constraint $\sum_{i=1}^n b_i \geq t$ is a core constraint of \mathcal{F} for all $t = 1 \dots r$, as is $C = \sum_{b \in S} b \geq 1$ for any set $S \subset O$ of literals containing at least $n - r + 1$ variables. To see why C is a core constraint, notice that any solution τ to \mathcal{F} sets at least r of the n literals in O to 1 will also set at least one literal in S to 1 as well.

Given a set \mathcal{C} of core constraints of an instance \mathcal{F} , we say that an assignment $\gamma: \text{VAR}(O) \rightarrow \{0, 1\}$ that satisfies \mathcal{C} is a *hitting set* of \mathcal{C} . A hitting set γ^o is optimal if $O(\gamma^o) \leq O(\gamma)$ holds for all hitting sets of \mathcal{C} . The term hitting set stems from an important special case of core constraints, namely, those of form $C = \sum l \geq 1$. Such constraints are satisfied by setting at least one $l \in C$ to 1, thus *hitting* that constraint. For our purposes, a central property of hitting sets is that they provide lower bounds on $O(\mathcal{F})$.

► **Proposition 3.** Let γ^o , \mathcal{C} and \mathcal{F} be as above. Then $O(\gamma^o) \leq O(\mathcal{F})$.

Proof. Let τ be an optimal solution of \mathcal{F} . Then $\tau(\mathcal{C}) = 1$ by the definition of a core constraint and $O(\gamma^o) \leq O(\tau) = O(\mathcal{F})$ by the optimality of γ^o . ◀

3 Implicit Hitting Sets for Pseudo Boolean Optimization

Algorithm 1 details the PBO-IHS algorithm for computing an optimal solution to a PBO instance \mathcal{F} . In short, the algorithm works by iteratively refining an upper and lower bound on $O(\mathcal{F})$, represented in the pseudocode by UB and LB , respectively. The algorithm also maintains a witness for the upper bound in the form of an assignment τ_{best} for which $O(\tau_{best}) = UB$. The search terminates when $LB = UB$ at which point τ_{best} is returned.

During initialization (Lines 2–5) the lower bound LB and set \mathcal{C} of core constraints of \mathcal{F} are initialized to 0 and \emptyset , respectively. Additionally, an upper bound UB (as well as its witness τ_{best}) is obtained by invoking a PB solver via the function **PB-Solve** on the constraints of \mathcal{F} . The call to **PB-Solve** returns a boolean *sat?* indicating whether or not the constraints in \mathcal{F} are satisfiable and a solution of \mathcal{F} if they are. Note that, if the constraints of \mathcal{F} are not satisfiable, then there do not exist any solutions to \mathcal{F} , so PBO-IHS terminates. Afterwards the main search loop is started.

■ **Algorithm 1** The base IHS algorithm for PBO.

```

1 PBO-IHS( $\mathcal{F}$ )
  Input: A PBO instance  $\mathcal{F}$ 
  Output: An optimal solution  $\tau$ 
2   $(\tau_{best}, sat?) \leftarrow \text{PB-Solve}(\mathcal{F})$ 
3  if not sat? then
4    return “no feasible solutions”
5   $UB \leftarrow O(\tau_{best}); LB \leftarrow 0; \mathcal{C} \leftarrow \emptyset$ 
6  while TRUE do
7     $\gamma \leftarrow \text{Min-Hs}(O, \mathcal{C})$ 
8     $LB \leftarrow O(\gamma)$ 
9    if  $UB = LB$  then break ;
10    $\mathcal{C} \leftarrow \mathcal{C} \cup \text{Extract-Cores}(\gamma, UB, \tau_{best}, \mathcal{F});$ 
11   if  $UB = LB$  then break;
12 return  $\tau_{best}$ 

```

Min-Hs(O, \mathcal{C}):

minimize: $\sum_{(w,l) \in O} w \cdot l$

subject to:

$C \quad \forall C \in \mathcal{C}$

$l \in \{0, 1\} \quad \forall (w, l) \in O$

return:

$\{l \mid l \text{ set to 1 in opt. soln}\} \cup$

$\{\bar{l} \mid l \text{ set to 0 in opt. soln}\}$

(a) An IP for computing an optimal hitting set over a set of core constraints

■ **Figure 1** The implicit hitting set approach to PBO.

■ **Algorithm 2** Extracting multiple core constraints from a single hitting set.

```

1 Extract-Cores( $\gamma, UB, \tau_{best}, \mathcal{F}$ )
2   $\mathcal{A} = \{l \mid l \in O \wedge \gamma(l) = 0\};$ 
3   $\mathcal{C}_n \leftarrow \emptyset;$ 
4  while TRUE do
5     $(sat?, \kappa, \tau) \leftarrow \text{PB-Solve-A}(\mathcal{F}, \mathcal{A});$ 
6    if  $(sat?)$  then
7      if  $O(\tau) < UB$  then  $\tau_{best} \leftarrow \tau; UB \leftarrow O(\tau);$ 
8      return  $\mathcal{C}_n;$ 
9    else  $\mathcal{C}_n \leftarrow \mathcal{C}_n \cup \{\sum_{l \in \kappa} l \geq 1 \mid l \in \kappa\}; \mathcal{A} \leftarrow \mathcal{A} - \kappa;$ 

```

During each iteration of the loop (Lines 6–11), the lower bound is refined by computing an optimal hitting set γ over \mathcal{C} on Line 8. In our implementation, the hitting set is computed by solving the integer program **Min-Hs** detailed in Figure 1a. If the new LB matches the known UB the algorithm terminates on Line 9. Otherwise, the upper bound UB and set \mathcal{C} are next refined by the function **Extract-Cores** detailed in Algorithm 2. After refining the upper bound and extracting new core constraints, the termination criteria is again checked. If the new UB matches the current LB , the algorithm terminates. Otherwise, the loop reiterates.

Extract-Cores computes new core constraints of \mathcal{F} by invoking a PB solver on the constraints of \mathcal{F} under a set \mathcal{A} of *assumptions*. The inputs to **Extract-Cores** is the current hitting set γ of \mathcal{C} , the upper bound UB , its witness τ_{best} and the constraints of \mathcal{F} . The function initialises a set \mathcal{A} to contain all literals in O set to 0 by γ . In other words, initially the set \mathcal{A} contains all literals of O that do not incur cost in γ . A set \mathcal{C}_n of new core constraints is also initialized to \emptyset . New core constraints are then computed by invoking a PB solver via the function **PB-Solve-A**. The function takes as input a set \mathcal{F} of constraints and a set \mathcal{A} of assumptions and then solves the formula $\mathcal{F} \cup \{l = 0 \mid l \in \mathcal{A}\}$. There are two options, either the formula is satisfiable ($sat?$ is true), or it is not ($sat?$ is false). In the first case, the call to **PB-Solve-A** returns a solution τ to \mathcal{F} that sets $\tau(l) = 0$ for all $l \in \mathcal{A}$. Then $O(\tau)$ is compared to UB which is updated if needed. Afterwards **Extract-Cores** terminates and

returns the set \mathcal{C}_n of new core constraints found. In the second case, $\kappa \subset \mathcal{A}$ is a set of literals for which $\mathcal{F} \cup \{l = 0 \mid l \in \kappa\}$ is also unsatisfiable. This implies that $\sum_{l \in \kappa} l \geq 1$ is a core constraint of \mathcal{F} so it is added to \mathcal{C}_n . The literals in κ are then removed from \mathcal{A} and the loop reiterated.

The following theorem establishes the correctness of PBO-IHS.

► **Theorem 4.** *Given an input PBO instance \mathcal{F} PBO-IHS terminates and returns an optimal solution τ to \mathcal{F} .*

Proof. (Sketch) To show that τ is optimal we note that $O(\tau) = O(\gamma)$ for an optimal hitting set γ over a set \mathcal{C} of core constraints of \mathcal{F} , which by Proposition 3 implies $O(\tau) \leq O(\mathcal{F})$.

To show that PBO-IHS terminates, we first show that each call to **Extract-Cores** terminates. This follows from each invocation of **PB-Solve-A** either resulting in termination of **Extract-Cores**, or elements being removed from \mathcal{A} and the fact that, by the check on Line 2, the call **PB-Solve-A**(\mathcal{F}, \emptyset) returns satisfiable.

For the final part of the argument, we say that a hitting set γ returned on Line 7 is feasible if **PB-Solve-A**($\mathcal{F}, \{l \mid l \in O \wedge \gamma(l) = 0\}$) is satisfiable, otherwise it is infeasible. We note that, as soon as a feasible hitting set γ is computed by **Min-Hs**, **PB-Solve-A** will find a solution τ for which $O(\tau) = O(\gamma) = LB$ in the first iteration of **Extract-Cores** and PBO-IHS will terminate. As there only are a finite number of possible hitting sets, we thus only need to show that a fixed infeasible hitting set γ^I can be computed at most once by **Min-Hs**. This follows from the fact that γ^I being infeasible implies that the invocation of **Extract-Cores** will add (at least) one new core constraint $\sum_{l \in \kappa} l \geq 1$ for some $\kappa \subset O \setminus \{l \mid \gamma(l) = 1\}$ into the set \mathcal{C} . Thus γ^I will not be a hitting set in subsequent iterations. ◀

We end this section with an example demonstrating the execution of PBO-IHS.

► **Example 5.** Invoke PBO-IHS on the instance $\mathcal{F}^{5,2}$ from Example 2 with $n = 5$ and $r = 2$. Assume that the first solution obtained on line 2 is $\tau_{best} = \{b_1, b_2, b_3, b_4, \bar{b}_5\}$. The initial upper bound is set to $UB = O(\tau_{best}) = 4$. In the first iteration of the search loop, there are no core constraints to satisfy. As such, the first call to **Min-Hs** returns $\gamma = \{\bar{b}_1, \bar{b}_2, \bar{b}_3, \bar{b}_4, \bar{b}_5\}$. As $O(\gamma) = 0$, the lower bound LB is not improved and the algorithm moves on to invoke **Extract-Cores**. The set \mathcal{A} is initialized to $\{b_1, b_2, b_3, b_4, b_5\}$. The first call to **PB-Solve-A** returns unsatisfiable. There are a number of subsets of the assumptions that could be returned, let $\kappa = \{b_1, b_2, b_3, b_4\}$ be the one obtained. Before the next call, the set \mathcal{A} is refined to $\{b_5\}$ and the core constraint $\sum_{i=1}^4 b_i \geq 1$ is added to \mathcal{C}_n . The next call returns satisfiable, returning for example the solution $\tau = \{b_1, b_2, b_3, \bar{b}_4, \bar{b}_5\}$. The solution has $O(\tau) = 3$ so the upper bound and τ_{best} are updated before **Extract-Cores** terminates. At this point $UB = 3 \neq 0 = LB$ so PBO-IHS does not terminate.

In the next iteration, the call to **Min-Hs** is done with $\mathcal{C} = \{\sum_{i=1}^4 b_i \geq 1\}$. Assume the call returns $\gamma = \{b_1, \bar{b}_2, \bar{b}_3, \bar{b}_4, \bar{b}_5\}$. The lower bound LB is now updated to 1 and the function **Extract-Cores** is again invoked. This time around, the first call to **PB-Solve-A** is done with $\mathcal{A} = \{b_2, b_3, b_4, b_5\}$. The first call is unsatisfiable, the only subset of assumptions that can be returned is $\kappa = \{b_2, b_3, b_4, b_5\}$. The next call to **PB-Solve-A** will return satisfiable. Assume that this time a solution $\tau = \{b_1, b_2, \bar{b}_3, \bar{b}_4, \bar{b}_5\}$ is returned. The solution has $O(\tau) = 2$ so the upper bound is again updated.

At this point, PBO-IHS has found an optimal solution of $\mathcal{F}^{5,2}$. However, since $UB = 2 > 1 = LB$, the algorithm does not terminate. Informally speaking, the algorithm has found an optimal solution, but not proven its optimality. The “proof” of optimality is obtained once **Min-Hs** returns a hitting set γ with $O(\gamma) = 2$, which in turn happens after enough

core constraints have been extracted for $\mathcal{C} = \{(b_1 + b_2 + b_3 + b_4 \geq 1), (b_1 + b_2 + b_3 + b_5 \geq 1), (b_1 + b_2 + b_4 + b_5 \geq 1), (b_1 + b_3 + b_4 + b_5 \geq 1), (b_2 + b_3 + b_4 + b_5 \geq 1)\}$. In other words for each b_i \mathcal{C} should contain at least one constraint in which b_i does not appear. Then **Min-Hs** returns a hitting set γ with $O(\gamma) = 2$, which updates $LB = 2$ and allows the algorithm to terminate.

4 Search Techniques and Refinements

We move on to describing a number of refinements and additional heuristics to **PBO-IHS**. We will later on empirically evaluate the impact of each of the techniques on the runtime performance of **PBO-IHS**.

Many of the refinements we consider are based on techniques first proposed for the IHS algorithm in the context of MaxSAT. These are motivated by the fact that, in order for **PBO-IHS** to terminate, the lower bound LB needs to be set to the optimal cost $O(\mathcal{F})$ of the instance \mathcal{F} that is being solved. This in turns means that the **Min-Hs** subroutine should compute a hitting set γ for which $O(\gamma) = O(\mathcal{F})$. In fact, by adapting a well known result from MaxSAT, we can show that there are families of instances on which **PBO-IHS** as presented in Section 3 requires an exponential number of core constraints from **Extract-Cores** in order to terminate.

► **Proposition 6** (Adapted from [16]). *For every even $n \in \mathbb{N}$ there exists a PBO instance \mathcal{F}_n for which **Extract-Cores** needs to extract $\Omega(2^n)$ core constraints before **PBO-IHS** terminates.*

Proof. (Sketch) Let $r = n/2$ and $\mathcal{F}_n = \mathcal{F}^{n,r}$ from Example 2 and, following similar reasoning as in [16] in the context of MaxSAT, to show that in order for **Min-Hs** to compute a hitting set γ with $O(\gamma) = n/2 = O(\mathcal{F}^{n,r})$, **Extract-Cores** needs to extract at least one core constraint of form $\sum_{l \in S} l \geq 1$ for each subset $S \subset O$ with $n - r + 1$ literals.

More precisely, if there exists a subset $S_p \subset O$ with $n - r + 1$ elements for which $\left(\sum_{l \in S_p} l \geq 1\right) \notin \mathcal{C}$, then the solution $\gamma = \{\bar{l} \mid l \in S_p\} \cup \{l \mid l \notin S_p\}$ is a hitting set over \mathcal{C} that has $O(\gamma) = n - (n - r + 1) = r - 1 < r = O(\mathcal{F}^{n,r})$. As a consequence, the minimum-cost hitting set γ computed by **Min-Hs** will have $O(\gamma) < O(\mathcal{F}^{n,r})$ and the algorithm will not terminate. In other words, **Extract-Cores** will need to extract at least $\binom{n}{r+1}$ core constraints before **Min-Hs** computes a hitting set γ with $O(\gamma) = O(\mathcal{F}^{n,r})$. ◀

In light of Proposition 6 we expect any technique for deriving more core constraints of an instance to improve on the empirical performance of **PBO-IHS**. In this work, we consider the following techniques.

4.1 Constraint Seeding

In constraint seeding, the input instance \mathcal{F} is scanned for constraints that only contain variables that appear in the objective function. Such constraints trivially satisfy the second requirement of Definition 1 and as such are core constraints of \mathcal{F} . Any such constraints are added to \mathcal{C} prior to starting the main search loop (Lines 6-11 of Algorithm 1). While a similar technique is employed in MaxSAT solving, in the context of PBO we can show that constraint seeding can have a significant effect on the number of core constraints that **PBO-IHS** needs to extract before termination.

► **Example 7.** Consider again the instance $\mathcal{F}^{5,2}$ from Example 2. On this instance constraint seeding is able to detect the core constraint $C = \sum_{i=1}^5 b_i \geq 2$ and add it to \mathcal{C} . Assume that the first solution τ_{best} obtained on line 2 is $\{b_1, b_2, b_3, b_4, \bar{b}_5\}$ implying an initial UB of 4. In the

■ **Algorithm 3** Computing multiple core constraints with weight-aware core extraction.

```

1 Extract-Cores-WCE( $\gamma, UB, \tau_{best}, \mathcal{F}$ )
2    $\mathcal{C}_n \leftarrow \emptyset; \mathcal{W} \leftarrow \emptyset;$ 
3   for  $(w, l) \in O$  do
4     if  $\gamma(l) = 1$  then  $\mathcal{W}(l) = 0;$ 
5     else  $\mathcal{W}(l) = w;$ 
6   while TRUE do
7      $(sat?, \kappa, \tau) \leftarrow \text{PB-Solve-A}(\mathcal{F}, \{l \in O \mid \mathcal{W}(l) > 0\});$ 
8     if  $(sat?)$  then
9       if  $O(\tau) < UB$  then  $\tau_{best} \leftarrow \tau; UB \leftarrow O(\tau);$ 
10      return  $\mathcal{C}_n;$ 
11    else
12       $\mathcal{C}_n \leftarrow \mathcal{C}_n \cup \{\sum_{l \in \kappa} l \geq 1 \mid l \in \kappa\};$ 
13       $w^\kappa = \min_{l \in \kappa} \{\mathcal{W}(l)\};$ 
14      for  $l \in \kappa$  do  $\mathcal{W}(l) \leftarrow \mathcal{W}(l) - w^\kappa;$ 

```

first iteration of the search loop, the core constraint C added by seeding results in the hitting set γ computed on Line 7 assigning at least two variables to 1. Assume $\gamma = \{b_1, b_2, \bar{b}_3, \bar{b}_4, \bar{b}_5\}$. The LB is then refined to 2 and the function **Extract-Cores** invoked. In the first iteration of **Extract-Cores**, the function **PB-Solve-A** is invoked with $\mathcal{A} = \{b_3, b_4, b_5\}$. The result is satisfiable and the function returns the assignment $\tau = \{b_1, b_2, \bar{b}_3, \bar{b}_4, \bar{b}_5\}$. Since $O(\tau) = 2$ the UB is then updated and search terminated.

The example combined with Proposition 6 implies the following.

► **Proposition 8.** *For every even $n \in \mathbb{N}$ there exists a PBO instance \mathcal{F}_n on which the **Extract-Cores** subroutine of PBO-IHS extracts $\Omega(2^n)$ cores before termination if constraint seeding is not used and no cores if seeding is used.*

We observe an interesting connection between constraint seeding and *abstract cores*, a recently proposed improvement to the IHS algorithm for MaxSAT [6]. Abstract cores are a compact representation of a large number of ordinary core constraints. More specifically, an abstraction variable $ab.c[k]$ defined over a set of n literals $ab \subset O$ that all have the same coefficient in O has the definition $ab.c[k] \leftrightarrow \sum_{l \in ab} l \geq k$, i.e., the linear constraints $\sum_{l \in ab} l - k \cdot ab.c[k] \geq 0$ and $\sum_{l \in ab} l - n \cdot ab.c[k] < k$. Let Abs be a set of abstraction variables. An abstract core constraint C is a linear constraint for which $\text{VAR}(C) \subset \text{VAR}(O) \cup Abs$ that is satisfied by any assignment that satisfies both \mathcal{F} and the definitions of the abstraction variables. Each such constraint containing an abstraction variable $ab.c[k]$ corresponds to $\binom{n}{n-k+1}$ (non-abstract) core constraints of form $\sum_{l \in C, l \neq ab.c[k]} l + \sum_{l \in ab_k} l \geq 1$ where $ab_k \subset ab$ is any subset containing $n - k + 1$ variables.

A central motivation for abstract cores in the context of MaxSAT is that the IHS algorithm for MaxSAT needs to extract an exponential number of cores when solving the CNF translation of the instance presented in Example 2. As demonstrated by Example 7, the technique of constraint seeding in PBO already allows avoiding the need to extract a large number of core constraints on this specific instance.

4.2 Weight-Aware Core Extraction

Weight-aware core extraction (WCE), first proposed in the context of core-guided MaxSAT solving in [7], is a technique for extracting more core constraints from a single hitting set by using information provided by the coefficients of the objective variables. The idea has previously been explored in the context of PBO under the name independent cores in [21]. Here we employ WCE for the first time in the context of IHS.

Algorithm 3 details **Extract-Cores-WCE**, the computation of new core constraints with WCE. Given an instance \mathcal{F} and a hitting set γ , the procedure initializes a weight $\mathcal{W}(l)$ for each objective function literal $l \in \mathcal{O}$. The weight of l equals its coefficient in \mathcal{O} if $\gamma(l) = 0$ and 0 otherwise. Each call to **PB-Solve-A** is then performed with a set of assumptions containing all literals for which $\mathcal{W}(l)$ is not 0. Note specifically that the first set of assumptions will be same with and without employing WCE. After a subset κ of assumptions is obtained from the PB oracle, the weight of each literal $l \in \kappa$ is lowered by w^κ , the minimum weight among all literals in κ . Importantly, this lowers the weight of at least one literal to 0, thus guaranteeing the eventual termination of **Extract-Cores-WCE**.

The intuition underlying WCE is that it allows for extracting not only a (variable) disjoint set of core constraints from each hitting set, but also core constraints whose variables intersect on a subset containing literals with large coefficients. The following example demonstrates how WCE can decrease the number of hitting sets that the IHS algorithm needs to compute before termination.

► **Example 9.** Consider an instance $\mathcal{F} = \{(b_1 + b_N \geq 1), (b_2 + b_N \geq 1), \dots, (b_n + b_N \geq 1)\}$ with $\mathcal{O} = \sum_{i=1}^{n-1} b_i + nb_N$. Invoke **Extract-Cores** (Algorithm 2) on \mathcal{F} with $\gamma = \emptyset$. In the first iteration, **PB-Solve-A** is invoked with $\mathcal{A}_1 = \{b_1, \dots, b_n, b_N\}$. Since any set κ that could be returned contains b_N it will be removed from the set of assumptions after one core has been computed. Since an assignment setting $b_N = 1$ satisfies the instance, **Extract-Cores** can only compute a single new core constraint before terminating. With WCE (i.e., **Extract-Cores-WCE**) the situation changes. The initial set of assumptions will again be \mathcal{A}_1 . Since any set κ returned by **PB-Solve-A** will have $w^\kappa = 1$, the weight \mathcal{W} of b_N is lowered by 1 and thus remains positive. Hence b_N will stay in the assumptions until either (i) n core constraints have been extracted or (ii) all other literals are removed from the set of assumptions.

4.3 Non-Optimal Hitting Sets

At early stages of IHS search, when \mathcal{C} only contains a few core constraints, we expect $\mathcal{O}(\gamma) < \mathcal{O}(\mathcal{F})$ to hold for an optimal hitting set γ over \mathcal{C} . Recalling that **PBO-IHS** can terminate only when $LB = \mathcal{O}(\mathcal{F})$, this implies that we do not expect an optimal hitting set over \mathcal{C} to result in termination before enough cores have been extracted. However, the **Extract-Cores** subroutine does not necessarily need an *optimal* hitting set in order to compute new core constraints. Hence instead of spending time computing a – potentially useless – optimal hitting set, we can instead focus our efforts on computing any hitting set that allows **Extract-Cores** to derive more core constraints.

More precisely, we terminate **Min-Hs** once an incumbent hitting set γ^i is obtained which is either optimal or satisfies $\mathcal{O}(\gamma^i) < UB$. Even if the lower bound LB can only be updated if γ^i is optimal, **Extract-Cores** will still either derive a new core constraint, or find a solution τ for which $\mathcal{O}(\tau) = \mathcal{O}(\gamma^i) < UB$. In both cases, the search progresses toward an optimal solution. The only way in which γ^i can be rediscovered in subsequent iterations is if it was in fact optimal. More formally, we can show that the requirement of the hitting set being

computed by **Min-Hs** either being optimal, or having cost lower than the current UB is sufficient for the correctness of **PBO-IHS**. This follows from the fact that each non-optimal hitting set can be computed by **Min-Hs** at most once and each optimal one at most twice. For a more detailed argument in the context of MaxSAT, we refer the reader to [2].

4.4 Core Shrinking through Shuffling Assumptions

The sizes of core constraints found during iterations of IHS directly impact the tightness of the hitting set constraints. In IHS MaxSAT solving, subset-minimization of cores is done by iteratively asking the SAT solver performing cores extraction whether some soft clauses can be removed from the cores while maintaining unsatisfiability. However, in the context of PBO, we observed that subset-minimization of cores through the PB solver during IHS search often becomes too time-consuming, and hence we do not – at least currently – attempt to subset-minimize cores in this way before turning to the hitting set solver. Instead, we make use of another, computationally less demanding way of potentially identifying smaller cores. In particular, at the time of termination of the PB solver (the **PB-Solve-A** subroutine of Algorithm 2) at a specific iteration, the subset of assumptions from which the core constraint is formed is obtained by propagating all assumptions one by one until the solver reports unsatisfiable. A central fact to note is that the specific core constraint obtained will depend on the order in which assumptions are propagated; other orders of propagating the assumptions during this “analyzeFinal” phase may provide at times smaller cores. With this aim, we randomly shuffle the order of the assumptions a number of times (set to 20 repetitions in our current implementation), and choose a smallest-cardinality core among the cores obtained this way as the core constraint that is then added to the hitting set solver. Since this shuffling approach to shrinking cores relies only on polynomial-time propagation within the PB solver, it avoids the worst-case exponential subset-minimization calls if core shrinking would be performed by iteratively asking the PB solver to identify assumptions that can be left out from a found core.

4.5 Reduced Cost Fixing

The hybrid approach of **PBO-IHS** combining IP solving and PB reasoning opens up the possibility of introducing techniques from IP solving into the PB reasoning part of **PBO-IHS**. One such technique that we consider in this work is *reduced cost fixing*, a standard technique in the realm of IP solving [14, 15, 32]. In IHS for PBO, reduced cost fixing can be applied in two ways: on the LP relaxation of the actual PBO instance, and on the level of solving the hitting set programs using IP solving. In the context of IHS for MaxSAT and in particular on the level of the hitting set IP, reduced cost fixing was first explored in [2].

First consider employing reduced costs obtained from solving the hitting set problems during IHS search. For a set \mathcal{C} of core constraints and an objective function O , let $\text{Min-Hs}(O, \mathcal{C})^{LP}$ be the LP relaxation of the IP depicted in Figure 1a, i.e., the linear program obtained by removing the requirement of the l variables being integral, and instead allowing them to take any value in the range $[0, 1]$. Informally speaking, given a solution η to $\text{Min-Hs}(O, \mathcal{C})^{LP}$, the reduced cost $rc(b)$ of a variable assigned to 1 (0) by η measures the effect that assigning b to 0 (1) instead would have on $O(\eta)$. Since the optimal cost of $\text{Min-Hs}(O, \mathcal{C})^{LP}$ is a lower bound on the optimal cost $\text{Min-Hs}(O, \mathcal{C})$ which in turn is a lower bound on $O(\mathcal{F})$, the reduced costs of a variable b in the objective function can sometimes be used to show that either $b = 1$ or $b = 0$ holds for at least one optimal solution to \mathcal{F} , which allows us to fix the value of b for the rest of the search.

More precisely, suppose τ_{best} is a feasible solution to \mathcal{F} and consider a non-basic variable x (i.e., a variable assigned to either 0 or 1 by η) of $\text{Min-Hs}(\mathcal{O}, \mathcal{C})^{LP}$. If $\eta(x) = 0$ and either: (i) $\mathcal{O}(\eta) + rc(x) > \mathcal{O}(\tau_{best})$ or (ii) $\mathcal{O}(\eta) + rc(x) = \mathcal{O}(\tau_{best})$ and $\tau_{best}(x) = 0$, then x is fixed to 0 in subsequent iterations of the PBO-IHS algorithm. Similarly, if $\eta(x) = 1$ and either: (i) $\mathcal{O}(\eta) - rc(x) > \mathcal{O}(\tau_{best})$ or (ii) $\mathcal{O}(\eta) - rc(x) = \mathcal{O}(\tau_{best})$ and $\tau_{best}(x) = 1$, then x is fixed to 1 in subsequent iterations. We emphasise, that in both cases, the variable is fixed both in the **Min-Hs**, and the **Extract-Cores** subroutines.

A detailed argument for the correctness of reduced cost fixing in implicit hitting set-based MaxSAT can be found in [2]. We sketch the proof of the case $\eta(x) = 0$. First note that if $x = 1$ is infeasible for the LP relaxation of **Min-Hs**, then it will be infeasible for the IP as well. In other words, then no hitting set over \mathcal{C} can set $x = 1$ and, by the definition of a core constraint, neither can any solution to \mathcal{F} . On the other hand, if $x = 1$ is feasible, then by the properties of reduced costs [4], any solution η^m to the LP for which $\eta^m(x) = 1$ will have $\mathcal{O}(\eta^m) \geq \mathcal{O}(\eta) + rc(x) \geq \mathcal{O}(\tau_{best}) \geq \mathcal{O}(\mathcal{F})$. Since the LP is a relaxation of the IP and the costs of the optimal solutions γ^o of the IP have $\mathcal{O}(\gamma^o) \leq \mathcal{O}(\mathcal{F})$, it follows that fixing $x = 0$ can be done without removing an optimal solution of the IP.

Secondly, we note that the LP relaxation of the input PBO instance itself can be solved for obtaining bounds information already before the IHS search, complementary to the information obtained from reduced costs from the hitting set computations during search. In particular, for obtaining reduced costs information on an input PBO instance \mathcal{F} , solve the LP relaxation \mathcal{F}^{LP} of \mathcal{F} prior to starting the main search loop, and apply reduced cost fixing based on the reduced costs obtained from an optimal solution η^i of \mathcal{F}^{LP} whenever the IHS search improves the upper bound UB during search.

5 Empirical Evaluation

We turn to overviewing results from an empirical evaluation of the IHS approach to PBO presented in this work. The experiments reported on were run on nodes with 8-core Intel Xeon E5-2670 2.6-GHz CPUs and 64-GB RAM. We set a per-instance 3600-second time and 16-GB memory limit.

5.1 Implementation

We implemented PBO-IHS in Python, with a PB solver (as the core extractor) and an integer programming solver (as the hitting set solver) imported as external modules. We use the Roundingsat version 2 [24] (commit 1476bf0bcd) as the PB solver, using its most recent configuration as described in [20]. To implement the **PB-Solve-A** function, we extended the Roundingsat implementation to include an `analyzeFinal` function similar to the one implemented in the MiniSat SAT solver [22, 23], so that we can call Roundingsat within PBO-IHS under assumptions and extract unsatisfiable cores over the assumptions. As the integer programming solver for hitting set computations we used IBM ILOG CPLEX C++ API version 12.8. We compiled both Roundingsat and CPLEX API components using pybind11, which is a utility that allows to compile C++ libraries as python modules. In the following, we will refer as PBO-IHS to our implementation of the IHS approach to PBO which applies HS reduced cost fixing, constraint seeding, assumption set shuffling, non-optimal hitting sets and weight-aware core extraction, but does not apply reduced cost fixing based on solving the LP relaxation of the input PBO instance and does not employ abstract cores. (To this end, we will also report on the marginal contribution of each of these search techniques on the overall performance of PBO-IHS.) For the experiments, our implementation of PBO-IHS runs single-threadedly. The PBO-IHS implementation is available in open source at <https://bitbucket.org/coreo-group/pbo-ih-solver/>.

5.2 Alternative Approaches

We extensively compare the empirical performance of PBO-IHS to those of previously proposed specialized approaches to PBO:

- **Open-WBO** [30] encodes the PBO instance into a MaxSAT instance by transforming the PB constraints into CNF by the well-known (generalized) totalizer encoding [29]. The MaxSAT instance is then solved with the OLL algorithm for MaxSAT [31].
- **Sat4J** [8] generalizes the CDCL procedure for SAT solving to PB solving and the cutting planes proof system. The cutting planes reasoning is implemented using the weakening and saturation rules similar to [10]. Computing an optimal solution to an instance \mathcal{F} is done by *solution improving search*, i.e., starting from $ub = \infty$ iteratively invoking the solver on the formula $\mathcal{F} \cup \{\sum_{(w,l) \in O} wl < ub\}$ which is satisfiable by an assignment τ iff τ is a solution to \mathcal{F} with $O(\tau) < ub$. When such τ is found, ub is updated to $O(\tau)$ and the loop reiterated. The search terminates when the solver reports the formula to be unsatisfiable, at which point the last found (optimal) solution is returned.
- **NaPS** [40] encodes the PBO instance into a MaxSAT instance using binary decision diagrams (BDDs). An optimal solution to the MaxSAT instance is then computed a combination of solution improving and binary search.
- **Roundingsat (RS)** [24] generalizes the CDCL procedure for SAT solving to PB solving and the cutting planes proof system. Cutting planes reasoning is implemented using the division and rounding rules. Optimization is then done by solution improving search.
- **RS/lp** [20], a version of RS that periodically invokes a linear programming (LP) solver on the LP relaxation of the instance being solved. The LP calls are used to derive more conflicts to the CDCL procedure implemented in basic roundingsat. For example, if there are no feasible solutions to the LP relaxation of the instance under the current partial assignment, then there will not be any feasible solutions to the PB instance either. Computing an optimal solution is done by solution improving search.
- **RS/oll** [21], a version of RS that combines the solution improving search with an extension of the OLL algorithm to PBO [1]. OLL is a lower bounding approach that extracts core constraints of the instance being solved. Based on the obtained constraints, the instance is then relaxed in a way that allows – in a controlled way – one more of the literals in the objective function to be set to 1 in subsequent iterations.

In addition to these academic specialized PBO solvers, we also investigate how PBO-IHS fares against **CPLEX** [13].

5.3 Benchmarks

For the experiments, we collected a large number of benchmarks from different sources. Firstly, we collected all benchmarks used in Pseudo-Boolean Competition 2016 [35] (so far the most recent instantiation of the competition) as well as benchmarks available on the competition website that were used in previous competition instantiations since 2005. Secondly, we collected all 0-1 integer programs from the MIPLIB 2017 library [26] as well as earlier MIPLIB releases. We filtered out 7914 benchmark instances that had no objective function and 249 unsatisfiable benchmarks which do not admit solutions, as uninteresting for benchmarking optimization solvers, as well as 206 benchmarks that have at least one coefficient with an absolute value higher than 2^{64} and 548 benchmarks with non-linear constraints or non-linear terms in their objective functions. Starting from 17312 Pseudo-Boolean Competition benchmarks and 1273 MIPLIB benchmarks, respectively, after filtering we were left with 8456 and 252 benchmarks, respectively, giving a total of 8708 benchmarks that we used in the experiments reported on here.

We categorized to the best of our knowledge the benchmarks (based on their source, related publications, and finally, by file names) into different problem domains, obtaining the problem domain categorization shown in Table 1. We observe that the whole benchmark set is significantly unbalanced in terms of the number of instances originating from specific problem domains. For a fair comparison of the overall performance of the different solvers across the different benchmark domains, we sampled at random (without repetition) from each problem domain 30 instances (or all of the instances from the domain, if the domain included less than 30 instances) for the comparison. The sampled benchmark set contains in total 1786 benchmarks. Unless explicitly stated otherwise, all results reported on in this section are with respect to the sampled benchmark set.

5.4 Results: Comparison with Specialized PBO Solvers

We first compare the empirical performance of PBO-IHS to those of other specialized PBO solvers on the sampled benchmark set. Figure 2(top) shows how many benchmarks each solver was able to solve (y-axis) under different per-instance time limits. We observe that PBO-IHS outperforms all of the other specialized solvers. The two recent variants of Roundingsat perform the second and third best; in particular, PBO-IHS also outperforms the version of Roundingsat (RS/lp) which is used within PBO-IHS for core extraction. To justify the sampling of benchmarks in order to achieve a balanced benchmark set, confirmed the results for the three best-performing solvers under 10 different random samplings. The results, shown in Figure 2(bottom), confirm that the relative performance of the solvers is robust against subsampling benchmarks in a balanced way. In more detail, For each solver S , Figure 2(bottom) includes 3 lines: S -max, S -median and S -min. A point (t, x) on the S -max line indicates that S was able to solve x benchmarks within t seconds for *at least one* of the ten benchmark set samples. Analogously, a point on the S -min line indicates solving x benchmarks within t seconds in *all* samples, and the S -median line indicates solving x benchmarks within t in *five* of the 10 samples.

More detailed data per benchmark domain (over the *full* benchmark set) is reported in Table 1, with the number of instances solved (left column) and the cumulative runtimes over solved instances (right column) shown for each solver, with all benchmarks from each problem domain included. Interestingly, we observe that the relative performance of the Roundingsat versions (RS/lp and RS/oll) and PBO-IHS depends significantly on the problem domain, suggesting that the approaches complement each other.

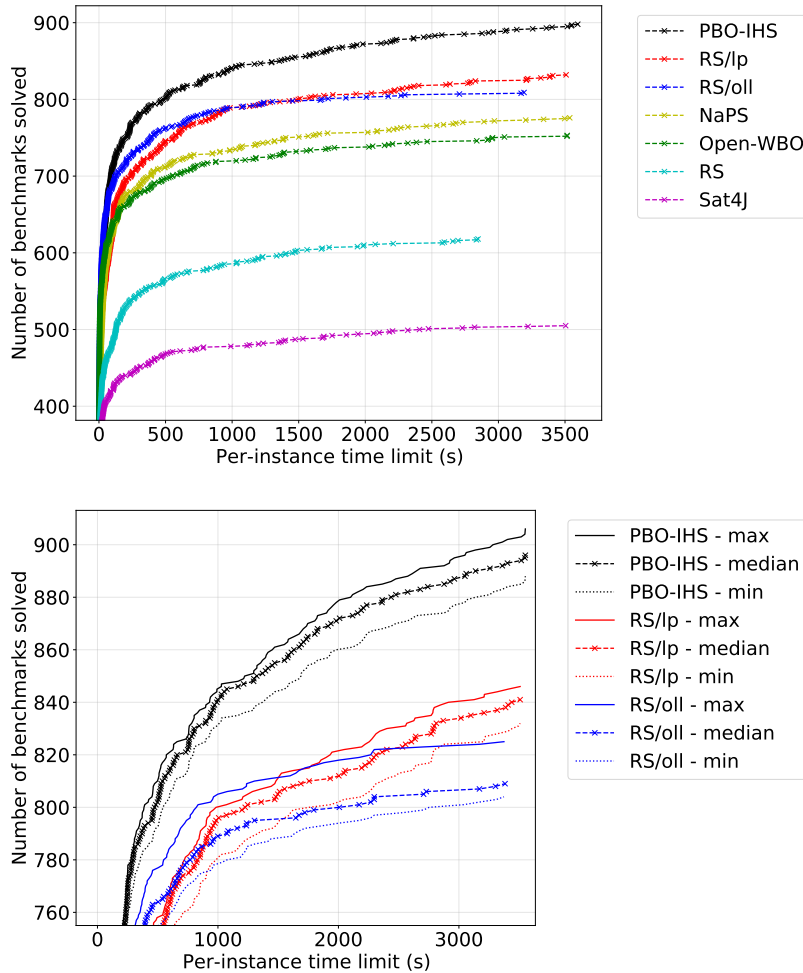
5.5 Results: Impact of Different Search Techniques in PBO-IHS

We also investigated the marginal impact of the different search techniques and refinements to PBO-IHS on the empirical performance of PBO-IHS. Figure 3(top) provides a comparison of the default configuration of PBO-IHS (with HS reduced cost fixing (hs-rc), constraint seeding, assumption set shuffling, non-optimal hitting sets, weight-aware core extraction, but without reduced cost fixing based on solving the LP relaxation of the input PBO instance (pb-rc) or abstract cores) to configurations of PBO-IHS with each of HS reduced cost fixing, constraint seeding, assumption set shuffling, non-optimal hitting set computation, and weight-aware core extraction separately switched off, as well the configurations using reduced cost fixing on the PBO LP and abstract cores separately. We observe that constraint seeding makes the largest positive marginal contribution to the empirical performance of PBO-IHS, and assumption set shuffling second largest positive marginal contribution. The third largest positive contribution is made by using non-optimal hitting sets, followed closely by weight-

aware core extraction. The use of abstract cores, at least as currently implemented, makes a significant negative marginal contribution, noticeably degrading the performance of the default version of PBO-IHS. Exploring the relationship between constraint seeding and abstract cores in PBO, as well as alternative instantiations of the abstract cores framework, remains interesting for future work. The two different forms of reduced cost have only a very modest impact. While reduced cost fixing based on the PBO LP does not make a significant negative marginal contribution, it does not appear to improve on the performance of PBO-IHS, which justifies disabling it together with abstract cores in the default configuration of PBO-IHS.

5.6 Results: Runtime Division between Core Extraction and MCHS

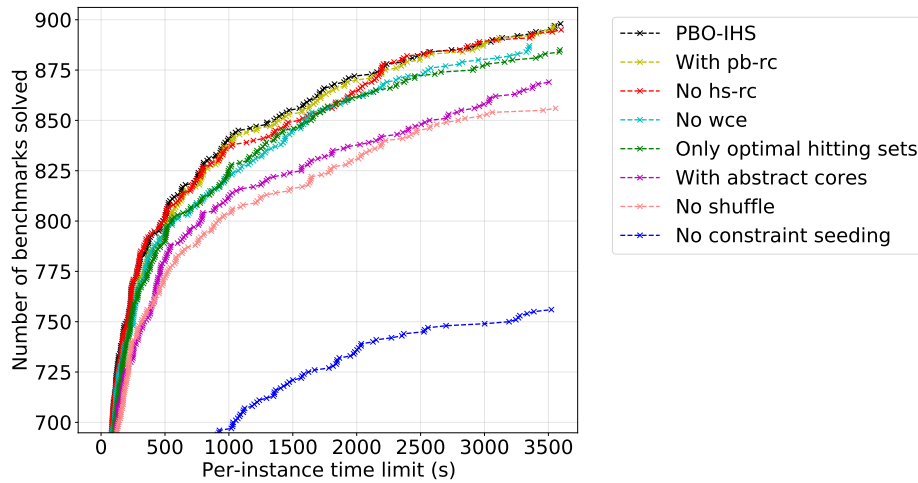
Figure 4(left) details the fraction of solving time spent in the `Min-Hs` subroutine of PBO-IHS on the 898 of the instances solved within the time limit. Note that since the `Min-Hs` and `PB-Solve-A` subroutines dominate the running time of PBO-IHS, the rest of the runtime is effectively spent in `PB-Solve-A`. We observe that on most of the instances, over 80% of the



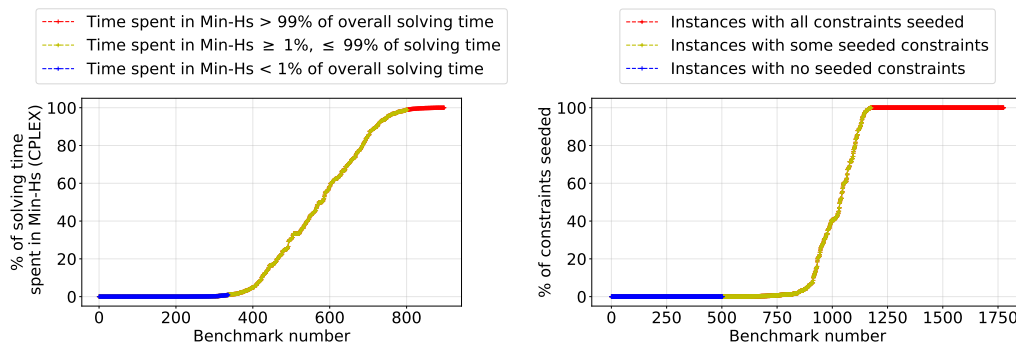
■ **Figure 2** Top: Runtime comparison of specialized PBO solvers. Bottom: Confidence intervals over 10 benchmark subset samples for the three best-performing solvers.

■ **Table 1** Comparison of specialized PBO solver per benchmark domain: number of solved instances (#) and cumulative runtimes over solved instances in seconds (cum.)

	Sat4J		RS		Open-WBO		Naps		RS/lp		RS/oll		PBO-IHS	
Domain (#instances)	#	cum.	#	cum.	#	cum.	#	cum.	#	cum.	#	cum.	#	cum.
10orplus/9orless (156)	55	99459	39	64252	156	202	156	14149	154	55344	156	1406	156	23670
caixa (24)	24	13	20	16	24	2	24	179	24	70	24	3	24	64
rand.*list (118)	113	5241	59	1961	118	44	118	2218	118	692	118	125	118	2296
area_* (59)	11	626	37	11998	59	138	54	3613	54	16176	57	9469	51	11784
trarea_ac (18)	1	1	1	2	13	2314	4	4582	16	3722	5	1868	18	7751
aries-da_nrp (70)	15	1747	16	7994	25	11938	19	7325	43	15442	21	10599	32	10413
BA (1440)	85	175161	301	221066	160	116377	0	0	588	472938	356	230143	20	30038
NG (960)	2	804	59	71042	11	11990	0	0	48	115499	138	194128	0	0
MANETs (150)	29	5744	0	0	20	13648	14	17875	40	23547	29	9525	25	21152
BioRepair (30)	30	457	30	8551	30	105	30	311	30	3258	30	35	30	262
Metro (30)	30	4413	30	1270	30	3341	30	775	30	1795	29	3291	27	12595
ShiftDesign (30)	12	2258	16	5671	28	10696	30	2781	18	12824	27	3371	9	9060
Timetabling (30)	17	11920	15	8026	27	10054	25	17502	23	15419	24	3295	28	8768
EmployeeScheduling (14)	0	0	0	0	9	480	9	506	0	0	0	0	0	0
golomb-rulers (34)	14	642	14	5765	11	1656	12	3451	12	1216	12	436	12	4212
bsg (60)	0	0	10	156	10	4767	10	813	10	465	10	1963	5	16
mis/mds (120)	0	0	44	8968	48	6605	47	6245	45	3853	45	5525	57	15335
course-ass (6)	0	0	2	1225	2	29	4	3226	3	33	2	1	1	6
decomp (10)	0	0	0	0	8	1809	8	4516	0	0	2	2200	0	0
data (68)	1	2	8	1628	0	0	4	2414	13	4044	13	5837	11	2163
dt-problems (60)	37	1712	40	3573	38	2777	59	8697	60	2	60	7	60	113
domset (15)	0	0	0	0	0	0	0	0	0	0	0	0	0	0
factor (186)	186	56	186	0	186	710	186	160	186	2	186	0	186	342
factor-mod-B (225)	0	0	225	67	199	39899	225	3243	225	60	225	25	225	344
ftcp (35)	2	36	2	0	1	141	6	468	5	940	5	2	12	499
featureSubscription (20)	20	1266	20	2492	20	76	20	112	20	8106	20	941	20	303
frbXX-XX-opb (40)	0	0	0	0	0	0	17	11552	0	0	0	0	6	11343
flexray (9)	5	1697	4	83	4	496	4	296	4	393	4	31	4	50
fome (3)	0	0	0	0	0	0	0	0	0	0	0	0	0	0
graca (100)	20	230	24	3664	97	4687	98	10487	31	21769	93	14428	84	40593
haplotype (8)	0	0	8	202	8	31	8	60	8	2385	8	57	7	4023
garden (7)	4	28	5	1	6	94	6	355	5	1	5	0	6	76



■ **Figure 3** Runtime comparison of various PBO-IHS variants.



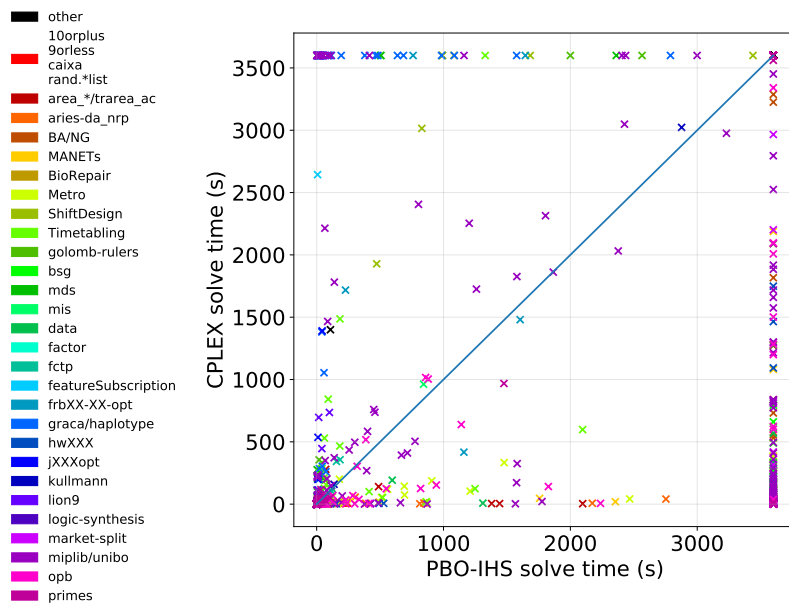
■ **Figure 4** Left: Ratio of solving time spent by PBO-IHS in `Min-Hs` subroutine for solved benchmarks. Right: Ratio of constraints seeded on all benchmarks.

overall solving time is spent computing core constraints: on 462 of the 893 instances, only 20% of the time was spent in `Min-Hs`, and one over 1/3 of the instances 99% of the overall solving time is spent in `PB-Solve-A` (marked by the blue line). On the other hand, the runtimes of `Min-Hs` dominates on approximately 1/5 of the instances.

Figure 4(right) shows the fractions of constraints that can be seeded over all benchmark instances. At least one constraint is seeded for 71.4% of the instances; at least half of all constraints are seeded for 41.6% of the instances; and all of the constraints are seeded for 33.7% of the instances. Note that while the whole instance is solved directly as an IP through a single `Min-Hs` call when all constraints are seeded, we also observed that there are instances on which the runtime of `Min-Hs` dominates even though all constraints are not seeded.

5.7 Results: Comparison with a Commercial IP Solver

Finally, we investigate how the prototype implementation of PBO-IHS fares in terms of runtime performance against CPLEX, one of the de-facto commercial MIP solvers with a significant number of person years behind it. For a fair comparison with CPLEX, we



■ **Figure 5** Per-instance runtime comparison of PBO-IHS (x-axis) vs CPLEX (y-axis).

used the CPLEX presolver also before calling PBO-IHS. This eliminates to an extent the differentiating contribution of the powerful preprocessor of CPLEX in terms of runtime performance (though it should be noted that CPLEX appears to employ further probing for e.g. clique inequalities after the presolving stage, which we were unable to employ before calling PBO-IHS). A per-instance runtime comparison is shown in Figure 5, with more details per benchmark domain provided in Appendix A. We observe that, while CPLEX fairs better in the overall number of solved instances, the two solvers exhibit noticeably complementary performance, relative performance depending on the problem domain considered.

6 Conclusions

We described and implemented a first instantiation of the implicit hitting set approach for pseudo-Boolean optimization. On one hand, the instantiation is motivated by the great success of the implicit hitting set approach in the context of maximum satisfiability, which motivates extending the approach to the more generic context of PBO. On the other hand, the instantiation is motivated by recent advances in pseudo-Boolean solving as a generalization of SAT solving, providing efficient unsatisfiable core extraction which is one of the critical requirements for realizing IHS for PBO. We studied the impact of liftings of various IHS search techniques from MaxSAT to PBO, and showed through an extensive empirical evaluation that our IHS PBO solver implementation provides in practice a competitive as well as complementary approach to pseudo-Boolean optimization.

References

- 1 Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub. Unsatisfiability-based optimization in CLASP. In Agostino Dovier and Vítor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, volume 17 of *LIPICs*, pages 211–221. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPICs.ICLP.2012.211.

- 2 Fahiem Bacchus, Antti Hyttinen, Matti Järvisalo, and Paul Saikko. Reduced cost fixing in MaxSAT. In Christopher Beck, editor, *Principles and Practice of Constraint Programming – 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 – September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 641–651. Springer, 2017. doi:10.1007/978-3-319-66158-2_41.
- 3 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. *Maximum Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 24, pages 929–991. IOS Press BV, 2021. doi:10.3233/FAIA201008.
- 4 Omid Sanei Bajgiran, André A. Ciré, and Louis-Martin Rousseau. A first look at picking dual variables for maximizing reduced cost fixing. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming – 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, volume 10335 of *Lecture Notes in Computer Science*, pages 221–228. Springer, 2017. doi:10.1007/978-3-319-59776-8_18.
- 5 Peter Barth. Linear 0-1 inequalities and extended clauses. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning, 4th International Conference, LPAR’93, St. Petersburg, Russia, July 13-20, 1993, Proceedings*, volume 698 of *Lecture Notes in Computer Science*, pages 40–51. Springer, 1993. doi:10.1007/3-540-56944-8_40.
- 6 Jeremias Berg, Fahiem Bacchus, and Alex Poole. Abstract cores in implicit hitting set MaxSat solving. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020 – 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 277–294. Springer, 2020. doi:10.1007/978-3-030-51825-7_20.
- 7 Jeremias Berg and Matti Järvisalo. Weight-aware core extraction in SAT-based MaxSAT solving. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming – 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 – September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 652–670. Springer, 2017. doi:10.1007/978-3-319-66158-2_42.
- 8 Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–6, 2010. doi:10.3233/sat190075.
- 9 Sam Buss and Jakob Nordström. Proof complexity and SAT solving. In Armin Biere, Marijn Heule, Hans Van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 7, pages 133–182. IOS Press BV, 2021. doi:10.3233/FAIA200990.
- 10 Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*, pages 830–835. ACM, 2003. doi:10.1145/775832.776041.
- 11 Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. A modular approach to MaxSAT modulo theories. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing – SAT 2013 – 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 150–165. Springer, 2013. doi:10.1007/978-3-642-39071-5_12.
- 12 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.
- 13 IBM ILOG Cplex. V12. 1: User’s manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009.
- 14 Harlan P. Crowder, Ellis L. Johnson, and Manfred W. Padberg. Solving large-scale zero-one linear programming problems. *Operational Research*, 31(5):803–834, 1983. doi:10.1287/opre.31.5.803.
- 15 George B. Dantzig, D. Ray Fulkerson, and Selmer M. Johnson. Solution of a large-scale traveling-salesman problem. *Operational Research*, 2(4):393–410, 1954. doi:10.1287/opre.2.4.393.

- 16 Jessica Davies. *Solving MaxSAT by decoupling optimization and satisfaction*. PhD thesis, University of Toronto, 2013.
- 17 Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In Jimmy Ho-Man Lee, editor, *Principles and Practice of Constraint Programming – CP 2011 – 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2011. doi:10.1007/978-3-642-23786-7_19.
- 18 Jessica Davies and Fahiem Bacchus. Postponing optimization to speed up MAXSAT solving. In Christian Schulte, editor, *Principles and Practice of Constraint Programming – 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2013. doi:10.1007/978-3-642-40627-0_21.
- 19 Erin Delisle and Fahiem Bacchus. Solving weighted CSPs by successive relaxations. In Christian Schulte, editor, *Principles and Practice of Constraint Programming – 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 273–281. Springer, 2013. doi:10.1007/978-3-642-40627-0_23.
- 20 Jo Devriendt, Ambros Gleixner, and Jakob Nordström. Learn to relax: Integrating 0-1 integer linear programming with pseudo-Boolean conflict-driven search. *Constraints*, 2021. doi:10.1007/s10601-020-09318-x.
- 21 Jo Devriendt, Stephan Gocht, Emir Demirovic, Jakob Nordström, and Peter J. Stuckey. Cutting to the core of pseudo-boolean optimization: Combining core-guided search with cutting planes reasoning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3750–3758. AAAI Press, 2021. URL: <https://ojs.aaai.org/index.php/AAAI/article/view/16492>.
- 22 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. doi:10.1007/978-3-540-24605-3_37.
- 23 Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003. doi:10.1016/S1571-0661(05)82542-3.
- 24 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-Boolean solving. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1291–1299. ijcai.org, 2018. doi:10.24963/ijcai.2018/180.
- 25 Katalin Fazekas, Fahiem Bacchus, and Armin Biere. Implicit hitting set algorithms for maximum satisfiability modulo theories. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning – 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2018. doi:10.1007/978-3-319-94205-6_10.
- 26 Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp M. Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans D. Mittelmann, Derya Ozyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: Data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 2021. doi:10.1007/s12532-020-00194-3.
- 27 Antti Hyttinen, Paul Saikko, and Matti Järvisalo. A core-guided approach to learning optimal causal graphs. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 645–651. ijcai.org, 2017. doi:10.24963/ijcai.2017/90.

- 28 Alexey Ignatiev, Alessandro Previti, Mark H. Liffiton, and João Marques-Silva. Smallest MUS extraction with minimal hitting set dualization. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming – 21st International Conference, CP 2015, Cork, Ireland, August 31 – September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 173–182. Springer, 2015. doi:10.1007/978-3-319-23219-5_13.
- 29 Saurabh Joshi, Ruben Martins, and Vasco M. Manquinho. Generalized totalizer encoding for pseudo-Boolean constraints. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming – 21st International Conference, CP 2015, Cork, Ireland, August 31 – September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 200–209. Springer, 2015. doi:10.1007/978-3-319-23219-5_15.
- 30 Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014 – 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14–17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer, 2014. doi:10.1007/978-3-319-09284-3_33.
- 31 António Morgado, Carmine Dodaro, and João Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In Barry O’Sullivan, editor, *Principles and Practice of Constraint Programming – 20th International Conference, CP 2014, Lyon, France, September 8–12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer, 2014. doi:10.1007/978-3-319-10428-7_41.
- 32 George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. Wiley Interscience Series in Discrete Mathematics and Optimization. Wiley, 1988. doi:10.1002/9781118627372.
- 33 Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991. doi:10.1137/1033004.
- 34 Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. URL: <https://www.sciencedirect.com/science/bookseries/15746526/2>.
- 35 Olivier Roussel. Pseudo-Boolean competition 2016. <http://www.cril.univ-artois.fr/PB16/>. Accessed: 25 April 2021.
- 36 Olivier Roussel and Vasco Manquinho. Pseudo-boolean and cardinality constraints. In Armin Biere, Marijn Heule, Hans Van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, Frontiers in Artificial Intelligence and Applications, chapter 28, pages 1087–1129. IOS Press BV, 2021. doi:10.3233/FAIA201012.
- 37 Paul Saikko, Jeremias Berg, and Matti Järvisalo. LMHS: A SAT-IP hybrid MaxSAT solver. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016 – 19th International Conference, Bordeaux, France, July 5–8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 539–546. Springer, 2016. doi:10.1007/978-3-319-40970-2_34.
- 38 Paul Saikko, Carmine Dodaro, Mario Alviano, and Matti Järvisalo. A hybrid approach to optimization in answer set programming. In Michael Thielscher, Francesca Toni, and Frank Wolter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October – 2 November 2018*, pages 32–41. AAAI Press, 2018. URL: <https://aaai.org/ocs/index.php/KR/KR18/paper/view/18021>.
- 39 Paul Saikko, Johannes Peter Wallner, and Matti Järvisalo. Implicit hitting set algorithms for reasoning beyond NP. In Chitta Baral, James P. Delgrande, and Frank Wolter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25–29, 2016*, pages 104–113. AAAI Press, 2016. URL: <http://www.aaai.org/ocs/index.php/KR/KR16/paper/view/12812>.

- 40 Masahiko Sakai and Hidetomo Nabeshima. Construction of an ROBDD for a PB-constraint in band form and related techniques for PB-solvers. *IEICE Transactions on Information and Systems*, 98-D(6):1121–1127, 2015. doi:10.1587/transinf.2014F0P0007.
- 41 Roberto Sebastiani and Patrick Trentin. OptiMathSAT: A tool for optimization modulo theories. *Journal of Automated Reasoning*, 64(3):423–460, 2020. doi:10.1007/s10817-018-09508-6.
- 42 Hossein M. Sheini and Karem A. Sakallah. Pueblo: A hybrid pseudo-Boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):165–189, 2006. doi:10.3233/sat190020.

A Detailed Results: PBO-IHS vs CPLEX

Table 2 provides a per-instance comparison of the performance of PBO-IHS and CPLEX on the *full* benchmark set.

■ **Table 2** Per-domain comparison of PBO-IHS and CPLEX: number of solved instances (#) and cumulative runtimes over solved instances in seconds (cum.)

Domain (#instances)	PBO-IHS		CPLEX	
	#	cum.	#	cum.
10orplus/9orless (156)	156	20309	156	1709
caixa (24)	18	38	24	61
rand.*list (118)	118	878	118	301
area_* (59)	57	10735	59	789
trarea_ac (18)	17	5209	18	47
aries-da_nrp (70)	55	17508	70	2278
BA (1440)	7	17028	761	419659
NG (960)	0	0	238	224058
MANETs (150)	27	13051	61	25757
BioRepair (30)	30	223	30	862
Metro (30)	27	10911	30	2626
ShiftDesign (30)	10	9688	6	7062
Timetabling (30)	28	8019	27	6313
EmployeeScheduling (14)	0	0	13	149
golomb-rulers (34)	12	4669	10	589
bsg (60)	5	16	15	3571
mis/mds (120)	64	26665	58	15127
course-ass (6)	1	8	6	12
decomp (10)	0	0	0	0
data (68)	10	2202	24	3076
dt-problems (60)	47	74	60	358
domset (15)	0	0	0	0
factor (186)	186	348	186	242
factor-mod-B (225)	225	317	216	4204
fctsp (35)	12	622	12	936
featureSubscription (20)	20	301	1	2644
frbXX-XX-opb (40)	5	5397	3	3615
flexray (9)	4	69	3	14
fome (3)	0	0	0	0
graca (100)	62	20019	27	14459

Domain (#instances)	PBO-IHS		CPLEX	
	#	cum.	#	cum.
haplotype (8)	7	2992	0	0
garden (7)	6	76	6	60
hw32/hw64/hw128 (27)	6	1324	18	5072
jXXopt (2040)	1581	47081	1487	136243
keeloq_tasca (4)	4	124	4	1412
kullmann (7)	3	3016	3	3183
lion9-single-obj (1513)	1487	33403	1480	57923
logic-synthesis (74)	71	767	71	690
miplib/neos (79)	36	9962	58	14578
miplib/other (405)	161	32009	217	50306
unibo (36)	8	4764	8	6826
market-split (20)	0	0	8	6075
opb/graphpart (31)	24	3795	28	715
opb/autocorr_bern (43)	8	1838	8	2180
opb/sporttournament (22)	11	3056	13	3089
opb/edgexcross (19)	12	3433	15	6316
opb/pb (8)	0	0	0	0
opb/faclay (10)	1	879	1	1004
opb/other (6)	1	2	3	4106
primes/aim (48)	44	236	46	235
primes/jnh (16)	16	52	16	42
primes/ii (41)	34	5148	34	5060
primes/par (30)	20	369	20	426
primes/other (13)	5	1512	5	976
routing (15)	15	32	15	28
radar (12)	11	62	12	39
synthesis-ptl-cmos (10)	10	17	10	18
testset (6)	6	8	6	12
ttp (8)	2	12	2	4
vtxcov (15)	0	0	0	0
wnq (15)	0	0	0	0

An Algorithm-Independent Measure of Progress for Linear Constraint Propagation

Boro Sofranac ✉

Zuse Institute Berlin, Germany
TU Berlin, Germany

Ambros Gleixner ✉

Zuse Institute Berlin, Germany
HTW Berlin, Germany

Sebastian Pokutta ✉

Zuse Institute Berlin, Germany
TU Berlin, Germany

Abstract

Propagation of linear constraints has become a crucial sub-routine in modern Mixed-Integer Programming (MIP) solvers. In practice, iterative algorithms with tolerance-based stopping criteria are used to avoid problems with slow or infinite convergence. However, these heuristic stopping criteria can pose difficulties for fairly comparing the efficiency of different implementations of iterative propagation algorithms in a real-world setting. Most significantly, the presence of unbounded variable domains in the problem formulation makes it difficult to quantify the relative size of reductions performed on them. In this work, we develop a method to measure – independently of the algorithmic design – the progress that a given iterative propagation procedure has made at a given point in time during its execution. Our measure makes it possible to study and better compare the behavior of bounds propagation algorithms for linear constraints. We apply the new measure to answer two questions of practical relevance: (i) We investigate to what extent heuristic stopping criteria can lead to premature termination on real-world MIP instances. (ii) We compare a GPU-parallel propagation algorithm against a sequential state-of-the-art implementation and show that the parallel version is even more competitive in a real-world setting than originally reported.

2012 ACM Subject Classification Mathematics of computing → Mixed discrete-continuous optimization

Keywords and phrases Bounds Propagation, Mixed Integer Programming

Digital Object Identifier 10.4230/LIPIcs.CP.2021.52

Related Version Full Version: <https://arxiv.org/abs/2106.07573>

Supplementary Material Software (Source Code):

<https://github.com/Sofranac-Boro/gpu-domain-propagator>

Funding This research was partially conducted within the Research Campus MODAL funded by the German Federal Ministry of Education and Research (BMBF grant numbers 05M14ZAM, 05M20ZBM) as well as the DFG Cluster of Excellence MATH+ (EXC-2046/1, project id 390685689) funded by the Deutsche Forschungsgemeinschaft (DFG).

1 Introduction

This paper is concerned with *Mixed-Integer Linear Programs* (MIPs) of the form

$$\min\{c^T x \mid Ax \leq b, \ell \leq x \leq u, x \in \mathbb{R}^n, x_j \in \mathbb{Z} \text{ for all } j \in I\}, \quad (1)$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, and $I \subseteq \mathbb{N} = \{1, \dots, n\}$. Additionally, $\ell \in \mathbb{R}_{-\infty}^n$ and $u \in \mathbb{R}_{\infty}^n$, where $\mathbb{R}_{\infty} := \mathbb{R} \cup \{\infty\}$ and $\mathbb{R}_{-\infty} := \mathbb{R} \cup \{-\infty\}$. For each variable x_j , the interval $[\ell_j, u_j]$ is called its *domain*, which is defined by its lower and upper *bounds* ℓ_j and u_j , which may be infinite.



© Boro Sofranac, Ambros Gleixner, and Sebastian Pokutta;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 52; pp. 52:1–52:17



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Surprisingly fast solvers for solving MIPs have been developed in practice despite MIPs being \mathcal{NP} -hard in the worst case [4, 15]. To this end, the most successful method has been the *branch-and-bound* algorithm [16] and its numerous extensions. The key idea of this method is to split the original problem into several sub-problems (*branching*) which are hopefully easier to solve. By doing this recursively, a *search tree* is created with nodes being the individual sub-problems. The *bounding* step solves relaxations of sub-problems to obtain a lower bound on their solutions. This bound can then be used to prune sub-optimal nodes which cannot lead to improving solutions. By doing this, the algorithm tries to avoid having to enumerate exponentially many sub-problems. The most common way to obtain a relaxation of a sub-problem is to drop the integrality constraints of the variables. This yields a *Linear Program* (LP) which can be solved e.g., by the simplex method [21].

This core idea is extended by numerous techniques to speed up the solution process. One of the most important techniques is called *constraint propagation*. It improves the formulation of the (sub)problem by removing parts of domains of each variable that it detects cannot lead to *feasible* solutions [23]. The more descriptive term *bounds propagation* or *bounds tightening* is used to denote the variants that maintain a continuous interval as domain. Modern MIP solvers make use of this technique during *presolving* in order to improve the global problem formulation [24], as well as during the branch-and-bound algorithm to improve the formulation of the sub-problems at the nodes of the search tree [1].

In practice, efficient implementations exist in MIP solvers [3, 1] and recently even a GPU-parallel algorithm [26] has been developed. These are iterative methods, which may converge to the tightest bounds only at infinity. For such methods, the presence of unbounded variable domains in the problem formulation makes the quantification of the relative distance to the final result at a given iteration difficult. (Iterative bounds tightening has a unique fixed point to which it converges, see Section 2.2.) In turn, this makes it difficult to define an implementation-independent measure of how much progress these algorithms have achieved at a given iteration.

In this paper, we address this difficulty and introduce tools to study and compare the behavior of iterative bounds tightening algorithms in MIP. We show that the reduction of infinite bounds to some finite values is a fundamentally different process from the subsequent (finite) improvements thereafter, and thus propose to measure the ability of an algorithm to make progress in each of the processes independently. We show how the challenge posed by infinite starting bounds can be solved and provide methods for measuring the progress of both the infinite and the finite domain reductions. Pseudocode and hints are provided to aid independent implementation of our procedure. Additionally, the code of our own implementation is made publicly available.

On the applications side, the new procedure is used to investigate two questions. First, we analyze to what extent heuristic, tolerance-based stopping criteria as typically imposed by real-world MIP solvers can cause iterative bounds tightening algorithms to terminate prematurely; we find that this situation occurs rarely in practice. Second, we compare a newly developed, GPU-based propagation algorithm [26] to a state-of-the-art sequential implementation in a real-world setting where both are terminated early; we show that the GPU-parallel version is even more competitive than originally reported.

The rest of the paper is organized as follows. After presenting the necessary background and motivation in Section 2, we discuss the properties of bounds propagation and its ability to perform reductions on infinite and on finite bounds in Section 3. Based on the findings, we present functions used to measure the progress of bounds tightening algorithms in Section 4. Lastly, in Section 5, we apply the developed procedure to answer the above-mentioned questions and present our computational results. Section 6 gives a brief outlook.

2 Background and Motivation

In Section 2.1, we introduce some basic terminology used in the Constraint Programming (CP) and MIP communities, related to constraint propagation. Section 2.2 formally presents bounds propagation of linear constraints alongside some known results from literature that are relevant for the discussions in the paper. In Section 2.3 we outline the problems that motivate the paper.

2.1 Constraint Propagation in CP and MIP

In the Constraint Programming (CP) community, constraint propagation appears in a variety of forms, both in terms of the algorithms and its desired goals [23]. The propagation algorithms are implemented via mappings called *propagators*. A propagator is a monotonically decreasing function from variable domains to variable domains [25]. The goal of most propagation algorithms is formalized through the notion of *consistency*, which these algorithms strive to achieve. The most successful consistency technique is *arc consistency* [18]. Multivariate extension of arc consistency has been called *generalized arc consistency* [20], as well as *domain consistency* [27], and *hyper-arc consistency* [19]. Informally speaking, a given domain is *domain consistent* for a given constraint if it is the least domain containing all solutions to the constraint (see [23] for a formal definition).

The main idea of *bounds consistency* is to relax the consistency requirement to only require the lower and the upper bounds of domains of each variable to fulfill it. There are several bounds consistency notions in the CP literature [10]. In this paper, we adopt the notion of bounds consistency from [1, Definition 2.7].

Modern CP solvers often work with a number of propagators which might or might not strive for different levels of consistency [25]. In this setting, the notions such as *greatest common fixed point* (see [9, Definition 4]) and consistency of a system of constraints are often analysed as a product of a set of propagators. Solvers often focus on optimizing the interplay between different propagators (e.g., see [25]) to quickly decide feasibility.

In MIP solving, constraint propagation additionally interacts with many other components that are mostly focused on reaching and proving optimality, see [2, 5, 6, 8] for examples of different approaches to integrate constraint propagation and MIP. As a result, the role of constraint propagation in the larger solving process changes and developers are faced with different computational trade-offs. In practice, propagation is almost always terminated before the fixed point is reached [1]. In this paper, we are concerned with constraint propagation of a set of linear constraints, where we explicitly include the presence of continuous variables and of variables with initially unbounded domains, which frequently occur in real-world MIP formulations.

2.2 Bounds Propagation of Linear Constraints

A *linear constraint* can be written in the form

$$\underline{\beta} \leq \sum_{i=1}^n a_i x_i \leq \bar{\beta}, \quad (2)$$

where $\underline{\beta} \in \mathbb{R}_{-\infty}$ and $\bar{\beta} \in \mathbb{R}_{\infty}$ are left and right hand sides, respectively, and $a \in \mathbb{R}^n$ is the vector of constraint coefficients. Variables x_i have lower and upper bounds $\ell_i \in \mathbb{R}_{-\infty}$ and $u_i \in \mathbb{R}_{\infty}$, respectively.¹ We require the following definitions:

¹ When $x \in \mathbb{Z}$, then $\ell \in \mathbb{Z}_{-\infty}$ and $u \in \mathbb{Z}_{\infty}$, however, because $\mathbb{Z} \subset \mathbb{R}$, integer variables can be handled the same way as real ones. In the remainder of the paper, \mathbb{Z} will be used only where necessary.

► **Definition 1** (activity bounds and residuals). *Given a constraint of the form (2) and bounds $\ell \leq x \leq u$, the functions $\underline{\alpha} : \mathbb{R}_{-\infty}^n, \mathbb{R}_{\infty}^n \mapsto \mathbb{R} \cup \{-\infty, \infty\}$ and $\bar{\alpha} : \mathbb{R}_{-\infty}^n, \mathbb{R}_{\infty}^n \mapsto \mathbb{R} \cup \{-\infty, \infty\}$ are called the minimum and maximum activities of the constraint, respectively, and are defined as*

$$\underline{\alpha} = \underline{\alpha}(\ell, u) = \sum_{i=1}^n a_i b_i \text{ with } b_i = \begin{cases} \ell_i & \text{if } a_i > 0 \\ u_i & \text{if } a_i < 0 \end{cases}, \quad (3a)$$

and

$$\bar{\alpha} = \bar{\alpha}(\ell, u) = \sum_{i=1}^n a_i b_i \text{ with } b_i = \begin{cases} u_i & \text{if } a_i > 0 \\ \ell_i & \text{if } a_i < 0 \end{cases}. \quad (3b)$$

The functions $\underline{\alpha}_j : \mathbb{R}_{-\infty}^n, \mathbb{R}_{\infty}^n, \{1, \dots, n\} \mapsto \mathbb{R} \cup \{-\infty, \infty\}$ and $\bar{\alpha}_j : \mathbb{R}_{-\infty}^n, \mathbb{R}_{\infty}^n, \{1, \dots, n\} \mapsto \mathbb{R} \cup \{-\infty, \infty\}$ are called the j -th minimum activity residual and the j -th maximum activity residual of the constraint, and are defined as

$$\underline{\alpha}_j = \underline{\alpha}_j(\ell, u, j) = \sum_{i=1, i \neq j}^n a_i b_i \text{ with } b_i = \begin{cases} \ell_i & \text{if } a_i > 0 \\ u_i & \text{if } a_i < 0 \end{cases}, \quad (4a)$$

and

$$\bar{\alpha}_j = \bar{\alpha}_j(\ell, u, j) = \sum_{i=1, i \neq j}^n a_i b_i \text{ with } b_i = \begin{cases} u_i & \text{if } a_i > 0 \\ \ell_i & \text{if } a_i < 0 \end{cases}. \quad (4b)$$

► **Definition 2** (bound candidate functions). *The functions $\mathcal{B}_{surplus}^j : \mathbb{R}_{-\infty}^n, \mathbb{R}_{\infty}^n \mapsto \mathbb{R} \cup \{-\infty, \infty\}$ and $\mathcal{B}_{slack}^j : \mathbb{R}_{-\infty}^n, \mathbb{R}_{\infty}^n \mapsto \mathbb{R} \cup \{-\infty, \infty\}$ are called the bound candidate functions and are defined as*

$$\mathcal{B}_{surplus}^j(\ell, u) = \frac{\bar{\beta} - \underline{\alpha}_j}{a_j}, \quad (5a)$$

and

$$\mathcal{B}_{slack}^j(\ell, u) = \frac{\underline{\beta} - \bar{\alpha}_j}{a_j}. \quad (5b)$$

Then, the following observations are true and can be translated into algorithmic steps, see, e.g., [1, 14, 9]:

► **Observation 3** (linear constraint propagation).

1. If $\underline{\beta} \leq \underline{\alpha}$ and $\bar{\alpha} \leq \bar{\beta}$, then the constraint is redundant and can be removed.
2. If $\underline{\alpha} > \bar{\beta}$ or $\underline{\beta} > \bar{\alpha}$, then the constraint cannot be satisfied and hence the entire (sub)problem is infeasible.
3. Let x satisfy (2), i.e., $\underline{\beta} \leq \sum_{i=1}^n a_i x_i \leq \bar{\beta}$, then for all $j = \{1, \dots, n\}$ with $a_j > 0$,

$$\ell^{new} = \mathcal{B}_{slack}^j(\ell, u) \leq x_j \leq \mathcal{B}_{surplus}^j(\ell, u) = u^{new}, \quad (6a)$$

and for all $j = \{1, \dots, n\}$ with $a_j < 0$,

$$\ell^{new} = \mathcal{B}_{surplus}^j(\ell, u) \leq x_j \leq \mathcal{B}_{slack}^j(\ell, u) = u^{new}. \quad (6b)$$

4. For all $j \in \{1, \dots, n\}$ such that $x_j \in \mathbb{Z}$,

$$\lceil \ell_j^{\text{new}} \rceil \leq x_j \leq \lfloor u_j^{\text{new}} \rfloor \quad (7)$$

If the first two steps are not applicable, the algorithm computes the new bounds ℓ^{new} and u^{new} in Steps 3 and 4. For a given variable j , if $\ell_j^{\text{new}} > \ell_j$, then the bound is updated with the new value. Similarly, u_j is updated if $u_j^{\text{new}} < u_j$.

An actual implementation may skip Steps 1 and 2 without changing the result. This is because for redundant constraints Steps 3 and 4 correctly detect no bound tightenings, and for infeasible constraints, Steps 3 and 4 lead to at least one variable with an empty domain, i.e., $\ell_j^{\text{new}} > u_j^{\text{new}}$.

When propagating a system of the type (1) which consists of several constraints, one simply applies the above steps to each constraint independently. Notice that in such systems, it is possible for two or more constraints to share the same variables (i.e., coefficients a_j are non-zero in several constraints). Therefore, if a bound of a variable is changed in one constraint, this can trigger further bound changes in the constraints which also have this variable. This gives the propagation algorithm its iterative nature, as one has to repeat the propagation process over the constraints as long as at least one bound change is found. A pass over all the constraints is also called a *propagation round*. If no bound changes are found during a given round, then no further progress is possible and the algorithm terminates. At this point, all constraints are guaranteed to be bound consistent [1].

This algorithm can be interpreted as a fixed-point iteration in the space of variable and activity bounds with a unique fixed point [9]; it converges to this fixed point, however not necessarily in finite time [7]. Additionally, even when it does converge to the fixed point in finite time, convergence can be very slow in practice [1, 9, 17]. To deal with this, practical implementations of bounds propagation introduce tolerance-based termination criteria which stop the algorithm if the progress becomes too slow, i.e., the relative size of improvements on the bounds falls below a specified threshold. With this modification, the algorithm always terminates in finite time (but not in worst-case polynomial-time), however, it may fail to compute the best bounds possible.

To distinguish the above-described approach from alternative methods to compute consistent bounds (see, e.g., [7] for a method solving a single LP instead), we will use the following definition:

► **Definition 4** (Iterative Bounds Tightening Algorithm). *Given variable bounds ℓ , u of a problem of the form (1), any algorithm updating these bounds by calculating ℓ^{new} , u^{new} via (6a), (6b), and (7) iteratively as described in Observation 3, thus traversing a sequence of bounds $(\ell, u)_1, (\ell, u)_2, \dots$ is called an iterative bounds tightening algorithm (IBTA).*

Note that this definition leaves the flexibility for individual algorithmic choices, for example, the timing of when bound changes are applied or the order in which the constraints are processed. If a given algorithm applies the found changes immediately, making them available to subsequent constraints in the same iteration, it might traverse a shorter sequence of bounds to the fixed point than the algorithm which delays updates of bounds until the end of the current iteration (e.g., because it processes constraints in parallel). The ordering of processed constraints can lead to different traversed sequences because a given bound change that depends on other changes being applied first might be missed in a given iteration if the constraint it depends on is not processed first.

2.3 Motivation

Our motivation for this paper is threefold:

1. Estimating the premature stalling effect of IBTAs: In the context of MINLP, Belotti et al. [7] propose an alternative bounds propagation algorithm that computes the bounds at the fixed point directly by solving a single linear program. This approach circumvents non-finite convergence behavior and shows that the bounds at the fixed point can in theory be computed in polynomial time.

Nevertheless, in practice, the trade-off between the quality of obtained bounds including their effect on the wider branch-and-bound algorithm and the algorithm's runtime makes the iterative bounds propagation with tolerance-based stopping criteria the most effective method in most cases, despite its exponential worst-case runtime. The use of stopping criteria still leaves individual instances or potentially even instance classes susceptible to the following effect stated by Belotti et al. as a motivation for their LP-based approach, which is also the motivation for our paper: *“However, because the improvements are not guaranteed to be monotonically non-increasing, terminating the procedure after one or perhaps several small improvements might in principle overlook the possibility of a larger improvement later on.”* In their paper, no attempt is made to quantify this statement, as likely out-of-scope and non-trivial to answer.

In this work, we aim to develop a methodology to quantify the overall progress that a given IBTA achieved up to a given point in its execution. Ideally, we would like to have a function f , which maps current variable bounds to a scalar value, for example in $[0, 100]$, which measures the achieved progress. The main difficulty in developing such a function comes in the form of unbounded variable domains in the input instances (and potentially during the algorithm's execution). Observing the values of such a function over the execution time of the algorithm could then be used to study the behavior of IBTAs on instances of interest and quantify the effect brought up by Belotti et al., which we call *premature stalling* (see Section 5.2 for formal definition). Furthermore, an algorithm-independent f would allow comparing the behavior of different IBTAs with respect to premature stalling.

2. Performance comparison of different IBTAs in practice: As already motivated by Definition 4, different IBTAs might traverse different sequences of bounds from the initial values to the fixed point. Additionally, we stated in Section 2.2 that in practice, iterative bounds propagation is used exclusively with tolerance-based stopping criteria, meaning that the algorithm is stopped potentially before reaching the fixed point. The following problem then arises: for two such algorithms traversing different sequences of bounds that are stopped before reaching the unique fixed point, how do we judge which one performed better? Perhaps a more natural way to formulate this question is: *in how much time do the two algorithms achieve the same amount of progress?* A function measuring the progress of iterative bounds propagation as already proposed can be used to answer this question.

As a concrete example, we will compare the following two IBTAs: the canonical, state-of-the-art sequential implementation, for example from [1], and a GPU-parallel algorithm recently proposed in [26]. In the preliminary computational study on the MIPLIB 2017 test set [13] presented in [26], the two algorithms are compared for the propagation to the fixed point (no tolerance-based stopping criteria). In this work, we will compare the performance of the two algorithms in a real-world setting, i.e., when terminated before reaching the fixed point.

3. Designing stopping criteria: as already stated, the tolerance-based stopping criteria are crucial for effective IBTAs. Notice that because different IBTAs might traverse different sequences of bounds, their average individual improvements on the bounds might be different in size. In fact, the study in [26] shows that on average, the size of improvements by the GPU-parallel algorithm is smaller than that of the canonical sequential implementation over the MIPLIB 2017 test set, despite its higher performance in terms of runtime to the fixed point. An important implication of this effect is that given two such algorithms, a given stopping criterion might be effective for one of them, but ineffective for the other. In this context, quantifying the magnitude and distribution of expected improvements of a given algorithm for a given problem class and its likelihood to prematurely stall, would allow one to make more informed decisions when designing *effective* stopping criteria.

Lastly, we believe that gaining insight into the behavior of these algorithms is a motivation in itself that could potentially benefit future and existing methodologies in the context of linear constraint propagation.

3 Finite and infinite domain reductions

Any IBTA starts with arrays of initial lower and upper bounds, $\ell^s \in \mathbb{R}_{-\infty}^n$ and $u^s \in \mathbb{R}_{\infty}^n$, respectively, and incrementally updates individual bounds towards the uniquely defined fixed-point bounds which we denote by ℓ^l and u^l . To denote the arrays of bounds at any given time between the start and the fixed point we simply use ℓ and u and call them *current bounds*. Obviously, it holds that $\ell_j^s \leq \ell_j \leq \ell_j^l$ and $u_j^s \geq u_j \geq u_j^l$ for all $j \in \{1, \dots, n\}$. Observe that both initial and limit bounds may contain infinite values.

3.1 Reducing Infinite Bounds to Finite Values

Variables that start with infinite value in either lower or upper bound, will either remain infinite if no bound change is possible or will become finite values. We start with the following simple observation:

► **Observation 5.** *Given a constraint of the form (2) and a given variable $j \in \{1, \dots, n\}$ with a bound $\ell_j = -\infty$ (or $u_j = \infty$), the possibility of tightening this bound to some finite value depends on the signs of coefficients $a_j, j \in \{1, \dots, n\}$, the finiteness of variable bounds $\ell_j, j \in \{1, \dots, n\} \setminus j$ and $u_j, j \in \{1, \dots, n\} \setminus j$, and the finiteness of $\underline{\beta}$ and $\bar{\beta}$, but not on the values that these variables take, if they are finite.*

Proof. To see the dependence on the sign of coefficients a , let the lower bound of a given variable j be $\ell_j = -\infty$ and let $\underline{\beta}$ and $\bar{\alpha}_j$ be finite, $\bar{\beta} = \infty$ and $\underline{\alpha}_j = -\infty$. Then, by (6a) and (6b), $a_j > 0$ implies $\ell_j^{\text{new}} \in \mathbb{R} > -\infty$ and the bound is updated. Else, if $a_j < 0$, then $\ell_j^{\text{new}} = -\infty$ and no bound change is possible.

The dependence on the finiteness of $\underline{\beta}$ and $\bar{\beta}$ is trivial, while the coefficients a are finite by problem definition. To see the dependence on the finiteness of variable bounds, consider the activities $\underline{\alpha}_j$ and $\bar{\alpha}_j$ of a variable j with $\ell_j = -\infty$, $u_j = \infty$, and $a_i > 0$ for all $i \in \{1, \dots, n\}$. If there exists k such that $u_k = \infty, k \in \{1, \dots, n\} \setminus j$ then $\bar{\alpha} = \infty$ and consequently ℓ_j cannot be tightened. Otherwise, if $u_k \in \mathbb{R}$ for all $k \in \{1, \dots, n\} \setminus j$, then $\bar{\alpha} \in \mathbb{R}$ and a bound tightening is possible.

The specific finite values that the variables in (6a) and (6b) take have no effect on the possibility to reduce an infinite bound to a finite value because arithmetic operations between finite values again produce a finite value (also $a_j \neq 0$ by definition) and $-\infty < k < \infty$ for all $k \in \mathbb{R}$. Variables which are restricted to integer values also do not affect this process, as the operations $\lceil \ell_j \rceil$ and $\lfloor u_j \rfloor$ give $\ell_j, u_j \in \mathbb{Z}$ and $\mathbb{Z} \subset \mathbb{R}$. The same argument from above then applies. ◀

Notice that the same effect of finite bound changes triggering new bound changes in the subsequent propagation rounds is also true for infinite domain reductions, hence this process might also require more than one iteration. Furthermore, these iterations have the following property:

► **Corollary 6.** *Let $k \in \mathbb{N}$ be the number of iterations a given IBTA takes to reach the fixed point. Then there is a number $c \leq k$, $c \in \mathbb{N}_0$ such that the first c propagation rounds have at least 1 reduction of an infinite to a finite bound, and none thereafter. By pigeonhole principle, c is at most the number of initially infinite bounds.*

Proof. The coefficients a and the left- and right-hand sides $\underline{\beta}$ and $\bar{\beta}$ are constants that do not change during the course of the algorithm. By Observation 5 the only thing left influencing the infinity reductions is the finiteness of variable bounds. If no infinite to finite reductions are made at any given round, then none can be made thereafter. Finite to infinite reductions are not possible as the algorithm only accepts improving bounds. ◀

In conclusion, the process of reducing infinite bounds to some finite values is independent and fundamentally different from the incremental improvements of finite values thereafter, which is driven by the values of the variables in (6b) and (6a). Accordingly, we will measure the ability of an algorithm to reduce the infinite bounds to some finite values separately from its ability to make improvements on the finite values of the bounds thereafter.

3.2 Finite Domain Reductions

Our main approach in measuring the progress of finite domain reductions (see Section 4.2) relies on the observation that the starting as well as the fixed point of propagation is uniquely defined for a given MIP problem and hence independent from the algorithm used. The measuring function then answers the following question: for given bounds ℓ and u at some time during the propagation process, how far have we gotten from the starting point ℓ^s and u^s , relative to the endpoint ℓ^l and u^l . When the bounds of a given variable did not change during the propagation process, or they are finite at both the start and the end, there is no difficulty in calculating such a measure. However, when a given variable bound started as an infinite value but was tightened to some finite value by the end of propagation, special care is needed to handle this case, which we address in this section.

In Section 2, we discussed how a sequential and a parallel propagation algorithm might traverse different sequences of bounds during their executions. Let us consider the first round of two such algorithms, and see what might happen to the bounds which start as infinite but are tightened during the course of the algorithm. When the sequential algorithm finds a bound change, it is immediately made available to the subsequent constraints in the same round. Consequently, if an infinite domain reduction happens in the subsequent constraints, it may produce a stronger finite value compared to the parallel algorithm which used the older (weaker) bound information. This serves to show that the first finite values that such bounds take may not be the same in different IBTAs. Hence they cannot be used safely to compare finite domain reductions across different implementations. In what follows, we construct a procedure to compute algorithm-independent reference values for each bound.

► **Definition 7** (weakest variable bounds). *Given an optimization problem of the form (1) with starting variable bounds ℓ^s and u^s , we call $\bar{\ell}_j$ weakest lower bound of variable j if*

- $\bar{\ell}_j = -\infty$ and no IBTA can produce a finite lower bound $\ell_j \in \mathbb{R}$, or
- $\bar{\ell}_j \in \mathbb{R}$ and no IBTA can produce a finite lower bound $\ell_j \in \mathbb{R}$ with $\ell_j < \bar{\ell}_j$.

We call \bar{u}_j weakest upper bound of variable j if

- $\bar{u}_j = \infty$ and no IBTA can produce a finite upper bound $u_j \in \mathbb{R}$, or
- $\bar{u}_j \in \mathbb{R}$ and no IBTA can produce a finite upper bound $u_j \in \mathbb{R}$ with $u_j > \bar{u}_j$.

When both the starting and the limit bounds are finite we have $\bar{\ell}_j = \ell_j^s$ resp. $\bar{u}_j = u_j^s$ (because all IBTAs only accept improving bounds) and when they are both infinite we have $\ell_j^s = \ell_j^l = \bar{\ell}_j = -\infty$ resp. $u_j^s = u_j^l = \bar{u}_j = \infty$. Notice that the cases of $\ell_j^s \in \mathbb{R}$ with $\ell_j^l = -\infty$ and $u_j^s \in \mathbb{R}$ with $u_j^l = \infty$ are not possible as $\ell_j^s \leq \ell_j^l$ and $u_j^s \geq u_j^l$. The main challenge in computing $\bar{\ell}$ and \bar{u} is due to the remaining case of $\ell_j^s = -\infty, \ell_j^l \in \mathbb{R}$ resp. $u_j^s = \infty, u_j^l \in \mathbb{R}$. In what follows, we will extend the notation introduced in Section 2.2 with $\mathcal{B}_{\text{slack}}^{ij}$ and $\mathcal{B}_{\text{surplus}}^{ij}$ denoting $\mathcal{B}_{\text{slack}}^j$ and $\mathcal{B}_{\text{surplus}}^j$ applied to constraint i and variable j , respectively. The procedure presented in Algorithm 1 computes $\bar{\ell}$ and \bar{u} .

■ **Algorithm 1** The Weakest Bounds Algorithm.

Input: System of m linear constraints $\beta \leq \sum_{i=1}^n a_i x_i \leq \bar{\beta}$, $\ell \leq x \leq u$

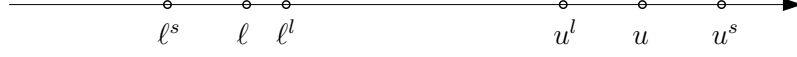
Output: Weakest variable bounds $\bar{\ell}$ and \bar{u}

```

1: mark all constraints
2: bound_change_found  $\leftarrow$  true
3:  $\bar{\ell} = \ell, \bar{u} = u$ 
4: while bound_change_found do
5:   bound_change_found  $\leftarrow$  false
6:   for each constraint  $i$  do
7:     if  $i$  marked then
8:       unmark  $i$ 
9:       for each variable  $j$  such that  $a_{ij} \neq 0$  do
10:        if  $a_{ij} > 0$  then
11:           $\ell_j^{\text{new}} = \mathcal{B}_{\text{slack}}^{ij}(\bar{\ell}, \bar{u})$ 
12:           $u_j^{\text{new}} = \mathcal{B}_{\text{surplus}}^{ij}(\bar{\ell}, \bar{u})$ 
13:        else
14:           $\ell_j^{\text{new}} = \mathcal{B}_{\text{surplus}}^{ij}(\bar{\ell}, \bar{u})$ 
15:           $u_j^{\text{new}} = \mathcal{B}_{\text{slack}}^{ij}(\bar{\ell}, \bar{u})$ 
16:        if  $x_j \in \mathbb{Z}$  then
17:           $\ell_j^{\text{new}} = \lceil \ell_j^{\text{new}} \rceil, u_j^{\text{new}} = \lfloor u_j^{\text{new}} \rfloor$ 
18:        if  $\ell_j = -\infty$  and  $\ell_j^{\text{new}} \in \mathbb{R}$  and ( $\bar{\ell}_j = -\infty$  or ( $\bar{\ell}_j \in \mathbb{R}$  and  $\ell_j^{\text{new}} < \bar{\ell}_j$ )) then
19:           $\bar{\ell}_j \leftarrow \ell_j^{\text{new}}$ 
20:          bound_change_found  $\leftarrow$  true
21:        if  $u_j = \infty$  and  $u_j^{\text{new}} \in \mathbb{R}$  and ( $\bar{u}_j = \infty$  or ( $\bar{u}_j \in \mathbb{R}$  and  $u_j^{\text{new}} > \bar{u}_j$ )) then
22:           $\bar{u}_j \leftarrow u_j^{\text{new}}$ 
23:          bound_change_found  $\leftarrow$  true
24:        if bound_change_found then
25:          mark all constraints  $k$  such that  $a_{kj} \neq 0$ 
26: return  $\bar{\ell}, \bar{u}$ 

```

The procedure starts by setting $\bar{\ell} = \ell^s$ and $\bar{u} = u^s$ and will proceed to iteratively update these bounds until they are all weakest bounds. Up to Lines 18 and 21, the procedure is very similar to the usual bounds propagation: it evaluates (6a), (6b), and (7) on the latest available bounds for all constraints and variables. As the bounds which start as finite values are already weakest by definition, the first part of the checks in Lines 18 and 21 makes



■ **Figure 1** Schematic representation of the starting (index s), current (no index) and limit bounds (index l) for a given variable on the real line. In this example, $\ell^s = \bar{\ell} \in \mathbb{R}$ and $u^s = \bar{u} \in \mathbb{R}$.

sure that these variables are not considered. For bounds that are infinite at the start, the algorithm checks if the new candidate is finite. If so, the new candidate becomes the weakest bound incumbent if the current weakest bound is infinite, or the new candidate is weaker than the current one. This process then repeats in iterations until no further weakenings are possible. Notice that the *constraint marking mechanism*, implemented in Lines 1, 7, 8, 24, and 25 is not necessary for the correctness of the weakest bounds procedure, but as it can substantially speed up the execution of the algorithm, we include it in the pseudocode.

4 An Algorithm-Independent Measure of Progress

As pointed out in Section 3.1, we will measure the ability of an IBTA to reduce infinite bounds to some finite values separately from the improvements of finite bounds. Section 4.1 presents the functions measuring infinite domain reductions, while Section 4.2 presents the functions measuring the progress in finite domain reductions.

As before, we denote the starting bounds of a variable j as ℓ_j^s and u_j^s , the weakest bounds as $\bar{\ell}_j$ and \bar{u}_j , the limit bounds as ℓ_j^l and u_j^l , and the bounds at a given point in time during the propagation as ℓ_j and u_j . Recall that the following relations hold: $\ell_j^s \leq \ell_j \leq \ell_j^l$ and $u_j^s \geq u_j \geq u_j^l$ for all $j \in \{1, \dots, n\}$. Additionally, if $\ell_j \in \mathbb{R}$, then $\ell_j^l \in \mathbb{R}$ and $\ell_j^s \leq \ell_j \leq \ell_j^l$. Likewise, if $u_j \in \mathbb{R}$ then $u_j^l \in \mathbb{R}$ and $u_j^s \geq \bar{u}_j \geq u_j \geq u_j^l$. Lastly, if $\ell_j^s \in \mathbb{R}$ then $\bar{\ell}_j = \ell_j^s$ and if $u_j^s \in \mathbb{R}$ then $\bar{u}_j = u_j^s$. Figure 1 illustrates example starting, current, and limit bounds of a given variable on the real line.

4.1 Measuring Progress in Infinite Domain Reductions

As bounds propagation has a unique fixed point to which it converges, we know the state of the algorithm at both the beginning and the end (a given bound is either finite or infinite). Denote by $n^{\text{total}} \in \mathbb{N}$ the total number of bounds that change from an infinite to some finite value between the starting and the limit bounds of the problem, and by $n^{\text{current}} \in \mathbb{N} \leq n^{\text{total}}$ the number of infinite bounds reduced to finite values by a given IBTA at a given point during its execution:

$$n^{\text{total}} = |\{j = 1, \dots, n : \ell_j^s = -\infty, \ell_j^l \in \mathbb{R}\}| + |\{j = 1, \dots, n : u_j^s = \infty, u_j^l \in \mathbb{R}\}|, \quad (8a)$$

and,

$$n^{\text{current}} = |\{j = 1, \dots, n : \ell_j^s = -\infty, \ell_j \in \mathbb{R}\}| + |\{j = 1, \dots, n : u_j^s = \infty, u_j \in \mathbb{R}\}|. \quad (8b)$$

Then, the progress in infinite domain reductions of the IBTA at that point is calculated as:

$$\mathcal{P}^{\text{inf}} = \frac{n^{\text{current}}}{n^{\text{total}}}, n^{\text{total}} \neq 0. \quad (9)$$

Observe that the total number of infinite domain reductions n^{total} is algorithm-independent and can be precomputed from the starting and the limit bounds. Because IBTAs never relax bounds, \mathcal{P}^{inf} is trivially non-decreasing.

4.2 Measuring Progress in Finite Domain Reductions

The concept of the weakest variable bounds developed in Section 3.2 gives us a natural starting point for finite domain reductions. As bounds propagation converges towards its unique fixed point, the endpoint is also well defined. Notice that the bounds which are infinite at the endpoint, also had to be infinite at the starting point, meaning that no change was made on this bound. The rest of the bounds are either infinite at the beginning, in which case we can compute the weakest bound by Algorithm 1, or the bound is finite at both the start and the end.

Our main approach is to measure the relative progress of each individual bound from its weakest value towards the limit value. Given a variable $j \in \{1, \dots, n\}$ we will denote by $\mathcal{P}_{\ell_j} \in \mathbb{R}$ and $\mathcal{P}_{u_j} \in \mathbb{R}$ the scores which measure the amount of progress made on its lower and upper bounds ℓ_j and u_j , respectively, at a given point in time. Afterward, we will combine the scores of all the variable bounds into the global progress in the form of a single scalar value $\mathcal{P}^{\text{fin}} \in \mathbb{R}$, which measures the global progress in finite domain reductions at a given point in time.

For variable j , \mathcal{P}_{ℓ_j} and \mathcal{P}_{u_j} are computed as

$$\mathcal{P}_{\ell_j} = \begin{cases} \frac{\ell_j - \bar{\ell}_j}{\ell_j^l - \bar{\ell}_j} & \text{if } \ell_j > \bar{\ell}_j \text{ and } \bar{\ell}_j \neq \ell_j^l, \\ 0 & \text{otherwise} \end{cases}, \quad (10a)$$

and

$$\mathcal{P}_{u_j} = \begin{cases} \frac{\bar{u}_j - u_j}{\bar{u}_j - u_j^l} & \text{if } u_j < \bar{u}_j \text{ and } \bar{u}_j \neq u_j^l, \\ 0 & \text{otherwise} \end{cases}. \quad (10b)$$

Given the vectors of scores for individual bounds $\mathcal{P}_{\ell} \in \mathbb{R}^n$ and $\mathcal{P}_u \in \mathbb{R}^n$, we calculate \mathcal{P}^{fin} as

$$\mathcal{P}^{\text{fin}} = \|\mathcal{P}_{\ell}\|_1 + \|\mathcal{P}_u\|_1 = \sum_j (\mathcal{P}_{\ell_j} + \mathcal{P}_{u_j}), \quad (11)$$

where $\|\cdot\|_1$ denotes the ℓ_1 norm. It holds that $\mathcal{P}_{\ell_j}, \mathcal{P}_{u_j} \in [0, 1]$ and

$$\mathcal{P}^{\text{fin}} \leq |\{j = 1, \dots, n : \bar{\ell}_j \neq \ell_j^l\}| + |\{j = 1, \dots, n : \bar{u}_j \neq u_j^l\}|. \quad (12)$$

This maximum score is algorithm-independent and can be precomputed for each instance. This makes it possible to normalize the maximum score to, e.g., 100%. Again, because IBTAs never relax bounds, this progress function is trivially non-decreasing.

4.3 Implementation Details

To precompute $\bar{\ell}$ and \bar{u} , we implemented Algorithm 1. To obtain ℓ^l and u^l , any correct bounds propagation algorithm can be run on the original problem, assuming that it propagates the problem to the fixed point (no tolerance-based stopping criteria).

Computing the progress measure is expensive relative to the amount of work that bounds propagation normally performs. Hence, it can considerably slow down the execution and incur unrealistic runtime measurements. To avoid this effect, we proceed as follows in our implementation. First, we run the bounds propagation algorithm together with progress measure computation and record the scores after each round. Then, we run the same bounds propagation algorithm but without the progress measure calculation and record the time elapsed to the end of each round. This gives us progress scores and times for each round, but also the time it took to reach the scores at the end of each round.

5 Applications of the Progress Measure

In this section, we apply the progress measure in order to answer two questions of practical relevance. In Section 5.1, we first describe the experimental setup that will form the base for subsequent evaluations. In Section 5.2 we show that MIP instances in practice rarely cause IBTAs to stall prematurely, i.e., have very slow progress followed by larger improvements thereafter, a concern brought up in [7] (see Section 2.3). In Section 5.3, we show that the newly-developed GPU-based propagation algorithm from [26] is even more competitive in a practical setting than reported in the original paper.

5.1 Experimental Setup

We will refer to two linear constraint propagation algorithms:

1. ***gpu_prop*** is the GPU-based algorithm from [26], and
2. ***seq_prop*** is the canonical sequential propagation as described in e.g. [1]. Our implementation closely follows the implementation in the academic solver SCIP [12].

We use the MIPLIB 2017 test set, which is currently the most adopted and widely used testbed of MIP instances [13]. This test set contains 1065 instances, however, the open-source MIP file reader we used had problems with reading 133 instances, leaving the test set at 932 instances. On 72 instances *gpu_prop* and *seq_prop* failed to obtain the same fixed point (due to e.g., numerical difficulties and other problems), and we remove these instances from the test set as well. Additionally, we impose an iteration limit of 100 for both propagation algorithms, with 2 instances hitting this limit.

During MIP solving, the case where no bound changes are found during propagation is valid and common. However, this is of no interest to us here, as we could make no measurements of progress. There are 310 such instances in the test set. Furthermore, 8 instances with challenging numerical properties showed inconsistent behavior with our implementations, and we remove these instances from the test set as well. Finally, the test set used for the evaluations is left with 540 MIP instances.

In terms of hardware, we execute the *gpu_prop* algorithm on a NVIDIA Tesla V100 PCIe 32GB GPU, and the *seq_prop* algorithm on a 24-core Intel Xeon Gold 6246 @ 3.30GHz with 384 GB RAM CPU. All executions are performed with double-precision arithmetic.

As we use this test set to measure the progress of propagation algorithms, they were run until the fixed point is reached with the progress recorded as described in Section 4.3. In this setting, IBTAs terminate after no bound changes are found at a given propagation round. What this means is that the last two rounds will both have the same maximum score (no bound changes in the last round). Because this feature reflects the design of the algorithms, in the results we assume that the maximum score is reached after the last round, and not after the second-to-last round. This is equivalent to removing the second-to-last round. On the other hand, when the (finite or infinite) score does not change its value between two rounds which are not the last and the second-to-last one, we assume that the score is reached at the first time when it is recorded.

Due to implementation reasons, we will sample progress after each propagation round of an algorithm, rather than after every single bound change. Then, we use linear interpolation to build the progress functions \mathcal{P}^{fin} and \mathcal{P}^{inf} and thus obtain an approximation of the true progress function.

5.2 Analyzing Premature Stalling in Linear Constraint Propagation

First, we have to quantitatively define the premature stalling effect. The danger it poses is that the stopping criteria might terminate the algorithm after an iteration with slow progress, and potentially miss on substantial improvements later on. While infinite domain reductions are usually easy to find by bounds propagation algorithms, they are nevertheless considered significant and the algorithm is usually not stopped after an iteration that contains these tightenings [1]. Accordingly, we will reflect this in our premature stalling effect definition.

We slightly adapt the notation introduced in Section 4.2 and define the progress in finite domain reductions as a function of time denoted by $\mathcal{P} : [0, 100] \rightarrow [0, 100]$. Observe that the input (time) and output (progress) of this function are normalized to values between 0 and 100. In this notation we assume that \mathcal{P} is continuous and twice differentiable, however, in practice, the progress is sampled only after each propagation round and \mathcal{P} built by linear interpolation. In our implementation, we approximate the derivatives of \mathcal{P} by second-order accurate central differences in the interior points and either first or second-order accurate one-sided (forward or backward) differences at the boundaries [22, 11]. Additionally, given a propagation round r , $t(r)$ denotes the normalized time at the end of propagation round r . All derivatives are w.r.t. time: $\mathcal{P}' = \frac{d}{dt}\mathcal{P}$. We denote by $k \in \mathbb{N}$ the number of iterations the propagation algorithm takes to reach the fixed point and by $\ell^r, u^r \in \mathbb{R}^n$ the arrays of lower and upper bounds at iteration r , respectively. Then, the premature stalling effect is defined as follows.

► **Definition 8.** *Let \mathcal{P} be a progress function of finite domain reductions for the propagation of a given MIP instance. Then, the propagation algorithm is said to prematurely stall with coefficients $p, q \in \mathbb{R}_{\infty \geq 0}$ at round $r \in \{2, \dots, k\}$ if the following conditions are true:*

1. *there does not exist $j \in \{1, \dots, n\}$ such that $\ell_j^{r-1} = -\infty$ and $\ell_j^r \in \mathbb{R}$,*
2. *there does not exist $j \in \{1, \dots, n\}$ such that $u_j^{r-1} = \infty$ and $u_j^r \in \mathbb{R}$,*
3. *$\mathcal{P}'(t(r)) < p$, and*
4. *there exists $x \in [t(r), 100]$ such that $\mathcal{P}''(x) > q$.*

The first two conditions simply state that there were no infinite domain reductions in round r . To understand the third condition, let $p = 0.1$ at r . This would mean that the algorithm is progressing at a rate of 1 percent of progress in 10 percent of the time at r (recall the normalized domains of \mathcal{P}). Taking another derivative and looking at the remainder of the time interval reveals if this rate will increase (is greater than 0), meaning that there are bigger improvements to follow than the improvements the algorithm is currently making. The parameter $q \geq 0$ allows quantification of increase in size of these improvements. Also, recall from Section 4.2 that \mathcal{P} is non-decreasing and hence $\mathcal{P}'(t) \geq 0$ for all $t \in [0, 100]$. With this, we can now detect instances where slow progress is followed by a significant increase in improvements.

Table 1 reports the number of premature stalls in the test set for several different combinations of parameters p and q . Notice that the 310 instances for which no bound changes are found cannot stall by definition. Additionally, 57 instances in the test set only recorded infinite domain reductions, and these instances also cannot prematurely stall by definition. The results of testing the remaining 432 instances which do record at least one finite domain reduction for premature stalling are shown in Table 1.

Let us first look into the results for *seq_prop*. From the first row of the table, we can see that only 48 instances experience any kind of increase in the second derivative during the execution, i.e., the improvements get smaller in time for all but 48 instances in the test set (equivalently, \mathcal{P} is concave for all but 48 instances). From the second row, we can see

■ **Table 1** Number of premature stalls in the test for different values of parameters p and q .

p	q	# stalls	
		seq_prop	gpu_prop
∞	0.0	48	44
0.1	0.0	14	18
0.1	0.2	1	0
0.1	0.5	0	0
0.5	0.5	1	0
0.5	2.0	0	0

that among these 48 instances that experience any kind of second derivative increase, 14 experience slow progress of $p = 0.1$ at least once during their execution. Among these, only 1 instance experiences an increase in second derivative of more than $q = 0.2$ following the slow progress of $p = 0.1$. If we further restrict the increase in the second derivative to $q = 0.5$, then no instances are shown to stall prematurely. In the last row we see that even if the slow progress is relaxed to $p = 0.5$, there are no instances that record a more significant increase in the second derivative of 2.0.

Additionally, even though *gpu_prop* performed similarly to *seq_prop* with respect to stalling, we can still observe that it is on average less susceptible to premature stalling than *seq_prop*, as it recorded a smaller or equal amount of instances with premature stalling for all but one parameter combinations.

We conclude that in practice, the premature stalling effect seems to occur only rarely and on individual instances. This shows that termination criteria based on local progress are reasonable.

5.3 Analyzing GPU-parallel Bounds Propagation in Practice

As pointed out in Section 2.3, *gpu_prop* traverses a potentially different sequence of bounds from the start to the fixed point than *seq_prop*. Because of this, computational experiments in [26] report the speedup of *gpu_prop* over *seq_prop* for propagation runs to the fixed point. As bounds propagation is stopped early in practice, we will now use the progress measure to compare the two algorithms when stopped at different points in the execution. For each instance in the set, given a progress value $x \in [0, 100]$, the speedup of *gpu_prop* over *seq_prop* is computed by $t_x^{\text{seq_prop}} / t_x^{\text{gpu_prop}}$, where t_x is the wall-clock time the algorithm takes to reach progress value x .² Then, the geometric mean of speedups over all the instances in the test set is reported. The results are shown on Figure 2. When a given instance only has bound changes in the infinite phase, it is excluded from the finite phase comparisons (57 instances). Likewise, instances with only finite progress are removed from the infinite phase (164 instances).

As we can see, for the propagation to the fixed point (100 percent progress), *gpu_prop* is about 4.9 times faster than *seq_prop* in finite domain reductions. For the infinite domain reductions, *gpu_prop* is a factor of about 5.4 times faster than *seq_prop*. Next, we can see that the speedup is minimal at the fixed point, i.e., for any progress value between 10 and 100, *gpu_prop* increases its speedup over *seq_prop* compared to the fixed-point speedup. The

² For $x = 100$, we get the identical speedup at the fixed point evaluation as done in [26].

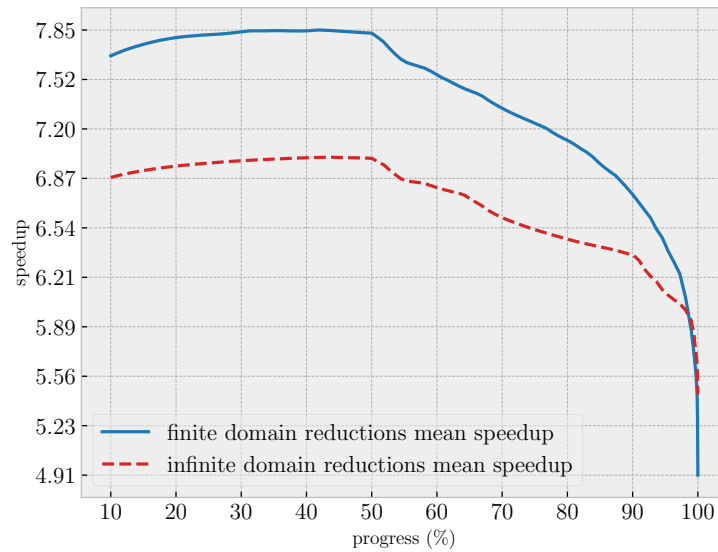


Figure 2 Speedup of the finite and the infinite domain reductions of *gpu_prop* over *seq_prop* for different percentages of progress made.

maximum speedups of around 7.8 for the finite domain reductions and about 7.0 for infinite domain reductions are achieved at the progress of roughly 50 percent. Additionally, notice that in the last few percent of progress there is a steep drop in speedup. This means that even for very weak stopping criteria which would stop the algorithms at the same point just before the limit is reached, *gpu_prop* would significantly increase its speedup over *seq_prop*. We conclude that *gpu_prop* is even more competitive against *seq_prop* in conjunction with stopping criteria than for the case of propagation to the fixed point.

6 Outlook

In this work, we proposed a method to measure progress achieved by a given algorithm in the propagation of linear constraints with continuous and/or discrete variables. We showed how such a measure can be used to answer questions of practical relevance in the field of Mixed-Integer Programming.

One question that remains open is to what extent the finite reference bounds produced by the weakest bounds procedure used here are actually realized by at least one iterative bounds tightening algorithm. The current procedure only guarantees that they are finite if iterative bounds propagation can produce a finite bound, and that no iterative bounds propagation algorithm can produce a weaker bound. A deeper analysis could yield a refined method to produce weakest bounds that are tightest in the sense that they are actually achieved by at least one iterative bounds propagation algorithm. This is part of future research and could provide a stronger version of the framework.

Though our development was described for linear constraints, there are no conceptual barriers that prevent the notion of weakest bounds to be extended to more general classes of constraints. We demonstrated how the key issue of unbounded variable domains can be solved in order to obtain an algorithm-independent measure of progress. In this sense, our method is also relevant for constraint systems on (partially) unbounded domains, where normalization can be nontrivial. An important example is the class of factorable programs from the field of Global Optimization and Mixed-Integer Nonlinear Programming.

References

- 1 Tobias Achterberg. *Constraint Integer Programming*. PhD thesis, TU Berlin, 2009.
- 2 Tobias Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, January 2009. doi:10.1007/s12532-008-0001-1.
- 3 Tobias Achterberg, Robert E. Bixby, Zonghao Gu, Edward Rothberg, and Dieter Weninger. Presolve reductions in mixed integer programming. *INFORMS Journal on Computing*, 32(2):473–506, 2020. doi:10.1287/ijoc.2018.0857.
- 4 Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. In Michael Jünger and Gerhard Reinelt, editors, *Facets of Combinatorial Optimization: Festschrift for Martin Grötschel*, pages 449–481. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-38189-8_18.
- 5 Ernst Althaus, Alexander Bockmayr, Matthias Elf, Michael Jünger, Thomas Kasper, and Kurt Mehlhorn. Scil – symbolic constraints in integer linear programming. In Rolf Möhring and Rajeev Raman, editors, *Algorithms – ESA 2002*, pages 75–87, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
- 6 Ionuț Aron, John N. Hooker, and Talys H. Yunes. Simpl: A system for integrating optimization techniques. In Jean-Charles Régin and Michel Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 21–36, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- 7 Pietro Belotti, Sonia Cafieri, Jon Lee, and Leo Liberti. Feasibility-based bounds tightening via fixed points. In Weili Wu and Ovidiu Daescu, editors, *Combinatorial Optimization and Applications, Proc. of COCOA 2010*, pages 65–76, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-17458-2_7.
- 8 Alexander Bockmayr and Thomas Kasper. Branch and infer: A unifying framework for integer and finite domain constraint programming. *INFORMS Journal on Computing*, 10(3):287–300, 1998. doi:10.1287/ijoc.10.3.287.
- 9 Lucas Bordeaux, George Katsirelos, Nina Narodytska, and Moshe Y. Vardi. The complexity of integer bound propagation. *J. Artif. Int. Res.*, 40(1):657–676, 2011.
- 10 C. W. Choi, W. Harvey, J. H. M. Lee, and P. J. Stuckey. Finite domain bounds consistency revisited. In Abdul Sattar and Byeong-ho Kang, editors, *AI 2006: Advances in Artificial Intelligence*, pages 49–58, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- 11 B. Fornberg. Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of Computation*, 51:699–706, 1988.
- 12 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.
- 13 Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp M. Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans D. Mittelmann, Derya Ozyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. Technical report, Optimization Online, 2019. URL: http://www.optimization-online.org/DB_HTML/2019/07/7285.html.
- 14 Warwick Harvey and Peter J. Stuckey. Improving linear constraint propagation by changing constraint representation. *Constraints*, 8(2):173–207, 2003. doi:10.1023/a:1022323717928.
- 15 Thorsten Koch, Alexander Martin, and Marc E. Pfetsch. Progress in academic computational integer programming. In Michael Jünger and Gerhard Reinelt, editors, *Facets of Combinatorial Optimization: Festschrift for Martin Grötschel*, pages 483–506. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-38189-8_19.

- 16 A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960. URL: <http://www.jstor.org/stable/1910129>.
- 17 Olivier Lhomme. Consistency techniques for numeric csps. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'93, page 232–238, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.
- 18 Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977. doi:10.1016/0004-3702(77)90007-8.
- 19 Kimbal Marriott and Peter Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, February 1998. doi:10.7551/mitpress/5625.001.0001.
- 20 Roger Mohr and Gérard Masini. Good old discrete relaxation. In *Proceedings of the 8th European Conference on Artificial Intelligence*, ECAI'88, page 651–656, USA, 1988. Pitman Publishing, Inc.
- 21 George Nemhauser and Laurence Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, Inc., 1988. doi:10.1002/9781118627372.
- 22 Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical mathematics*, volume 37. Springer, 2007.
- 23 Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier Science Inc., USA, 2006.
- 24 Martin W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.
- 25 Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *ACM Trans. Program. Lang. Syst.*, 31(1), 2008. doi:10.1145/1452044.1452046.
- 26 Boro Sofranac, Ambros Gleixner, and Sebastian Pokutta. Accelerating domain propagation: An efficient gpu-parallel algorithm over sparse matrices. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 1–11, 2020. doi:10.1109/IA351965.2020.00007.
- 27 Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). *The Journal of Logic Programming*, 37(1):139–164, 1998. doi:10.1016/S0743-1066(98)10006-7.

Differential Programming via OR Methods

Shannon Sweitzer ✉

Department of Industrial and Systems Engineering,
University of Southern California, Los Angeles, CA, USA

T. K. Satish Kumar ✉

Department of Computer Science, Department of Physics and Astronomy,
Department of Industrial and Systems Engineering, Information Sciences Institute,
University of Southern California, Los Angeles, CA, USA

Abstract

Systems of ordinary differential equations (ODEs) and partial differential equations (PDEs) are extensively used in many fields of science, including physics, biochemistry, nonlinear control, and dynamical systems. On the one hand, analytical methods for solving systems of ODEs/PDEs mostly remain an art and are largely insufficient for complex systems. On the other hand, numerical approximation methods do not yield a viable analytical form of the solution that is often required for downstream tasks. In this paper, we present an approximate approach for solving systems of ODEs/PDEs analytically using solvers like Gurobi developed in Operations Research (OR). Our main idea is to represent entire functions as Bézier curves/surfaces with to-be-determined control points. The ODEs/PDEs as well as their boundary conditions can then be reformulated as constraints on these control points. In many cases, this reformulation yields quadratic programming problems (QPPs) that can be solved in polynomial time. It also allows us to reason about inequalities. We demonstrate the success of our approach on several interesting classes of ODEs/PDEs.

2012 ACM Subject Classification Applied computing

Keywords and phrases Differential Programming, Operations Research, Bézier Curves

Digital Object Identifier 10.4230/LIPIcs.CP.2021.53

1 Introduction

Systems of ordinary differential equations (ODEs) and partial differential equations (PDEs) are so extensively used that they are the mathematical language of many sciences. The following are just a few examples. In physics, they are used to describe harmonic motion, radioactive decay, and propagation of electromagnetic waves [18]. In biochemistry, they are used to model biological processes ranging from the biosynthesis of phospholipids and proteins to the growth of cancer cells and viral dynamics [2]. In control theory, they are used to describe the behavior of dynamical systems and optimal strategies for controlling them [6]. An abundance of other applications can be found in many other sciences.

Plenty of analytical methods have been developed for solving ODEs/PDEs. These include standard techniques like separation of variables, the method of characteristics, integral transform, change of variables, fundamental solutions, and superposition [14], and newer techniques that utilize Lie groups [13] and Bäcklund transforms [9]. Despite the existence of many such techniques, the applicability of analytical methods has been restricted to special classes of systems of ODEs/PDEs such as first-order systems, second-order systems with constant coefficients, and second-order systems with variable coefficients. A comprehensive study of ODEs/PDEs amenable to different analytical methods can be found in [14].

Many numerical approximation methods have also been developed to address the limitations of analytical methods. These include the finite element method (FEM) [20], the finite difference method (FDM) [10], and the finite volume method (FVM) [3]. Although numerical approximation methods have very general applicability, they too have drawbacks



© Shannon Sweitzer and T. K. Satish Kumar;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 53; pp. 53:1–53:15

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of their own. They don't return a solution in an analytical form. Instead, they return a solution as a set of values calculated at discrete points on a meshed geometry. This makes the solution unviable for downstream tasks in which analytical operations may be involved.

While fitting high-order polynomials on the set of values calculated at discrete points can sometimes salvage an analytical form and enable some downstream derivatives, it is not a satisfactory method either. This is because enforcing a desired property, such as an inequality constraint, at the discrete points does not enforce it everywhere. For example, in a system of ODEs that describes the velocity profile of a robot, enforcing a maximum velocity constraint on a finite number of discrete time points doesn't enforce it at all time points if high-order polynomials are used for interpolation.¹

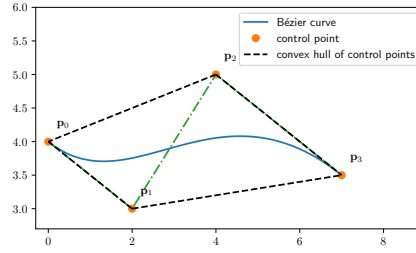
Because of the problems associated with both analytical and numerical approximation methods, controller design and synthesis problems in many domains still remain very hard. Such problems involve decision variables in addition to ODEs/PDEs. Typically, the ODEs/PDEs can be solved in a "simulation" phase only when decision variables have assigned values. Neither the analytical nor the numerical approximation method can facilitate the search for optimal values of the decision variables themselves. For example, in nanoscale photonics, a set of teflon dielectric cylinders are used to focus electromagnetic power. Given values for the decision variables, i.e., positions of the dielectric cylinders, the PDEs that describe the resulting distribution of electromagnetic power can be solved numerically. However, the optical filter design problem of choosing where to optimally place the dielectric cylinders themselves is very hard.

In this paper, we present an approximate approach for solving systems of ODEs/PDEs analytically using solvers like Gurobi developed in Operations Research (OR). Our main idea is to represent entire functions as Bézier curves/surfaces with to-be-determined control points. Bézier curves have a number of useful mathematical properties [11]. They can uniformly approximate any continuous function; their derivatives are also Bézier curves; and a Bézier curve lies entirely within the convex hull of its control points. Because of their attractive mathematical properties, Bézier curves have been widely used in many application domains, including computer graphics [12], computer-aided design [5], path planning [1], and trajectory planning [16]. Using the Bézier curve/surface representation in our case, we show that ODEs/PDEs as well as their boundary conditions can be reformulated as constraints on their control points.

Our proposed approach has several advantages. First, not only does it have the general applicability of numerical approximation methods but it also produces an analytical form that is useful for downstream tasks. Such downstream tasks are commonplace in physics-based machine learning where the *principle of least action* can be expressed as a second-order PDE, known as the Euler-Lagrange equation [15], on which further data-driven inferences must be carried out.

Second, our approach uses *control points* instead of a *discretization* of the independent variables' domains. For example, consider a function $f(t)$ with $t \in [0, T]$. Numerical approximation methods require the discretization of the interval $[0, T]$. However, discretization not only necessitates an increase in the number of discrete points for growing values of T but also creates a dependency on interpolation methods for values of t between the discrete

¹ There exist other numerical approximation methods, called meshfree methods [7], useful for simulations in which the discrete points can be dynamically created or destroyed. Meshfree methods can also be combined with FEM, FDM, or FVM to yield hybrid methods [4]. But, in general, they too have the same drawbacks.



■ **Figure 1** Illustrates an important property of Bézier curves: A Bézier curve is enclosed entirely within the convex hull of its control points. Here, the Bézier curve has 4 control points in a 2-dimensional space.

points. In contrast, our approach represents $f(t)$ as a linear combination of Bernstein basis polynomials on t . The to-be-determined coefficients of the linear combination are its control points. $f(t)$ is therefore automatically defined for all values of $t \in [0, T]$. Moreover, the number of control points depends on the complexity of $f(t)$ and not on the domain size of t . In fact, some of the benefits of such a representation are used in recent numerical approximation methods like isogeometric analysis [8].

Third, although our approach uses only a finite number of control points, it allows us to enforce desired properties at all points on the resulting solution rather than at a set of discrete points. In particular, our approach also allows for inequalities, e.g., to enforce the maximum acceleration of a robot at all times. For this reason, we say that our method is more generally applicable to *differential programming*.

Fourth, since our approach reformulates ODEs/PDEs and their boundary conditions as constraints on their control points, they can be combined with other decision variables. This allows us to cast controller design and synthesis problems as optimization problems that don't require expensive simulation. In addition, the nature of the resulting constraints provides insights into the nature of the ODEs/PDEs, allowing us to draw parallels between the mathematical theory of ODEs/PDEs and the computational theory of optimization. In fact, upon reformulation, many interesting classes of ODEs/PDEs yield quadratic programming problems (QPPs) that can be solved in polynomial time.

In this paper, we demonstrate the success of our approach on several interesting classes of ODEs/PDEs. However, given the enormous literature relevant to ODEs/PDEs, our paper can only qualify as a feasibility study in an important direction with preliminary results.

2 Background

In mathematics, Bernstein basis polynomials of degree n are defined to be

$$B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i \in \{0, 1 \dots n\},$$

where $\binom{n}{i}$ is the binomial coefficient equal to $\frac{n!}{i!(n-i)!}$.

A k -dimensional Bézier curve of degree n is of the form

$$\mathbf{B}(t) = \sum_{i=0}^n \mathbf{p}_i B_{i,n}(t), \quad t \in [0, 1],$$

where $P = \{\mathbf{p}_0, \mathbf{p}_1 \dots \mathbf{p}_n\}$ is the set of $n + 1$ k -dimensional *control points*. Therefore, it is a curve parameterized by t and interpretable as a linear combination of the Bernstein basis polynomials of degree n . The coefficients of the linear combination are the $n + 1$ k -dimensional control points.

A Bernstein polynomial $B(t)$ of degree n is a 1-dimensional Bézier curve of degree n . Therefore, it is a linear combination of the Bernstein basis polynomials of degree n . The coefficients of the linear combination are $n + 1$ real numbers acting as 1-dimensional control points.

Bernstein polynomials and Bézier curves have many useful mathematical properties. For example, the Weierstrass Approximation Theorem [11] establishes that any continuous real-valued function defined on the interval $[0, 1]$ can be uniformly approximated by Bernstein polynomials.

Two other useful properties of Bernstein polynomials and Bézier curves are with respect to their derivatives and their control points. The derivative of a Bézier curve $\mathbf{B}(t)$ of degree n is a Bézier curve of degree $n - 1$. In particular,

$$\frac{d\mathbf{B}(t)}{dt} = \sum_{i=0}^{n-1} \mathbf{p}'_i B_{i,n-1}(t),$$

where the control point $\mathbf{p}'_i = n(\mathbf{p}_{i+1} - \mathbf{p}_i)$, for $i \in \{0, 1 \dots n - 1\}$.

A Bézier curve $\mathbf{B}(t)$ is bounded by the convex hull of its control points P for $t \in [0, 1]$, as shown in Figure 1. Intuitively, this is because for any given value of $t \in [0, 1]$: (a) $B_{i,n}(t) \geq 0$ for $i \in \{0, 1 \dots n\}$, and (b) $\sum_{i=0}^n B_{i,n}(t) = 1$. Therefore, $\mathbf{B}(t)$ for $t \in [0, 1]$ is interpretable as a non-negative linear combination of its control points, necessitating its presence in the convex hull. In particular, $\mathbf{B}(0) = \mathbf{p}_0$ and $\mathbf{B}(1) = \mathbf{p}_n$. In the case of Bernstein polynomials, the control points are 1-dimensional real numbers, and $B(t)$ lies entirely within $[\inf(P), \sup(P)]$.

3 Solving ODEs

To solve ODEs/PDEs using Bézier curves/surfaces, we have to develop the general theory in stride. It is best illustrated through examples and case studies. In this and the next sections, we apply our proposed methodology to a series of examples that are chosen to be in increasing order of complexity and generality. Unless stated otherwise, we use the interval $[0, 1]$ for all independent variables since the Bernstein basis polynomials are also defined within the same interval.

3.1 First-Order Homogeneous Linear ODEs with Constant Coefficients

An ODE is said to be *linear* if it is of the form

$$a_0(t)y(t) + a_1(t)y'(t) \dots a_m(t)y^{(m)}(t) + b(t) = 0, \quad (1)$$

where $a_0(t), a_1(t) \dots a_m(t)$ and $b(t)$ are differentiable functions of t , and the functions $y(t), y'(t) \dots y^{(m)}(t)$ are the successive derivatives of the to-be-determined function $y(t)$.

A first-order linear ODE has $m = 1$. Moreover, a first-order homogeneous linear ODE has $b(t) = 0$. Consider a first-order homogeneous linear ODE with constant coefficients, i.e., $a_0(t)$ and $a_1(t)$ are constants denoted by a_0 and a_1 , respectively. Therefore, the ODE is of the form

$$a_0 y(t) + a_1 y'(t) = 0. \quad (2)$$

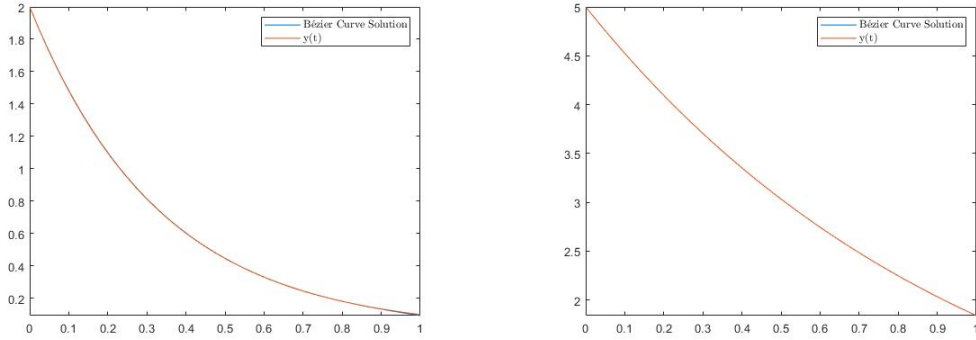
(a) $3y(t) + y'(t) = 0; y(0) = 2$.(b) $y(t) + y'(t) = 0; y(0) = 5$.

Figure 2 Shows the solutions of two first-order homogeneous linear ODEs with constant coefficients. The solutions generated by the Bézier curve method (blue) match the analytical solutions (orange) exactly. (The blue color is not visible because of the exact match.) The analytical solution for (a) is $y(t) = 2e^{-3t}$. The analytical solution for (b) is $y(t) = 5e^{-t}$. In both cases, 6 control points and 6 test points were used, resulting in a running time of 0.60 s for (a) and 0.45 s for (b).

The above ODE has a known family of solutions of the form $\{Ce^{(-a_0/a_1)t} : C \in \mathbb{R}\}$. A particular solution from this family can be identified based on the initial conditions. Consider the example

$$3y(t) + y'(t) = 0, \quad (3)$$

with the accompanying initial condition $y(0) = 2$.

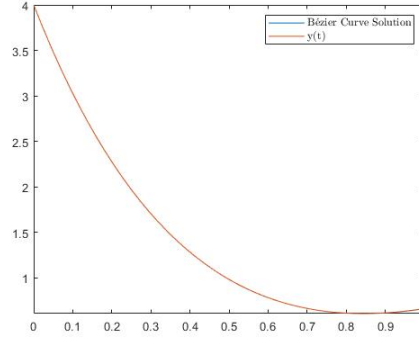
Suppose we represent $y(t)$ using a 1-dimensional Bézier curve $\mathbf{B}(t)$ of degree n and $n + 1$ to-be-determined control points $P = \{p_0, p_1 \dots p_n\}$. By substituting $\mathbf{B}(t)$ for $y(t)$ and its derivative $\mathbf{B}'(t)$ for $y'(t)$, the problem reduces to

$$3 \sum_{i=0}^n p_i B_{i,n}(t) + \sum_{i=0}^{n-1} n(p_{i+1} - p_i) B_{i,n-1}(t) = 0. \quad (4)$$

The initial condition reduces to $p_0 = 2$. In essence, this reduced formulation enforces the polynomial $g(t) = 3 \sum_{i=0}^n p_i B_{i,n}(t) + \sum_{i=0}^{n-1} n(p_{i+1} - p_i) B_{i,n-1}(t)$ of degree n to be identically equal to 0. Since this can happen only when all the coefficients of the powers of t are individually equal to 0, the problem further reduces to linear equalities on the control points.

Although linear equalities can be solved very efficiently, our first attempt fails for the following reason. We have $n + 1$ linear constraints coming from $g(t) \equiv 0$; and we have 1 linear constraint coming from the initial condition. This accounts for a total of $n + 2$ linear constraints on $n + 1$ variables, creating an over-constrained problem that doesn't necessarily have a solution.

In a second attempt, we split the constraints to *hard* and *soft* constraints. The linear constraints coming from the initial conditions are retained as hard constraints, while the linear constraints coming from $g(t) \equiv 0$ are relaxed to be soft constraints, with a penalty for violation measured using squared error. Of course, the soft constraints should fully incentivize enforcing $g(t) \equiv 0$. Therefore, the squared error is measured on $g(t)$ evaluated at $M \geq n + 1$ test points sampled from the interval $[0, 1]$.

(a) $2y(t) + y'(t) = 5t - 3; y(0) = 4$.

■ **Figure 3** Shows the solution of a first-order non-homogeneous linear ODE with constant coefficients, when $b(t)$ is a polynomial. The solution generated by the Bézier curve method (blue) matches the analytical solution (orange) exactly. (The blue color is not visible because of the exact match.) The analytical solution is $y(t) = \frac{27}{4}e^{-2t} + \frac{5}{2}t - \frac{11}{4}$. Here, 6 control points and 6 test points were used, resulting in a running time of 0.61 s.

Let $t_1, t_2 \dots t_M$ be the test points. We formulate the following QPP:

$$\begin{aligned} & \underset{p_0, p_1 \dots p_n}{\text{Minimize}} && \sum_{i=1}^M (g(t_i))^2 \\ & \text{s.t.} && p_0 = 2. \end{aligned} \tag{5}$$

Solving the QPP yields optimal values of $p_0, p_1 \dots p_n$, which in turn can be used to construct the desired $\mathbf{B}(t)$ as an approximation for $y(t)$. Figure 2(a) shows the Bézier curve solution to the above problem. Figure 2(b) shows the Bézier curve solution to another first-order homogeneous linear ODE with constant coefficients given by

$$y(t) + y'(t) = 0, \tag{6}$$

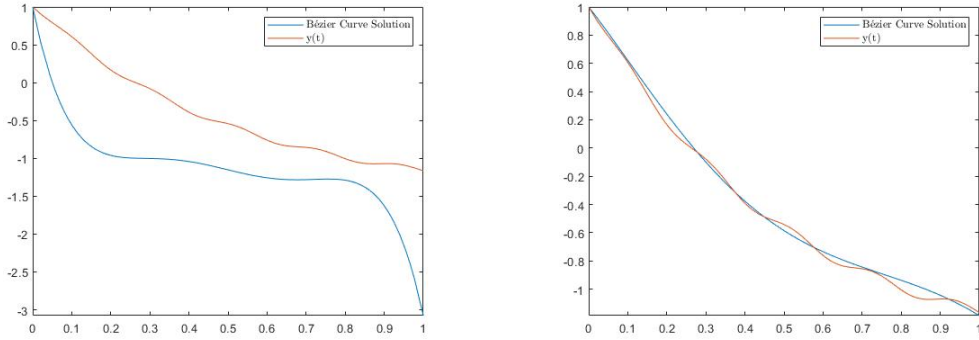
with the accompanying initial condition $y(0) = 5$.

For the QPP solver, we used CVX, a MatLab R2020b package for specifying and solving convex programs. We report the CVX running times for all examples discussed in this paper. All experiments were conducted on a laptop with a 2.8GHz Quad-Core Intel Core i7 processor and 16GB 2133MHz DDR4 memory. We used the default CVX settings for all experiments.

3.2 First-Order Non-Homogeneous Linear ODEs with Constant Coefficients

We now examine first-order non-homogeneous linear ODEs with constant coefficients. This is similar to the previous subsection, except that $b(t)$ is not necessarily 0. We refer to a non-zero $b(t)$ as the non-homogeneity term.

If $b(t)$ is a polynomial, the formulation using Bézier curves again reduces to a case of polynomial equivalence, and our proposed methodology continues to be directly applicable. However, we note that if the degree of $b(t)$ exceeds n , then the number of test points M should be $\geq \deg(b(t)) + 1$.

(a) $2y(t) + y'(t) = \sin(30t) - 3; y(0) = 1.$ (b) $2y(t) + y'(t) = \sin(30t) - 3; y(0) = 1.$

■ **Figure 4** Shows the solutions of a first-order non-homogeneous linear ODE with constant coefficients, when $b(t)$ is not a polynomial. The solution generated by the Bézier curve method (blue) better approximates the analytical solution (orange) with an increasing number of test points. The analytical solution is $y(t) = (1145e^{-2t} + \sin(30t) - 15 \cos(30t) - 678)/452$. In (a), 6 control points and 6 test points were used, resulting in a running time of 0.38 s. In (b), 6 control points and 20 test points were used, resulting in a running time of 0.57 s.

Consider the following example:

$$\begin{aligned} 2y(t) + y'(t) &= 5t - 3 \\ y(0) &= 4. \end{aligned} \quad (7)$$

The analytical solution of this ODE is given by $y(t) = \frac{27}{4}e^{-2t} + \frac{5}{2}t - \frac{11}{4}$. Figure 3 shows this analytical solution and the Bézier curve solution obtained using $n = 5$ and $M = 6$. Once again, the Bézier curve solution provides very accurate results.

If $b(t)$ is not a polynomial, the formulation is not reducible to one of polynomial equivalence. Nonetheless, our method can still be used since Bézier curves can uniformly approximate any function [11].

Consider the following example:

$$\begin{aligned} 2y(t) + y'(t) &= \sin(30t) - 3 \\ y(0) &= 1. \end{aligned} \quad (8)$$

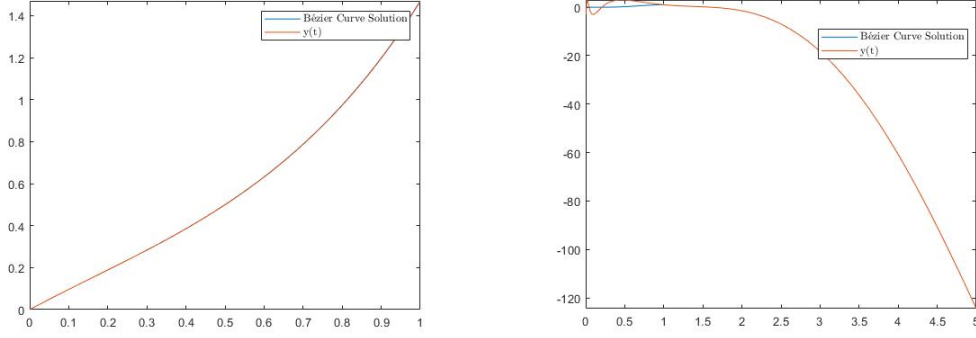
The non-homogeneity term $b(t)$ is no longer a polynomial and is in fact very oscillatory. The analytical solution of this ODE is given by $y(t) = (1145e^{-2t} + \sin(30t) - 15 \cos(30t) - 678)/452$. Figure 4(a) shows that the Bézier curve solution has a large deviation from the analytical solution when $n = 5$ and $M = 6$. However, Figure 4(b) shows that the accuracy of the Bézier curve solution improves significantly as M increases, i.e., when $n = 5$ and $M = 20$.

3.3 Higher-Order Linear ODEs

In this subsection, we discuss higher-order linear ODEs. These are linear ODEs with $m > 1$. They can be classified as homogeneous or non-homogeneous, depending on $b(t)$. If $b(t) = 0$, the ODE is homogeneous; otherwise, it is non-homogeneous with $b(t)$ referred to as the non-homogeneity term. We consider two illustrative types of higher-order linear ODEs.

Consider an ODE of the form

$$a_0y(t) + a_1y'(t) + a_2y''(t) = 0, \quad (9)$$



(a) $-6y(t) + y'(t) + y''(t) = 0$; $y(0) = 0, y'(0) = 1$. (b) $\frac{13}{t^2}y(t) - \frac{5}{t}y'(t) + y''(t) = 0$; $y(1) = 1, y'(1) = 0$.

Figure 5 Shows the solutions of two higher-order linear ODEs. (a) shows an ODE with constant coefficients, and (b) shows an Euler-Cauchy ODE. In (a), the solution generated by the Bézier curve method (blue) matches the analytical solution (orange) exactly. (The blue color is not visible because of the exact match.) In (b), the solution generated by the Bézier curve method (blue) approximates the analytical solution (orange). The analytical solution for (a) is $y(t) = \frac{1}{5}(e^{2t} - e^{-3t})$. The analytical solution for (b) is $y(t) = 0.5t^3(2\cos(2\log t) - 3\sin(2\log t))$. In (a), 6 control points and 6 test points were used, resulting in a running time of 0.45 s. In (b), 6 control points and 20 test points were used, resulting in a running time of 0.71 s.

where a_0, a_1 and a_2 are constants. This type of ODE can be solved using the method of characteristic equations, i.e., by finding the roots of the polynomial $a_0 + a_1\lambda + a_2\lambda^2 = 0$ and using them to form linearly independent solutions of the ODE.

We can apply our Bézier curve method to this class of ODEs as well. This is because the second derivative $\frac{d^2\mathbf{B}(t)}{dt^2}$ of a Bézier curve $\mathbf{B}(t)$ is also a Bézier curve. If $\mathbf{B}(t)$ is of degree n , $\frac{d^2\mathbf{B}(t)}{dt^2}$ is of degree $n-2$. Moreover, the control points of $\frac{d^2\mathbf{B}(t)}{dt^2}$ are simple linear combinations of the control points of $\mathbf{B}(t)$.

When $a_0 = -6, a_1 = 1$ and $a_2 = 1$, the analytical solution is given by $y(t) = \frac{1}{5}(e^{2t} - e^{-3t})$ for the initial conditions $y(0) = 0, y'(0) = 1$. Figure 5(a) shows the solution generated by the Bézier curve method. This solution matches the analytical solution exactly.

Now consider the Euler-Cauchy ODE of the form

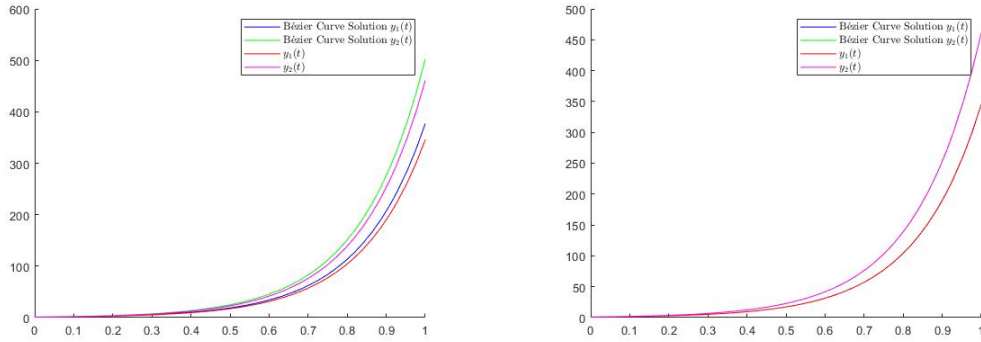
$$\frac{q}{t^2}y(t) + \frac{p}{t}y'(t) + y''(t) = 0, \quad (10)$$

where p and q are constants. This class of ODEs also has well-studied analytical methods for finding general solutions. When $p = -5$ and $q = 13$, the analytical solution is given by $y(t) = 0.5t^3(2\cos(2\log t) - 3\sin(2\log t))$ for the initial conditions $y(1) = 1, y'(1) = 0$. Figure 5(b) shows the solution generated by the Bézier curve method. This solution approximates the analytical solution fairly well.

The Bézier curve solution of the Euler-Cauchy ODE improves with increasing n and M . Similarly, the Bézier curve approximations improve with increasing n and M when $b(t)$ is oscillatory.

4 Solving Systems of ODEs

In this section, we apply our methods to systems of ODEs. In such cases, we have to solve for a vector of unknown functions $\bar{y}(t)$ that satisfy differential equations involving their derivatives. Although systems of ODEs invoke matrices to describe the relationships between

(a) $\bar{y}'(t) = A\bar{y}(t); \bar{y}(0) = (1, 1)^\top$.(b) $\bar{y}'(t) = A\bar{y}(t); \bar{y}(0) = (1, 1)^\top$.

■ **Figure 6** Shows the solutions of a system of ODEs. The solution generated by the Bézier curve method (blue and green for $y_1(t)$ and $y_2(t)$, respectively) better approximates the analytical solution (red and pink, respectively) with increasing n . The analytical solution is $\bar{y}(t) = \frac{1}{7}(1, -1)^\top e^{-t} + \frac{2}{7}(3, 4)^\top e^{6t}$. In (a), 8 control points and 20 test points were used, resulting in a running time of 1.03s. In (b), 11 control points and 20 test points were used, resulting in a running time of 0.90s. (The blue and green colors are not visible because of the exact match.) The columns of A are $(2, 4)^\top$ and $(3, 3)^\top$ in that order.

the various unknown functions and their derivatives, they involve only one independent variable t . We focus our discussion on an example that illustrates the generality of our Bézier curve method.

Consider the system of ODEs

$$\bar{y}'(t) = A\bar{y}(t), \quad (11)$$

where $\bar{y}(t) : \mathbb{R} \rightarrow \mathbb{R}^2$ and A is a 2×2 matrix of real numbers. We have to solve for $\bar{y}(t) = (y_1(t), y_2(t))^\top$.

The family of solutions for this system of ODEs is intimately related to the eigenvalues of A . If A has two distinct real eigenvalues, λ_1 and λ_2 , with corresponding eigenvectors \bar{v}_1 and \bar{v}_2 , the general solution is given by $C_1\bar{v}_1e^{\lambda_1 t} + C_2\bar{v}_2e^{\lambda_2 t}$, for constants C_1 and C_2 . If A has complex conjugate eigenvalues, $\lambda_1 \pm i\lambda_2$, with corresponding eigenvectors $\bar{v}_1 \pm i\bar{v}_2$, the general solution is given by $C_1(\bar{v}_1 \cos(\lambda_2 t) - \bar{v}_2 \sin(\lambda_2 t))e^{\lambda_1 t} + C_2(\bar{v}_1 \sin(\lambda_2 t) + \bar{v}_2 \cos(\lambda_2 t))e^{\lambda_1 t}$. If A has a repeated real eigenvalue λ and the eigenvectors \bar{v}_1 and \bar{v}_2 are linearly independent, the general solution is given by $C_1\bar{v}_1e^{\lambda t} + C_2\bar{v}_2e^{\lambda t}$. If A has only one linearly independent eigenvector \bar{v} , the general solution is given by $C_1\bar{v}e^{\lambda t} + C_2(\bar{v}te^{\lambda t} + \bar{\eta}3^{\lambda t})$, where $\bar{\eta}$ is any solution of $(A - \lambda I)\bar{\eta} = \bar{v}$.

For illustration, suppose $A = \begin{pmatrix} 2 & 3 \\ 4 & 3 \end{pmatrix}$. Its eigenvalues are $\lambda_1 = -1$ and $\lambda_2 = 6$, with corresponding eigenvectors $\bar{v}_1 = (1, -1)^\top$ and $\bar{v}_2 = (3, 4)^\top$. When accompanied by the initial condition $\bar{y}(0) = (1, 1)^\top$, the analytical solution is $\bar{y}(t) = \frac{1}{7}(1, -1)^\top e^{-t} + \frac{2}{7}(3, 4)^\top e^{6t}$.

We can also solve for $\bar{y}(t)$ using our Bézier curve method. The idea is to represent it as a Bézier curve $\mathbf{B}(t)$ with 2-dimensional control points. If $\mathbf{B}(t)$ is chosen to be of degree n , it has $n + 1$ to-be-determined control points $P = \{\bar{p}_0, \bar{p}_1 \dots \bar{p}_n\}$. Using the test points $t_1, t_2 \dots t_M$, we formulate the following QPP:

$$\begin{aligned}
& \underset{\bar{p}_0, \bar{p}_1, \dots, \bar{p}_n}{\text{Minimize}} && \sum_{i=1}^M \epsilon_i^2 \\
& \text{s.t.} && \mathbf{B}(0) = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\
& \forall 1 \leq i \leq M : && \begin{pmatrix} -\epsilon_i \\ -\epsilon_i \end{pmatrix} \leq \mathbf{B}'(t_i) - A\mathbf{B}(t_i) \leq \begin{pmatrix} \epsilon_i \\ \epsilon_i \end{pmatrix}.
\end{aligned} \tag{12}$$

The constraints in this problem are linear since $\mathbf{B}(t)$ and $\mathbf{B}'(t)$ yield linear combinations of the to-be-determined control points when evaluated at a specific t .

Figure 6 shows the solutions generated by the Bézier curve method. The solutions approximate the analytical solution very well; and the accuracy increases with increasing n and M .

5 Solving PDEs

In this section, we apply our methods to PDEs. In such cases, we have multiple independent variables; and the required function is a surface in high-dimensional space. The differential equations specifying the characteristics of the required function can involve its partial derivatives. We generalize our Bézier *curve* method to the Bézier *surface* method. For illustration, we focus our discussion on solving PDEs for a function $f(t, u)$ on two independent variables t and u .

A k -dimensional Bézier surface $\mathbf{B}(t, u)$ of degrees $n_t \times n_u$ is characterized by the k -dimensional control points $\mathbf{p}_{i,j}$, for $0 \leq i \leq n_t$ and $0 \leq j \leq n_u$. It is given by

$$\mathbf{B}(t, u) = \sum_{i=0}^{n_t} \sum_{j=0}^{n_u} \mathbf{p}_{i,j} B_{i,n_t}(t) B_{j,n_u}(u), \tag{13}$$

where $B_{i,n_t}(t)$ and $B_{j,n_u}(u)$ are the Bernstein basis polynomials. The partial derivatives of $\mathbf{B}(t, u)$ are given by

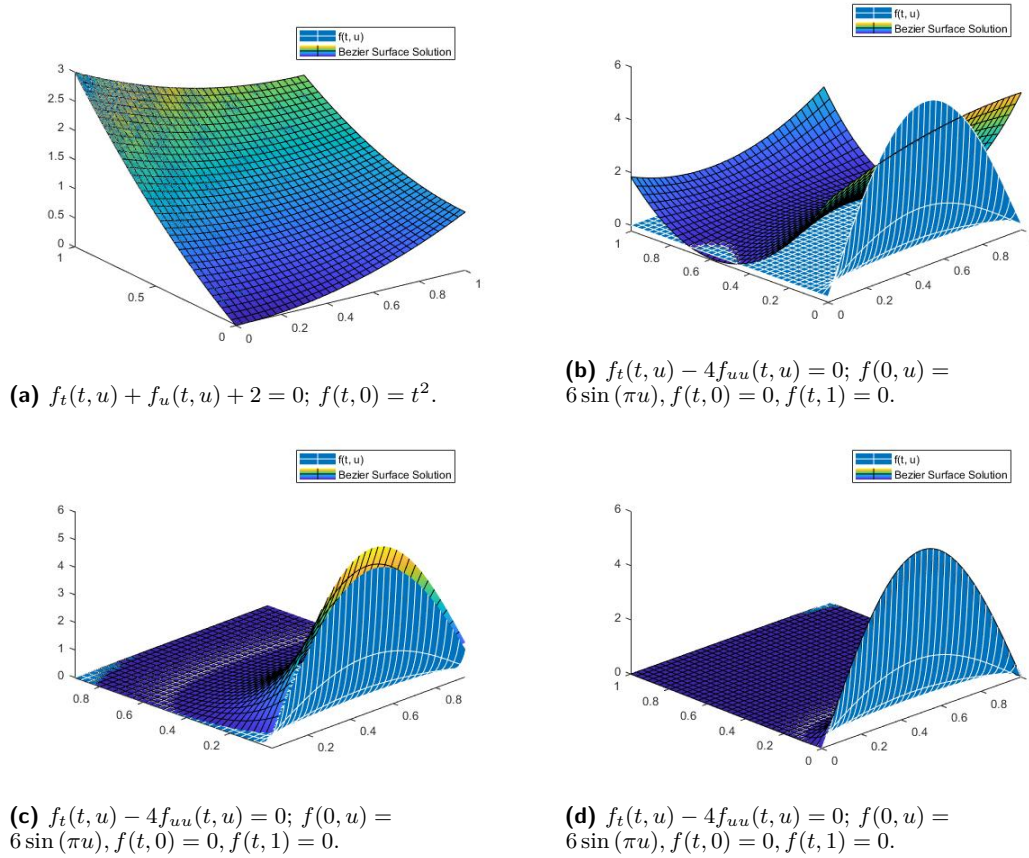
$$\begin{aligned}
\frac{\partial \mathbf{B}(t, u)}{\partial t} &= n_t \sum_{i=0}^{n_t-1} \sum_{j=0}^{n_u} (\mathbf{p}_{i+1,j} - \mathbf{p}_{i,j}) B_{i,n_t-1}(t) B_{j,n_u}(u) \\
\frac{\partial \mathbf{B}(t, u)}{\partial u} &= n_u \sum_{j=0}^{n_u-1} \sum_{i=0}^{n_t} (\mathbf{p}_{i,j+1} - \mathbf{p}_{i,j}) B_{i,n_t}(t) B_{j,n_u-1}(u)
\end{aligned}$$

Bézier surfaces have attractive mathematical properties equivalent to those of Bézier curves [17]. These include their ability to approximate any surface with a sufficient number of control points, being closed under the operations of differentiation, and being entirely within the convex hull of their control points. For a function $f(t, u)$ represented as a Bézier surface $\mathbf{B}(t, u)$ of degrees $n_t \times n_u$, there are $(n_t + 1) \times (n_u + 1)$ to-be-determined control points. Evaluating $\mathbf{B}(t, u)$ at a specific test point (t, u) yields a linear combination of these control points that can be easily incorporated into the formulation of a QPP.

Consider the following example PDE:

$$\begin{aligned}
f_t(t, u) + f_u(t, u) + 2 &= 0 \\
f(t, 0) &= t^2,
\end{aligned} \tag{14}$$

where $f_t(t, u)$ and $f_u(t, u)$ denote the partial derivatives of $f(t, u)$ with respect to t and u , respectively, and $f(t, 0) = t^2$ is a boundary condition.

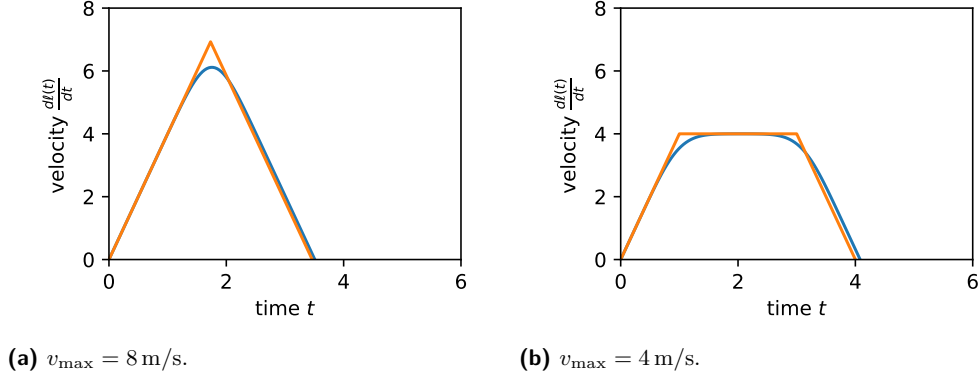


■ **Figure 7** Shows the solutions of some PDEs. The solutions generated by the Bézier surface method are good approximations to the analytical solutions. The quality of the solutions increases with the number of test points and the degrees used in the Bézier surface. In (a), the analytical solution is $f(t, u) = 2u + (t - u)^2$. In (b)-(d), the analytical solution is $f(t, u) = 6 \sin(\pi u)e^{-4\pi^2 t}$. In (a), 4×4 control points and 17 test points were used, resulting in a running time of 2.36 s. In (b), 3×3 control points and 10 test points were used, resulting in a running time of 1.85 s. In (c), 6×6 control points and 37 test points were used, resulting in a running time of 2.33 s. In (d), 11×11 control points and 122 test points were used, resulting in a running time of 20.45 s. In (a) and (d), the Bézier surface solution is an exact match to the analytical solution.

Using the test points $(t_1, u_1), (t_2, u_2) \dots (t_M, u_M)$, we formulate the following QPP:

$$\begin{aligned}
 & \underset{p_{i,j}: 0 \leq i \leq n_t, 0 \leq j \leq n_u}{\text{Minimize}} \quad \sum_{l=1}^M (\epsilon_l^2 + \varepsilon_l^2) \quad \text{s.t.} \\
 & \forall 1 \leq l \leq M: \quad -\epsilon_l \leq \frac{\partial \mathbf{B}(t, u)}{\partial t} + \frac{\partial \mathbf{B}(t, u)}{\partial u} + 2 \leq \epsilon_l \\
 & \forall 1 \leq l \leq M: \quad -\varepsilon_l \leq \mathbf{B}(t_l, 0) - t_l^2 \leq \varepsilon_l.
 \end{aligned} \tag{15}$$

Our test points are chosen from $[0, 1] \times [0, 1]$. Unlike the initial conditions in ODEs that were imposed as hard constraints at specific points, the boundary conditions in PDEs typically involve entire subspaces. For example, the boundary condition $f(t, 0) = t^2$ involves all $t \in [0, 1]$. While we can express this condition as a hard constraint equating two polynomials, it risks posing an over-constrained problem. Therefore, we include it as a soft constraint in the objective function, with each test point contributing a term to it.



■ **Figure 8** Shows the result of applying our Bézier curve method to the motion profile problem. The orange curves are the optimal velocity profiles derived from physical interpretation. The blue curves are close approximations produced by our method. In both cases, 40 control points were used for the approximation. In both (a) and (b), $L = 12 \text{ m}$, $a_{\max} = 4 \text{ m/s}^2$ and $a_{\min} = -4 \text{ m/s}^2$. In (a), $v_{\max} = 8 \text{ m/s}$, and in (b), $v_{\max} = 4 \text{ m/s}$.

Figure 7(a) shows the result of our method for the above PDE with $n_t = 3$, $n_u = 3$ and $M = 17$. The result is an exact match to the known analytical solution $f(t, u) = 2u + (t - u)^2$.

We now consider a popular Heat Equation widely used in physics [18]. With Dirichlet boundary conditions, the PDE is as follows:

$$\begin{aligned} f_t(t, u) - 4f_{uu}(t, u) &= 0 \\ f(0, u) &= 6 \sin(\pi u) \\ f(t, 0) &= 0, f(t, 1) = 0. \end{aligned} \tag{16}$$

Here, $f_{uu}(t, u)$ refers to the second partial derivative $\frac{\partial^2 f(t, u)}{\partial u^2}$. The analytical solution is given by $f(t, u) = 6 \sin(\pi u)e^{-4\pi^2 t}$.

Figures 7(b)-(d) show the results of our method for the Heat Equation with different values of n_t , n_u and M . The accuracy improves with increasing degrees and number of test points. In fact, an exact match to the analytical solution is achieved relatively quickly, as shown in Figure 7(d).

6 Differential Programming with Inequalities

In the foregoing sections, we demonstrated the viability of our approach on various kinds of ODEs and PDEs. As already outlined in the Introduction, we conducted this feasibility study in anticipation of reaping the many benefits of our approach compared to existing methods. In this section, we show one such benefit in allowing the use of inequalities.

Inequalities and differential operators are commonplace in robotics, physics, and hybrid systems, among many other areas of science and engineering. For example, in robotics, a robot might have a maximum acceleration or deceleration capability that is posed as an inequality involving the second derivative of its motion profile. Existing analytical techniques are not capable of handling inequalities; and existing numerical techniques do not produce an analytical solution that may be required for downstream tasks. However, our Bézier curve method is viable in such situations.

Consider the following simple motivating example. Suppose a robot is required to travel a distance L in a straight line between two points A and B . Suppose it is required to start at A and end at B with 0 velocities; and suppose it has maximum velocity $v_{\max} \geq 0$, maximum acceleration $a_{\max} \geq 0$ and minimum acceleration $a_{\min} \leq 0$. The goal is to minimize the traversal time T . Intuitively, the optimal solution is to start with maximum acceleration a_{\max} and stop with maximum deceleration $|a_{\min}|$. In between, the robot should cap off at the maximum velocity v_{\max} . The two possible scenarios are illustrated in Figure 8.

Stated purely mathematically, the differential programming problem involves inequalities and is as follows:

$$\begin{aligned} \text{Find } \ell(t) : [0, T] &\rightarrow \mathbb{R} \text{ and minimum } T \quad \text{s.t.} \\ \forall t : \ell'(t) &\leq v_{\max} \\ \forall t : a_{\min} &\leq \ell''(t) \leq a_{\max} \\ \ell(0) &= 0, \ell(T) = L \\ \ell'(0) &= 0, \ell'(T) = 0. \end{aligned} \tag{17}$$

As such, solving this mathematical problem without the physical interpretation is not straightforward even from the perspective of techniques available in calculus. This is primarily because of the inequalities imposed on the derivatives of continuous functions.

In contrast, our Bézier curve method solves this problem efficiently since inequalities are naturally allowed in OR solvers. Figure 8 shows the Bézier curve solutions for $\ell(t)$, the distance function that represents the distance covered at time t starting from A , for the two possible scenarios. (See [19] for more details on this approach to solve the motion profile problem and its generalization to the multi-robot scenario.)

We also note that our Bézier curve method is not just any polynomial-fitting method. General polynomial-fitting methods cannot enforce global conditions on a function since they are required to hold for *all* t . In our method, the convex hull property of Bézier curves is invoked to ensure that satisfying inequalities at only the control points entails that they are also globally satisfied.

7 Discussion

There are many anticipated benefits of our approach since it casts differential operators in the language of OR, and consequently, in the language of search. Many optimization problems in science and engineering that may not be directly amenable to analytical methods can instead be solved programmatically using OR solvers. In turn, powerful OR solvers like Gurobi are scalable to millions of variables. They also employ efficient parallelization techniques. Moreover, since OR is already being studied in relation to constraint programming (CP) and artificial intelligence (AI), our framework paves the way for combining the strengths of variational techniques used in calculus, primal-dual techniques used in OR, constraint propagation techniques used in CP, and heuristic search techniques used in AI.

Many problems in the real world can also benefit from rendering differential operators in the language of search. In addition to scalability and reasoning with inequalities, this reformulation allows us to introduce extra decision variables. Testing and verification of complex systems involving ODEs/PDEs can be done via highly scalable search-based methods instead of prohibitively expensive simulation-based methods. Optimization problems in computational physics, e.g., how to optimally place a set of teflon dielectric cylinders to focus electromagnetic power, can be solved using search-based methods after rendering the PDEs of electromagnetism in the language of OR.

8 Conclusions and Future Work

In this paper, we presented an OR-based approach for differential programming to address the drawbacks of existing analytical and numerical methods. Analytical methods mostly remain an art and are largely insufficient for complex systems. Numerical approximation methods do not yield a viable analytical form of the solution that is often required for downstream tasks. Our main idea was to represent entire functions as Bézier curves or Bézier surfaces with to-be-determined control points. The ODEs/PDEs as well as their boundary conditions can then be reformulated as constraints on these control points. In many cases, we showed that this reformulation yields QPPs that can be solved efficiently. We also demonstrated the use of our approach in differential programming with inequalities.

More generally, our work facilitates search-based methods for solving problems that involve differential operators and sets the stage for combining the strengths of variational methods used in calculus and search-based pruning methods used in OR, CP and AI. There are many avenues of future work based on the foregoing discussions. We are also interested in the idea of representing local regions of functions using separate Bézier curves/surfaces and “stitching” them together under conditions of continuity to achieve more efficiency.

References

- 1 Ji-wung Choi, Renwick Curry, and Gabriel Elkaim. Path planning based on bézier curve for autonomous ground vehicles. In *Advances in Electrical and Electronics Engineering – IAENG Special Edition of the World Congress on Engineering and Computer Science*, pages 158–166, 2008.
- 2 Ken Dill and Sarina Bromberg. *Molecular Driving Forces: Statistical Thermodynamics in Biology, Chemistry, Physics, and Nanoscience*. Garland Science, 2012.
- 3 Robert Eymard, Thierry Gallouët, and Raphaële Herbin. Finite volume methods. *Handbook of Numerical Analysis*, 7:713–1018, 2000.
- 4 N. Fallah and N. Nikraftar. Meshless finite volume method for the analysis of fracture problems in orthotropic media. *Engineering Fracture Mechanics*, 204:46–62, 2018.
- 5 Gerald Farin. *Curves and Surfaces for Computer-Aided Geometric Design: A Practical Guide*. Elsevier, 2014.
- 6 Torkel Glad and Lennart Ljung. *Control Theory*. CRC Press, 2018.
- 7 Antonio Huerta, Ted Belytschko, Sonia Fernández-Méndez, Timon Rabczuk, Xiaoying Zhuang, and Marino Arroyo. Meshfree methods. *Encyclopedia of Computational Mechanics (Second Edition)*, pages 1–38, 2018.
- 8 Thomas Hughes, Giancarlo Sangalli, and Mattia Tani. *Isogeometric Analysis: Mathematical and Implementational Aspects, with Applications*. Lecture Notes in Mathematics Book Series (LNM, volume 2219), 2018.
- 9 I. Krasilshchik and A. Vinogradov. Nonlocal trends in the geometry of differential equations: Symmetries, conservation laws, and bäcklund transformations. In *Symmetries of Partial Differential Equations*, pages 161–209. Springer, 1989.
- 10 Tadeusz Liszka and Janusz Orkisz. The finite difference method at arbitrary irregular grids and its application in applied mechanics. *Computers & Structures*, 11(1-2):83–95, 1980.
- 11 George Lorentz. *Bernstein Polynomials*. American Mathematical Soc., 1986.
- 12 Michael Mortenson. *Mathematics for Computer Graphics Applications*. Industrial Press Inc., 1999.
- 13 Lev Ovsiannikov. *Group Analysis of Differential Equations*. Academic Press, 2014.
- 14 Michael Renardy and Robert Rogers. *An Introduction to Partial Differential Equations*. Springer, 2006.

- 15 Alberto Rojo and Anthony Bloch. *The Principle of Least Action: History and Physics*. Cambridge University Press, 2018.
- 16 Sarah Tang and Vijay Kumar. Safe and complete trajectory generation for robot teams with higher-order dynamics. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1894–1901, 2016.
- 17 Lianqiang Yang and Xiao-Ming Zeng. Bézier curves and surfaces with shape parameters. *International Journal of Computer Mathematics*, 86:1253–1263, 2009.
- 18 Hugh Young, Roger Freedman, and Albert Ford. *Sears and Zemansky's University Physics*. Pearson Addison-Wesley, 2006.
- 19 Han Zhang, Neelesh Tiruvilumala, Sven Koenig, and T. K. Satish Kumar. Temporal reasoning with kinodynamic networks. In *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling*, 2021.
- 20 Olgierd Zienkiewicz, Robert Taylor, Perumal Nithiarasu, and J. Zhu. *The Finite Element Method*. McGraw-Hill London, 1977.

Learning Max-CSPs via Active Constraint Acquisition

Dimosthenis C. Tsouros ✉

Dept. of Electrical & Computer Engineering, University of Western Macedonia, Kozani, Greece

Kostas Stergiou ✉

Dept. of Electrical & Computer Engineering, University of Western Macedonia, Kozani, Greece

Abstract

Constraint acquisition can assist non-expert users to model their problems as constraint networks. In active constraint acquisition, this is achieved through an interaction between the learner, who posts examples, and the user who classifies them as solutions or not. Although there has been recent progress in active constraint acquisition, the focus has only been on learning satisfaction problems with hard constraints. In this paper, we deal with the problem of learning soft constraints in optimization problems via active constraint acquisition, specifically in the context of the Max-CSP. Towards this, we first introduce a new type of queries in the context of constraint acquisition, namely *partial preference queries*, and then we present a novel algorithm for learning soft constraints in Max-CSPs, using such queries. We also give some experimental results.

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

Keywords and phrases Constraint acquisition, modeling, learning

Digital Object Identifier 10.4230/LIPIcs.CP.2021.54

1 Introduction

Constraint programming (CP) is a powerful paradigm for solving combinatorial problems, with successful applications in various domains. The basic assumption in CP is that the user models the problem and a solver is then used to solve it. One of the major challenges that CP has to deal with is that of efficiently obtaining a good model of a real problem without relying on experts [21, 33, 23, 22]. As a result, automated modeling and constraint learning technologies attract a lot of attention nowadays, and a number of approaches based on Machine Learning have been developed [31, 18, 17].

An area of research in CP towards this direction is that of *constraint acquisition* where the model of a constraint problem is acquired (i.e. learned) using examples of solutions and non-solutions [8, 7, 2, 32]. Constraint Acquisition can be *passive* or *active*. In *passive* acquisition, examples of solutions and non-solutions are provided by the user and based on these examples, the goal is to learn a set of constraints that correctly classifies the given examples [3, 5, 29, 2, 8]. In *active* or *interactive* acquisition the system interacts with an oracle, e.g. a human user, while acquiring the constraint network [24, 6, 37, 8]. State-of-the-art active constraint acquisition systems like QuAcq [4], MQuAcq [42] and MQuAcq-2 [41] use the version space learning paradigm [30], extended for learning constraint networks. They learn the target constraint network by proposing examples to the user to classify them as solutions or not [6, 8, 37]. These questions are called membership queries [1].

Although constraint learning has focused on satisfaction problems, soft constraints, within constraint optimization frameworks such as (weighted) Max-CSP, have also been considered [15, 43, 18, 12] as part of the wider literature on learning preferences [19, 36].



© Dimosthenis C. Tsouros and Kostas Stergiou;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 54; pp. 54:1–54:18



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In [34, 9] ML techniques are exploited in order to infer constraint preferences from given solution ratings. Rossi and Sperduti [35] extend these methods so that the scoring function is estimated via an interactive process where high-scoring assignments are posted as queries to the user, who then ranks them according to her preferences. Campigotto et al. [14] consider combinatorial utility functions expressed as weighted combinations of terms.

Techniques for computing minimax optimal decisions have also been developed [10, 11, 44]. [27] uses weighted first-order logical theories to represent constrained optimization problems. Dragone et al. [20] exploits comparison queries in a context where preferences are modeled by individual variables. In [28] MAX-SAT models that can be probably approximately correct (PAC) are learned for combinatorial optimization. Preference elicitation methods for Incomplete Soft Constraint Problems (ISCPs) [25] and Distributed Constraint Optimization Problems (DCOPs) [39, 38] have also been studied.

More closely related to the framework of constraint acquisition are the works of Vu and O’Sullivan [45, 46, 47]. However, all these works concern passive constraint acquisition. In this paper, we deal with the problem of learning Max-CSPs via active constraint acquisition. To the best of our knowledge, there is no study on learning soft constraints in this context. We first introduce a new type of queries in this context, namely *(partial) preference queries*, inspired by works on preference elicitation [16]. Such a query posts two examples to the user and asks her to specify if either of them is preferable or if she is indifferent between the two. We then describe an algorithm that, driven by the user’s replies to preference queries, is able to learn all the soft constraints appearing in a Max-CSP. We highlight the differences between learning soft constraints within our proposed framework and standard constraint acquisition of hard constraints and give preliminary experimental results.

2 Background

The *vocabulary* (X, D) is the common knowledge shared by the user and the system. It is a finite set of n variables $X = \{x_1, \dots, x_n\}$ and a set of domains $D = \{D(x_1), \dots, D(x_n)\}$, where $D(x_i) \subset \mathbb{Z}$ is the set of values for x_i .

A *constraint* c is a pair $(\text{rel}(c), \text{var}(c))$, where $\text{var}(c) \subseteq X$ is the *scope* of the constraint, while $\text{rel}(c)$ is a relation between the variables in $\text{var}(c)$ that specifies which of their assignments satisfy c . $|\text{var}(c)|$ is called the *arity* of the constraint. A *constraint network* is a set C of constraints on the vocabulary (X, D) . A constraint network that contains at most one constraint on each subset of variables (i.e. for each scope) is called *normalized*. Following the literature on constraint acquisition, we will assume that the target constraint network is normalized.

An example e_Y is an assignment on a set of variables $Y \subseteq X$ and it belongs to $D^Y = \prod_{x_i \in Y} D(x_i)$. If $Y = X$, the example e is called a *complete* example. Otherwise, it is called a *partial* example. An example e_Y is rejected (or accepted) by a constraint c iff $\text{var}(c) \subseteq Y$ and the projection $e_{\text{var}(c)}$ of e_Y is not in (or is in) $\text{rel}(c)$. A complete assignment that is accepted by all the constraints in C is a solution of C . $\text{sol}(C)$ denotes the set of solutions of C . An assignment e_Y is a partial solution of C iff it is not rejected by any constraint in C . Note that such a partial assignment is not necessarily part of a complete solution. $\kappa_C(e_Y)$ denotes the set of constraints in C that reject e_Y , while $\lambda_C(e_Y)$ denotes the set of constraints in C that satisfy e_Y .

Besides the vocabulary, the learner is given a *language* Γ consisting of *bounded arity* constraints. The *constraint bias* B is a set of constraints on the vocabulary (X, D) , built using the constraint language Γ . The bias is the set of all candidate constraints from which the system can learn the target constraint network.

In ML, the classification question asking the user to determine if an example e_X is a solution to the problem or not, is called a *membership query* $ASK(e)$. The answer to such a query is positive if e is a solution and negative otherwise. A *partial membership query* $ASK(e_Y)$, with $Y \subset X$, asks the user to determine if $e_Y \in D^Y$ is a partial solution or not. Following the literature on constraint acquisition, we assume that all queries are answered correctly by the user.

In active constraint acquisition, the system iteratively generates a set E of complete or partial examples, which are labelled by the user as positive or negative. A constraint network C agrees with E if C accepts all examples in E labelled as positive and rejects those labelled as negative. The acquisition process has *converged* on the learned network of constraints $C_L \subseteq B$ iff C_L agrees with E and for every other network $C \subseteq B$ that agrees with E , we have $sol(C) = sol(C_L)$.

2.1 Max-CSP

A *Max-CSP* is a quadruple $P = (X, D, Ch, Cs)$, with X being the set of variables, D the set of domains, Ch being the set of *hard* constraints that have to be satisfied mandatorily, and Cs being the set of *soft* constraints whose satisfaction should be maximized, called *soft* constraints. The optimal solution to a Max-CSP maximizes the number of satisfied soft constraints, while satisfying all the hard constraints. In a *weighted Max-CSP* each soft constraint $c_i \in Cs$ is associated with a positive real value (a weight) w_i and the optimal solution maximizes the total sum of the satisfied constraints' weights.

As in learning a standard CSP, it is important to be able to determine whether the version space has converged, or not. If this is indeed the case, the learning system will stop posting queries as the user has an exact characterization of her target problem. But convergence must be defined in a different way compared to the standard case. We now define the target constraint network and the convergence problem in the context of constraint acquisition of soft constraints in Max-CSPs.

► **Definition 1.** The *target soft constraint network* Cs_T is the constraint network that correctly states the preferences of the user in the problem she has in mind.

► **Definition 2.** Given a bias B being able to represent the target soft constraint network Cs_T , the system has *converged* to Cs_T iff $\forall c \in B, Ch_L \models c \vee \exists c' \in Cs_L \mid c' \models c$ w.r.t. Ch_L , with Ch_L and Cs_L being the learned networks of hard and soft constraints respectively.

Hence, the system converges to the target network Cs_T when all the constraints that are still in the bias B are implied by Ch_L or by a constraint we have already learned w.r.t. Ch_L .

3 Partial Preference Queries

In active constraint acquisition, the interaction between the learner and the user is established via membership queries. This process can be used while learning Max-CSPs to acquire any hard constraints that may be present in the problem, but membership queries cannot be used to acquire soft constraints, as such constraints are allowed to be violated in both solutions and non-solutions.

As a result, several other types of queries have been considered in preference learning. For example, the user can be asked to associate a precise desired value to each presented solution [35]. As another example, a comparison query posts two examples to the user and asks her to state which of them she prefers [16, 26]. To be precise, a comparison query posts two complete assignments e_X and e'_X to the user, and the possible answers to such a query are:

1. $e_X \succ e'_X$: the user prefers e_X to e'_X ,
2. $e_X \prec e'_X$: the user prefers e'_X to e_X ,
3. $e_X \sim e'_X$: the user is indifferent between e_X and e'_X .

Such queries are easier for the user to answer compared to other types used in preference elicitation. We now introduce *partial preference queries* as a variant of comparison queries. Specifically, in a partial preference query, denoted as $\text{PrefAsk}()$, we can have the following cases regarding the two examples included in the query:

1. Both examples are (partial) assignments e_Y, e'_Y over the same set of variables $Y \subseteq X$. In this case, the query is similar to a comparison query, generalized so that the assignments can be partial.
2. One of the examples is e_Y , with $Y \subseteq X$, and the other one is $e_{Y'}$, with $Y' \subset Y$. That is, the second example is a projection of the first one on some of its variables, which means that both examples share the same assignment in the variables in Y' , while the first example includes additional variable assignments (the variables in $Y \setminus Y'$). Hence, the answer of the user in this case can either be $e_Y \sim e_{Y'}$ or $e_Y \succ e_{Y'}$.

Let us now demonstrate the use of preference queries through a typical scenario from the literature [14]. Consider a house sales system suggesting candidate houses according to their characteristics. Assume that we have several variables, including the price of the house, its total area, whether it has a garden, whether it has a parking spot, the construction year, etc. Now assume that the preferences of the user are: 1. to have a parking spot, 2. the total area of the house to be $\geq 100m^2$. Now consider a partial preference query consisting of the following examples:

- House #1: Construction year = 2000, parking spot = “yes”, garden = “no”
- House #2: Construction year = 2004, parking spot = “no”, garden = “no”

In this case the examples in the query are both partial assignments on the same variables, and the user would prefer the first one (i.e. House #1), because it satisfies the requirement to have a parking spot. Now consider the following partial preference query:

- House #1: Construction year = 2000, parking spot = “yes”, garden = “no”
- House #2: Construction year = 2000, garden = “no”

This is a case where the second example is a projection on the assignment of the first one. Again, the user would prefer House #1, because it satisfies the requirement to have a parking spot. Hence, it “offers greater satisfaction” of the preferences compared to House #2.

Now assume we have:

- House #3: Construction year = 2000, parking spot = “no”, garden = “no”

If the system asks the user to compare House #2 and House #3 the user would answer that she is indifferent, as no additional requirement is satisfied by House #3. This is due to the fact that this type of preference queries is asking the user to state if the additional information offered by House #3 helps satisfy the preferences to a greater degree, and is not a comparison between 2 different examples (i.e. Houses).

A query posted to the user must give the system more information that it already has. So now we define the notion of *informative* queries.

► **Definition 3.** A (partial) preference query q is called *irredundant* (or *informative*) iff the answer of the user to q is not predictable. Otherwise, it is called *redundant*. The answer of the user to a query is predictable when the satisfied constraints from Cs_L and B by the two examples imply that $\lambda_{Cs_T}(e) \supset \lambda_{Cs_T}(e')$ or $\lambda_{Cs_T}(e) = \lambda_{Cs_T}(e')$.

4 Learning Soft Constraints

In this section, we present our proposed approach for learning Max-CSPs. We first detail the differences between the acquisition of soft constraints within our framework and standard active constraint acquisition, and then present our algorithm. We then present our proposed algorithm, PrefAcq, in detail.

4.1 Differences with Constraint Acquisition of hard constraints

In our proposed method the entire network is learned in two separate steps:

1. The hard constraints (if any) are learned via a standard constraint acquisition algorithm. Only membership queries are used in this step, while the soft constraints do not affect the answers of the user.
2. The soft constraints representing the preferences of the user are learned via our proposed algorithm. Only preference queries are used in this step, but as we explain, the examples generated must satisfy the already learned hard constraints.

Although in the context of standard constraint acquisition, learning hard constraints is well defined, some things differ when acquiring soft constraints. Let us first recall how active constraint acquisition algorithms operate. They typically comply with the following generic procedure:

1. Generate an example e_Y in D^Y and post it as a query to the user.
 - a. If the answer is positive, update the version space, removing from the bias B the constraints rejecting the example.
 - b. If the answer is negative, search for one or more constraints of C_T that reject the example e_Y , via partial membership queries.
2. If not converged, return to step 1.

In more detail, once a generated example e_Y is classified as negative, the system discovers the scope of one of the violated constraints, as follows. It successively decomposes e_Y to a simpler problem by removing entire blocks of variables from the example while posting partial queries to the user. If after the removal of some variables the answer of the user to the partial query posted is “yes”, then it has discovered that the removed block contains at least one variable from the scope of a violated constraint. Then the acquisition system focuses on this block. When, after repeatedly removing variables, the size of the considered block is 1, then this variable surely belongs to the scope of a violated constraint. A logarithmic complexity in terms of the number of queries posted to the user is achieved by splitting. In each decomposition step the set of variables is approximately split in half.

Our proposed approach uses a similar technique. We exploit the 2nd type of partial preference queries described above to locate the scope of satisfied soft constraints in a generated example. That is, we repeatedly post a query comparing an example e_Y with its projection on a subset of variables $Y' \subset Y$. The 1st type of partial preference queries is used to find the specific relation of the constraint, after the scope has been located.

Let us now detail the differences between standard learning of hard constraints and learning of soft constraints.

4.1.1 Violation vs. satisfaction of constraints

A main difference is that in standard constraint acquisition, when trying to find a hard constraint via membership queries, it is the violation of constraints that drives the search. This is because the violation of a constraint results in a negative answer by the user, which

forces the system to continue searching. On the other hand, in a preference query, it is the satisfaction of constraints that drives the search process. This is because the preference of one example over another means that more constraints are satisfied in the former compared to the latter, which will force the system to continue searching for these constraints.

4.1.2 Information derived from user answers

In standard constraint acquisition, the procedure to find the scope of one or more violated constraints exploits the fact that the information that is derived from the answer to a membership query $ASK(e_Y)$ concerns the variables in Y , and only them. That is, the answer will inform us about the existence or not of a violated constraint c with $var(c) \subset Y$. In a preference query, when comparing an example e_Y to its projection on a subset of variables $Y' \subset Y$, the answer of the user gives information about the variables in $Y \setminus Y'$. That is, the answer will inform us about the existence or not of a satisfied soft constraint c with $var(c) \subset Y \wedge \exists x \in var(c) \mid x \in Y \setminus Y'$. Hence, if the answer is that the user is indifferent between the examples, any constraint c' with $var(c') \subset Y \wedge \exists x \in var(c') \mid x \in Y \setminus Y'$ has to be removed from B because it certainly does not belong to Cs_T (if it did belong then the user would have preferred e_Y).

4.1.3 Top-down vs. bottom-up

Because of the above, another important difference lies in the algorithmic approach. Standard constraint acquisition algorithms follow a top-down procedure when searching for constraints to learn. They post membership queries to the user while successively decomposing the initial example. As we will explain in the next section, our algorithm for Max-CSPs also performs a top-down decomposition of the initial query, but crucially, no queries are posted while this decomposition takes place. Once this process is finished, having decomposed the query as much as possible, the algorithm continues in a bottom-up fashion, with preference queries being posted to guide the search for satisfied soft constraints.

Example 1 shows the series of queries posted by our method to locate the scope of a constraint.

► **Example 1.** Assume that the vocabulary (X, D) given to the system is $X = \{x_1, \dots, x_8\}$ and $D = \{D(x_1), \dots, D(x_8)\}$ with $D(x_i) = \{1, \dots, 8\}$, the target network of soft constraints Cs_T is the set $\{c_{37}, c_{38}\}$ and $B = \{c_{ij} \mid 1 \leq i < j \leq 8\}$, with $|B| = 28$. Also, assume that we have a complete example which satisfies all the constraints in B . Table 1 shows the preference queries posted to the user until the scope of one of the two constraints in Cs_T has been found.

In a process explained below, our method recursively creates sub-examples by splitting the example, approximately in half, until it creates a sub-example with only one variable assignment. Assuming that this sub-example is $e_{\{x_1\}}$, we will now search for a constraint that is satisfied by $e_{\{x_1\}}$. As no constraint c exists in B with $var(c) = \{x_1\}$, we will go back to search in $e_{\{x_1, x_2\}}$, by posting the query $\text{PrefAsk}(e_{\{x_1\}}, e_{\{x_1, x_2\}})$ to the user. As the user will answer that she is indifferent between the two examples (because none of the target constraints is satisfied by them), we will first remove c_{12} from B , as it is definitely not in Cs_T , and then will continue adding variables to the examples and posting queries. The user's answer to the third query will be that the second example is preferable to the first (because both target constraints are satisfied by $e_{\{x_1, \dots, x_8\}}$). Hence, we find that at least one variable of the scope of a satisfied constraint from Cs_T is in $Y \setminus Y'$, i.e. in $\{x_5, \dots, x_8\}$.

■ **Table 1** Searching for a soft constraint via preference queries, in Example 1.

#	Preference query	answer	information we get
1.	$e_{\{x_1\}} \text{ vs } e_{\{x_1, x_2\}}$	\sim	no satisfied constraints, c_{12} removed from B
2.	$e_{\{x_1, x_2\}} \text{ vs } e_{\{x_1, \dots, x_4\}}$	\sim	no satisfied constraints, $c_{13}, c_{14}, c_{23}, c_{24}$ removed from B
3.	$e_{\{x_1, \dots, x_4\}} \text{ vs } e_{\{x_1, \dots, x_8\}}$	\prec	at least one satisfied constraint with $\text{var}(c) \subset \{x_1, \dots, x_8\} \wedge \exists x \in \text{var}(c) \mid x \in \{x_5, \dots, x_8\}$
4.	$e_{\{x_1, \dots, x_4\}} \text{ vs } e_{\{x_1, \dots, x_5\}}$	\sim	no satisfied constraints here, $c_{15}, c_{25}, c_{35}, c_{45}$ removed from B
5.	$e_{\{x_1, \dots, x_5\}} \text{ vs } e_{\{x_1, \dots, x_6\}}$	\sim	no satisfied constraints, $c_{16}, c_{26}, c_{36}, c_{46}, c_{56}$ removed from B
6.	$e_{\{x_1, \dots, x_6\}} \text{ vs } e_{\{x_1, \dots, x_8\}}$	\prec	at least one satisfied constraint with $\text{var}(c) \subset \{x_1, \dots, x_8\} \wedge \exists x \in \text{var}(c) \mid x \in \{x_7, x_8\}$
7.	$e_{\{x_1, \dots, x_6\}} \text{ vs } e_{\{x_1, \dots, x_7\}}$	\prec	$x_7 \in \text{var}(c)$
8.	$e_{\{x_7\}} \text{ vs } e_{\{x_1, x_7\}}$	\sim	no satisfied constraints, c_{17} removed from B
9.	$e_{\{x_1, x_7\}} \text{ vs } e_{\{x_1, \dots, x_3, x_7\}}$	\prec	at least one satisfied constraint with $\text{var}(c) \subset \{x_1, \dots, x_3, x_7\} \wedge \exists x \in \text{var}(c) \mid x \in \{x_2, x_3\}$
10.	$e_{\{x_1, x_7\}} \text{ vs } e_{\{x_1, x_2, x_7\}}$	\sim	no satisfied constraints, c_{27} removed from B
11.	$e_{\{x_1, x_2, x_7\}} \text{ vs } e_{\{x_1, \dots, x_3, x_7\}}$	\prec	$x_3 \in \text{var}(c)$

Then, trying to discover the complete scope, again we will decompose the projection of the example on this discovered set of variables $\{x_5, \dots, x_8\}$ until we reach a sub-example with the fewest variables possible. However, in each query posted now, both examples will include the variables that have already been searched, because one (or more) variable(s) of the sought scope may be among them. For instance, in the 4th query both the examples include the assignments of variables $\{x_1, \dots, x_4\}$ in which we have already searched. But although we know that there is no satisfied constraint $c \in Cs_T$ with $\text{var}(c) \in \{x_1, \dots, x_4\}$, it is possible that one or more variables among $\{x_1, \dots, x_4\}$ participate in the sought scope (as is actually the case with both c_{37} and c_{38}).

In query #6 the set of variables to search in is narrowed to $\{x_7, x_8\}$. As we will explain, this query (and query #11) will not be actually posted because it is redundant. We only include it here to make the example easier to understand. With query #7, we will find that x_7 is in the scope of a constraint. Then we will start searching again, with the same reasoning as before, in order to find the remaining variables of the scope, knowing that they are in $\{x_1, \dots, x_6\}$.

During this process, each query includes, in both examples, the assignment of the variable in the sought scope that we have already found (i.e. the assignment of x_7). Including the assignment of x_7 means that the answer of the user to the query posted will now depend only on the presence or absence of the other variable of the scope in the two examples. Hence, if x_3 is present in one of the examples and absent from the other, the former example will be preferred. After a few queries we will find scope $\{x_3, x_7\}$ and then we will continue searching for more constraints.

4.2 Description of PrefAcq

PrefAcq (Algorithm 1) is a novel active learning algorithm for soft constraints. It starts by setting the learned network Cs_L equal to the empty set (line 1). Then, it iteratively generates examples (line 3) in which it will search for satisfied soft constraints via the function

SearchSC (line 5) until it detects convergence at line 4. Each example generated must be a solution to the problem, i.e. to satisfy all the hard constraints. Also, it must satisfy at least one of the candidate soft constraints in the version space, i.e. at least one constraint from B , so that the version space is reduced with each generated example.

■ **Algorithm 1** The PrefAcq Algorithm.

Input: Ch , B , X , D (Ch : The set of the hard constraints, B : the bias, X : the set of variables, D : the set of domains)

Output: Cs_L : a constraint network

```

1:  $Cs_L \leftarrow \emptyset$ ;
2: while true do
3:   Generate  $e$  in  $D^Y$ , with  $Y \subseteq X$ , accepted by  $Ch$  s.t.  $\lambda_B(e_Y) \neq \emptyset$ ;
4:   if  $e = \text{nil}$  then return " $Cs_L$  converged";
5:    $\text{SearchSC}(e, \emptyset, Y, \emptyset, \text{true})$ ;
```

Function *SearchSC* (Algorithm 2) is used to search for satisfied soft constraints in the example generated. It finds *all* the constraints from Cs_T that are satisfied in the example given. It recursively decomposes the example as much as possible, until a sub-example with the minimum number of variables (typically just one) is reached, as in Example 1.

Then *SearchSC* starts the bottom-up search for satisfied constraints that rewinds the recursive decomposition of the initial example, with more variables taken into account step by step. In this way, it exploits the information that can be derived via the preference queries, i.e. for a preference query $\text{PrefAsk}(e_Y, e_{Y'})$, with $Y' \subset Y$, the answer of the user will reveal if $\exists c \in Cs_T \mid \text{var}(c) \in Y \setminus Y'$. *SearchSC* starts posting queries when the example with minimum number of variables is reached and then goes bottom-up so that in any subsequent query we will already have derived all the information we can in $e_{Y'}$ before we search in $Y \setminus Y'$. For example, in Example 1, when query 3 is posted to the user, we already know that there are no satisfied constraints from Cs_T in $\{x_1, x_2, x_3, x_4\}$.

In more detail, *SearchSC* takes as input an example e , three sets of variables R , Y , S and a Boolean variable *ask_query*. In each call, S contains variables which we have found to be in the scope of a satisfied constraint, for which we seek the rest of the variables. The set Y is the one in which we will search for satisfied constraints. R contains the variables that *SearchSC* has already searched in previous calls. The Boolean variable *ask_query* is set to true if a query is needed to be posted and to false if the query may be redundant. True is returned if a constraint has been found and false otherwise. In the first call to *SearchSC* in PrefAcq, we have $R = S = \emptyset$. Also, Y is set to the assigned variables in the example generated and *ask_query* = true.

First, *SearchSC* initializes the boolean variable *found_flag* to false and the set Q to $R \cup S$ (lines 2-3), i.e. the variables in which we have already searched in previous recursive calls (R) and the variables we have found to be in the scope we seek (S). The set Q stores the variables that will be present in both the partial examples posted to be compared by the user. It is empty in the first call. If the projection of the example e on $Q \cup Y$ (i.e. the variables in which we have already searched and the ones we are searching in the current recursive call) does not satisfy more constraints from B than its projection on Q , then there is no point searching for a satisfied constraint with at least one variable of its scope in Y . Hence, false is returned in this case (line 4). Otherwise, at line 5 the set Y is split in two balanced parts, with $|Y_1| = \lfloor |Y|/2 \rfloor$ and then the function recursively calls itself with $Y = Y_1$, reducing the set of variables to be searched. Notice that when splitting Y at line 5 we ensure that if $|Y|$ is not even, Y_2 will contain one more variable than Y_1 . This is important because otherwise the algorithm would never terminate because of the recursive call at line 6.

Algorithm 2 SearchSC: Searching for soft constraints.

Input: e, R, Y, S, ask_query (e : the example, R, Y, S : sets of variables, ask_query : a boolean variable)

Output: $found_flag$: returns true if a constraint is found, false otherwise

```

1: function SearchSC( $e, R, Y, S, ask\_query$ )
2:    $found\_flag \leftarrow false$ ;
3:    $Q \leftarrow R \cup S$ ;
4:   if  $\lambda_B(e_Q) = \lambda_B(e_{Q \cup Y})$  then return  $false$ ;
5:   split  $Y$  into  $\langle Y_1, Y_2 \rangle$  such that  $|Y_1| = \lfloor |Y|/2 \rfloor$  and  $Y_2 = Y \setminus Y_1$ ;
6:   if  $|Y_1| > 0$  then  $found\_flag \leftarrow$  SearchSC( $e, R, Y_1, S, true$ );
7:   if  $\exists c \in \lambda_{Cs_L}(e_{Q \cup Y}) \mid \exists var(c) \cap Y_2 \neq \emptyset$  then
8:      $c \leftarrow$  pick random  $c \in \lambda_{Cs_L}(e_{Q \cup Y})$  s.t.  $var(c) \cap Y_2 \neq \emptyset$ ;
9:     for each  $x_i \in var(c)$  do
10:       $found\_flag \leftarrow$  SearchSC( $e, R, Y \setminus \{x_i\}, found\_flag \vee ask\_query$ )
11:     $\vee found\_flag$ ;
12:   return  $found\_flag$ ;
13:   if  $\lambda_B(e_{Q \cup Y_1}) = \lambda_B(e_{Q \cup Y})$  then return  $found\_flag$ ;
14:    $ask\_query \leftarrow (ask\_query \vee found\_flag)$ ;
15:   if  $ask\_query \wedge PrefAsk(e_{Q \cup Y}, e_{Q \cup Y_1}) = (e_{Q \cup Y} \sim e_{Q \cup Y_1})$  then
16:      $B \leftarrow B \setminus \lambda_B(e_{Q \cup Y})$ ;
17:     return  $found\_flag$ ;
18:   if  $|Y_2| > 1$  then SearchSC( $e, R \cup Y_1, Y_2, S, false$ );
19:   else
20:     if SearchSC( $e, \emptyset, R \cup Y_1, S \cup Y_2, true$ ) =  $false$  then
21:        $Cs_L \leftarrow Cs_L \cup FindSC(e, S \cup Y_2)$ ;
22:   return  $true$ ;
```

In case the example $e_{Q \cup Y}$ satisfies constraints that are already in Cs_L , with at least one variable in Y_2 , we call *SearchSC* recursively for each subset of Y created by removing one of the variables of the scope of such a constraint (lines 7-11). This is done to ensure soundness, as we now explain.

Ensuring Soundness. Before continuing with the description of the algorithm, let us clarify an important issue. It is possible that when *SearchSC* has focused on a partial example in a set of variables Y and posts a preference query including this example, there may already exist satisfied constraints from Cs_L (i.e. constraints we have already learned) with scopes having at least one variable in Y_2 , i.e. in the set of variables in which we search. Consider the running example. After the system finds the constraint c_{37} from example e , it will continue searching. After finding that x_8 is in the scope of a constraint we seek, it will now search in $Y_2 = \{x_1, \dots, x_7\}$ for the rest of the variables in the same scope. However, the already learned constraint c_{37} is satisfied in the projection of e on $\{x_1, \dots, x_7\}$. This will affect the answers of the user in the subsequent queries and will mislead the algorithm. As a result, instead of learning scope $\{x_3, x_8\}$, it will also include x_7 in the learned scope, which is incorrect, making the algorithm unsound.

To resolve this problem, *SearchSC* does the following: For each satisfied constraint in Cs_L , with $var(c) \subset Y \wedge var(c) \cap Y_2 \neq \emptyset$, it recursively searches in partial examples created by removing one of the variables in $var(c)$. This guarantees that any constraint c' , with

$var(c') \subset Y \wedge var(c') \cap Y_2 \neq \emptyset$ that will be learned is indeed in Cs_T . As our assumption is that the constraint network is normalized, at least one variable appears in $var(c)$ but not in $var(c')$, meaning that by removing this variable, the algorithm will be able to search in an example where c is not satisfied while c' is. Therefore, it will be able to learn c' , without c affecting the answers of the user to the preference queries posted.

With this method, the system exploits the fact that for any satisfied constraint $c' \in Cs_T$ which we have not already learned, we have $var(c) \setminus var(c') \neq \emptyset$ for every $c \in Cs_L$. This is true because of the assumption that the target constraint network is normalized. In our example, *SearchSC* will search in the two sets of variables created by removing one of the variables in the scope of the learned constraint c_{37} , i.e. $\{x_1, x_2, x_4, \dots, x_7\}$ and $\{x_1, \dots, x_6\}$. Constraint c_{37} is not satisfied in the projection of e in either of these sets of variables, so eventually the algorithm will discover that x_3 is the other variable it seeks, and learn c_{38} .

We now continue with the description of the algorithm. A preference query is posted to the user at line 14, when the example cannot be simplified any more. The examples compared are $e_{Q \cup Y}$ and $e_{Q \cup Y_1}$, meaning that the information we get from the answer of the user regards the variables in Y_2 , which is the set of variables that belong to Y but not to Y_1 . As Y_1 is previously given as Y to the recursive call at line 6, we definitely have already searched in there. So, searching now in Y_2 , we finish searching in Y . The preference query is posted only if it is not redundant (checks at lines 12,13), e.g. query 6 in the running example will not be actually posted because the answers to previous queries (queries 3-5) imply the user's answer. From query 3 we know that there is at least one satisfied constraint with a variable of its scope in $\{x_5, \dots, x_8\}$, and from queries 4-5 we know that this variable is not in $\{x_5, x_6\}$, so it is certain that it is in $\{x_7, x_8\}$, and the relevant query can be avoided. This is what the check in line 13 does, with the Boolean variable *ask_query* (given as a parameter) denoting whether we know that at least one satisfied constraint exists in Y and *found_flag* specifying whether a constraint has been already found in any recursive call until now or not (i.e. if a satisfied constraint was found in Y_1).

If the answer to the query is that the user does not prefer any example, then the satisfied constraints in $B[Q \cup Y]$ are removed from B and false is returned (lines 15,16). Otherwise, if we have reached line 17, we know that Y_2 contains at least one variable from a satisfied constraint from Cs_T that we have not learned yet, because the preference query of type 2 can only have two answers: Either the user is indifferent, or she prefers the example containing more variables. If $|Y_2| > 1$, then *SearchSC* is recursively called with $R = R \cup Y_1$ and $Y = Y_2$, to continue searching in Y_2 (line 17). Notice that *ask_query* is set to false, because we now know that a constraint certainly exists in Y_2 . So, in the next recursive call (where Y_2 will be given as Y) we know that if no constraint is found in any sub-call, then the query does not have to be posted. With *ask_query* set to false in a recursive call, we know that Y contains at least one variable of the scope. So, if the variable is not found in Y_1 , we know it is in Y_2 . *found_flag* will show us if it is found in Y_1 or not in the check of line 13, as in query 6 of the running example where we know that there is a variable of the scope we seek in $Y = \{\{x_5, \dots, x_8\}\}$ (so in this recursive call we have *ask_query* = false) and no constraint has been found in Y_1 because of queries 4-5. In case $|Y_2| = 1$, we know that it is in the scope of the constraint we seek because the user answered that she prefers the example having Y_2 instantiated. Thus, *SearchSC* is recursively called with $R = \emptyset$, $Y = R \cup Y_1$ and $S = S \cup Y_2$, to search for more variables of the scope in $R \cup Y_1$ (line 19).

If no more variables are discovered at a recursive call, then $S \cup Y_2$ is the scope we seek, so *FindSC* is called to find the specific constraint, which is then added to Cs_L (line 20). Finally, having reached this point means that a constraint has been found either by this call or by a recursive call, so true is returned.

4.2.1 Finding the specific relation

In order to find the specific relation of the constraint sought, in the scope found by *SearchSC*, two functions are used, *FindSC* and *FindSC-2*. *FindSC* is the main function used to find the specific constraint, after a scope has been located, while *FindSC-2* is used under certain circumstances, in case there is any constraint c in Cs_L , with $var(c) \subset S$, that is satisfied by the examples posted to the user, affecting her answers and preventing *FindSC* from learning the specific relation. If this is the case, the constraint may not be learned via the main loop of *FindSC*, so another technique is used in *FindSC-2*.

FindSC (Algorithm 3) takes as parameters e and S , where e is the example in which *SearchSC* located the scope of a satisfied constraint from Cs_T , and S is that scope. It posts partial preference queries of the 1st type to find the specific relation. It returns the soft constraint found to be in Cs_T . The main idea is to compare example e to an example e' that satisfies at least one candidate constraint that e also satisfies, but not all such constraints. In this way, we can shrink the set with the candidate constraints after each query.

■ **Algorithm 3** FindSC:

Input: e, S (e : the example, S : the scope of the soft constraint we seek)

Output: c : the constraint found

```

1: function FindSC( $e, S$ )
2:    $\Delta \leftarrow \{c \in B \mid var(c) = S\};$ 
3:    $B \leftarrow B \setminus \Delta;$ 
4:    $\Delta \leftarrow \lambda_\Delta(e_S);$ 
5:   while true do
6:     Generate  $e'$  in  $D^S$  accepted by  $Ch$ , s.t.  $\lambda_{Cs_L}(e') = \lambda_{Cs_L}(e) \wedge \lambda_\Delta(e') \neq \lambda_\Delta(e) \wedge \lambda_\Delta(e') \neq \emptyset;$ 
7:     if  $e' \neq nil$  then
8:        $answer \leftarrow PrefAsk(e', e);$ 
9:       if  $answer = (e \succ e')$  then  $\Delta \leftarrow \kappa_\Delta(e');$ 
10:    else
11:       $found \leftarrow false;$ 
12:      if  $\exists c \in \lambda_B(e') \mid var(c) \subset S$  then
13:        for each  $x_i \in S$  do
14:           $found \leftarrow SearchSC(e, \emptyset, S \setminus \{x_i\}, true) \vee found\_flag;$ 
15:      if  $found = false$  then
16:         $\Delta \leftarrow \lambda_\Delta(e');$ 
17:    else break;
18:  if  $\exists c \in Cs_L \mid var(c) \subset S \wedge |\Delta| > 1$  then  $\Delta \leftarrow FindSC-2(S, \Delta);$ 
19:  pick random  $c \in \Delta;$  return  $c;$ 
```

FindSC first initializes the set Δ to the candidate constraints, i.e. the constraints from B with scope S that are satisfied by e , and removes them from B (lines 2-4) because after the constraint is found no other constraint with scope S can exist in Cs_T , given our normalization assumption. In line 5, *FindSC* enters its main loop in which it posts preference queries to the user. In line 6, a partial example e' is generated, to be compared to e_S , that is accepted by Ch and satisfies fewer constraints from Δ than e_S , but at least one. On the way example e'_S is generated, if there is no satisfied constraint $c \in Cs_T$ with $var(c) \subset S$ that will affect the answer of the user, then whatever the answer of the user is, at least one candidate

constraint will be eliminated. This is because if the user is indifferent between the examples, the satisfied constraints from e_S that are not satisfied by e'_S are removed (line 16), while if the user prefers e_S , the satisfied constraints by e'_S cannot contain the one we seek, so they are removed (line 9).

In order to make sure that there is no satisfied constraint $c \in Cs_T$ with $\text{var}(c) \subset S$ in the examples e_S and e'_S that will affect the answer of the user on which example is preferable, the system tries to generate an example that satisfies the same constraints in Cs_L as e_S . However, there are two cases that this may not be possible and special handling is required.

First, when the generated example e'_S may satisfy a not yet learned constraint in Cs_T . To handle this, if there are candidate constraints from B , with a scope subset of S , that are satisfied by e'_S (line 12), *FindSC* checks if any of them is in the target network, in lines 13,14. This is done, by calling *SearchSC* in all the examples $e_{S \setminus x_i}$, $\forall x_i \in S$, in a technique similar to the one used in lines 6-10 of *SearchSC*. If at least one constraint is found to be in the target network, the system returns on generating a new example in line 6, as no useful information can be derived from the answers of the user with the current example.

The second case is when there exists an already learned constraint c (i.e. in Cs_L), with $\text{var}(c) \subset S$. This may affect the answers of the user, preventing *FindSC* from generating an example e'_S with the necessary properties, and thus making it not possible to learn the specific relation. Specifically, the reason that such an example may not be generated in line 6, although all the constraints in Δ are not yet equivalent w.r.t Ch , is the existence of constraint(s) $c \in Cs_L$ with a scope $\text{var}(c) \subset S$, that may not allow the generation of an example e' accepted by Ch , with $\lambda_{Cs_L}(e') = \lambda_{Cs_L}(e) \wedge \lambda_{\Delta}(e') \neq \lambda_{\Delta}(e) \wedge \lambda_{\Delta}(e') \neq \emptyset$.

Let us give an example: Assume that $\Delta = \{c1, c2\}$, and thus $e \in \text{sol}(c1) \cap \text{sol}(c2)$. Also, we have constraints $c3, c4 \in Cs_L$ with $\text{var}(c3), \text{var}(c4) \subset S$ and we have $\text{sol}(c3) = \text{sol}(c1) \setminus \text{sol}(c2)$ and $\text{sol}(c4) = \text{sol}(c2) \setminus \text{sol}(c1)$

In this case, if we want to generate an example e' with $\lambda_{\Delta}(e') \neq \lambda_{\Delta}(e) \wedge \lambda_{\Delta}(e') \neq \emptyset$, e' will satisfy either $c1$ or $c2$, and this will affect the user preferences in the query in line 8, which makes it not possible to use the answer to find which of $c1, c2$ is in Cs_T . So, in order to have an informative query in line 8, we must be able to generate an example e' s.t. $\lambda_{Cs_L}(e') = \lambda_{Cs_L}(e)$ along with the rest of the properties, which we cannot in this case.

Also, another case is where there may be a constraint $c \in Cs_L$ accepted by e that cannot be accepted by any example e' with the desired properties.

To deal with such cases, function *FindSC-2* is called in line 18, to find the constraint we seek, or to prove that all constraints in Δ are equivalent w.r.t the hard constraints.

FindSC-2 (Algorithm 4) has two main loops, to cover all different cases where a constraint with a scope subset of S exists in Cs_L . The first loop is in lines 2-16 and the second loop is in lines 17-31. Each one deals with different cases. They both remove constraints from Δ that provably cannot be in Cs_T . *FindSC-2* returns the final Δ with all constraints being equivalent w.r.t. the hard constraints of the problem.

Algorithm 4 FindSC-2:

Input: S, Δ (S : the scope of the soft constraint we seek, Δ : the set containing the constraints that may belong to Cs_T)

Output: Δ : the set containing the constraints that may belong to Cs_T

```

1: function FindSC-2( $S, \Delta$ )
2:   while true do
3:     Generate  $e, e'$  in  $D^S$  accepted by  $Ch$ , s.t.  $\lambda_{Cs_L}(e') = \lambda_{Cs_L}(e) \wedge \lambda_\Delta(e') \neq \lambda_\Delta(e) \wedge \lambda_\Delta(e'), \lambda_\Delta(e) \neq \emptyset$ ;
4:     if  $e' = nil \vee e = nil$  then break;
5:      $found \leftarrow false$ ;
6:     if  $\exists c \in \lambda_B(e) \mid var(c) \subset S$  then
7:       for each  $x_i \in S$  do
8:          $found \leftarrow SearchSC(e, \emptyset, S \setminus \{x_i\}, true) \vee found\_flag$ ;
9:     if  $\exists c \in \lambda_B(e') \mid var(c) \subset S$  then
10:      for each  $x_i \in S$  do
11:         $found \leftarrow SearchSC(e', \emptyset, S \setminus \{x_i\}, true) \vee found\_flag$ ;
12:     if  $found = false$  then
13:        $answer \leftarrow PrefAsk(e', e)$ ;
14:       if  $answer = (e \sim e')$  then  $\Delta \leftarrow \Delta \setminus ((\lambda_\Delta(e) \cup \lambda_\Delta(e')) \setminus (\lambda_\Delta(e) \cap \lambda_\Delta(e')))$ ;
15:       else if  $answer = (e \succ e')$  then  $\Delta \leftarrow \lambda_\Delta(e) \setminus \lambda_\Delta(e')$ ;
16:       else  $\Delta \leftarrow \lambda_\Delta(e') \setminus \lambda_\Delta(e)$ ;
17:   while true do
18:     Generate  $e$  in  $D^S$  accepted by  $Ch$ , s.t.  $\emptyset \subset \lambda_\Delta(e) \subset \Delta \wedge \lambda_{Cs_L}(e) \neq \emptyset$ ;
19:     if  $e \neq nil$  then
20:        $answer \leftarrow PrefAsk(e_S, e_{var(\lambda_{Cs_L}(e))})$ ;
21:       if  $answer = (e_S \sim e_{var(\lambda_{Cs_L}(e))})$  then  $\Delta \leftarrow \kappa_\Delta(e_S)$ ;
22:       else
23:          $found \leftarrow false$ ;
24:         if  $\exists c' \in \lambda_B(e) \mid var(c') \subset S \wedge var(c') \supset var(c)$  then
25:           for each  $x_i \in S$  do
26:              $found \leftarrow SearchSC(e, \emptyset, S \setminus \{x_i\}, true) \vee found\_flag$ ;
27:         if  $found = false$  then
28:            $\Delta \leftarrow \lambda_\Delta(e)$ ;
29:     else break;
  return  $\Delta$ ;

```

In more details, in the first loop, a partial example e_S is generated (line 3), that is accepted by Ch and satisfies fewer constraints from Δ than e_S , but at least one.

The main idea is the following. The system generates two new examples e, e' s.t. each one satisfies a different *subset* of Δ and the same subset of Cs_L . Next, these examples are compared, eliminating parts from Δ from the candidate constraints in a repetitive process, depending on the answer of the user on which example is preferable. If one is preferable then only the constraints satisfied only by that example stay in Δ . If the user is indifferent between the two, then the examples satisfied by one example and not by the other are removed from Δ .

If the system cannot generate such a pair of examples, because e.g. all examples satisfying a (different) subset of Δ also satisfy a (different) subset of Cs_L , then it tries to generate a new example e_S , satisfying at least one constraint from Δ but not all, satisfying also a constraint

$c \in Cs_L$ with $\text{var}(c) \subset S$, if one exists. Then it posts a preference query comparing the example e_S with its projection in $\text{var}(c)$, in order to find out if the constraint(s) in Δ that are satisfied are in the target network. If the user is indifferent between the examples, it means that the constraints in Δ satisfied by e_S are not in the target network and are removed. Otherwise, only these constraints remain in Δ .

If no example can be generated, then the system can return randomly a constraint from Δ in line 19 of *FindSC*, because they are all equivalent w.r.t. *Ch*.

5 Experimental Evaluation

We first detail the experimental setting.

- Experiments were run on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with 16 GB of RAM
- The max_B heuristic [42] was used for query generation, altered to fit the context of soft constraints. max_B normally focuses on examples violating as many constraints from B as possible. But it now focuses on examples satisfying as many constraints from B as possible. The best such example (according to max_B) found within 1 second is returned, even if not proved optimal. Also, bdeg was used for variable ordering. The variable appearing in the most constraints in B is chosen [40]. Random value ordering was used.
- We evaluated our algorithm in the extreme case where Cs_L is initially empty, meaning that we have no background knowledge. This results in a large number of queries. However, in real applications it is common to have background knowledge and use it e.g. by giving a frame of basic constraints.
- We evaluate our algorithm on learning the soft constraints, while hard constraints are given by the user. If the hard constraints are unknown too, a constraint acquisition algorithm like QuAcq [4], MQuAcq [42] or MQuAcq-2 [41] can be used to learn them before using PrefAcq to learn the soft constraints.
- We measure the size of the learned network Cs_L , the total number of queries $\#queries$, the average time per query \bar{T} and the total cpu time of the acquisition process T . The time measured is in seconds. PrefAcq was run 5 times on each benchmark. The means are presented along with the standard deviation .

We used the following benchmarks:

Random. We generated two classes of random Max-CSPs with 50 variables and domains of size 10. The first instance consists of 12 hard and 100 soft constraints, while the second consists of 122 hard and 30 soft constraints. All the hard ones are \neq constraints, while the soft are among $\{\neq, >, <\}$. The bias was initialized with 7,350 constraints, using the language $\Gamma = \{=, \neq, >, <, \geq, \leq\}$.

Radio Link Frequency Assignment Problem. The RLFAP is the problem of providing communication channels from limited spectral resources [13]. We used a simplified version of the RLFAP [13], with 50 variables having domains of size 15. The target network contains 100 hard and 25 soft constraints. B contains 12,250 constraints from the language of 2 distance constraints ($\{|x_i - x_j| > y, |x_i - x_j| = y\}$) with 5 different possible values for y .

Exam Timetabling Problem. We used a simplified version of the exam timetabling problem from the Elect. Eng. Dept. of UOWM, Greece. We considered 24 courses, and 2 weeks of exams, meaning that there are 10 possible days for each course to be assigned. We assumed

that there are 3 timeslots in each day. This resulted in a model with 24 variables and domains of size 30. There are hard \neq constraints between any two courses, assuming that only one course is examined during each time slot. Also, hard constraints prohibit courses of the same semester being examined on the same day. 30 soft constraints from the language $\Gamma = \{\neq, >, <, |x_i - x_j| > y, |x_i - x_j| < y\}$, capture the lecturers' and the students' preferences about the examination of specific courses. We built a bias with 5,796 constraints from the language $(\Gamma = \{\neq, =, >, <, \geq, \leq, |x_i - x_j| > y, |x_i - x_j| < y, \lfloor x_i/3 \rfloor - \lfloor x_j/3 \rfloor > y\})$ with 5 different possible values for y . This resulted in a bias containing 5,796 constraints in total.

Note that although all the benchmarks we used are binary (i.e. their constraint network consists of binary constraints only), our proposed method works of constraints of any arity, as long as it is bounded, as mentioned in Section 2. However, it does not work on global constraints, but neither does any active constraint acquisition algorithm, as they are unbounded, which means the bias should have an exponential size on the number of variables of the problem.

■ **Table 2** Results of PrefAcq.

Benchmark	$ C_s L $	$\#q$	\bar{T}	T_{total}
Random122-30	30 ± 0	859 ± 25	0.03 ± 0.001	27.29 ± 0.78
Random12-100	100 ± 0	2234 ± 65	0.02 ± 0.001	39.43 ± 1.06
RLFAP	25 ± 0	621 ± 26	0.33 ± 0.02	209.26 ± 12.56
Exam TT	30 ± 0	751 ± 13	0.19 ± 0.02	146.54 ± 16.30

Table 2 presents the results of PrefAcq. We see that the number of queries is proportional to the number of constraints learned. Also, comparing the two random problems, we see that although the number of queries increases when learning more soft constraints, the average time between two queries is about the same. This is because the number of queries depends only on the number of soft constraints we have to learn and on the size of B , while the waiting time depends on the time taken for query generation. As both random problems are easy to solve, queries are generated very fast. In contrast, in the RLFAP and the Exam Timetabling problem (denoted as Exam TT), which are harder to solve, the system takes more time to generate examples that do not violate any hard constraints and satisfy at least one $c \in B$, at line 3 of PrefAcq. But still, the average waiting time for the user is under 1 second.

6 Conclusion

We have extended the framework of active constraint acquisition to the learning of soft constraints in Max-CSPs. Based on a type of query used in preference elicitation, we introduced partial preference queries. Then we presented PrefAcq, a novel algorithm for learning soft constraints in Max-CSPs using such queries. Finally, we give some preliminary experimental results. Our method can be extended to weighted Max-CSP if PrefAcq is used to learn the constraints and a method such as the one in [35] is then used to learn their weights. The existence of weights in constraints does not affect the procedure followed by our method.

References

- 1 Dana Angluin. Queries and concept learning. *Machine learning*, 2(4):319–342, 1988.
- 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*, pages 141–157. Springer, 2012.

- 3 Christian Bessiere, Remi Coletta, Eugene C Freuder, and Barry O’Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In *International Conference on Principles and Practice of Constraint Programming*, pages 123–137. Springer, 2004.
- 4 Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, Toby Walsh, et al. Constraint acquisition via partial queries. In *International Joint Conference on Artificial Intelligence (IJCAI)*, volume 13, pages 475–481, 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 *International Joint Conference on Artificial Intelligence (IJCAI)*, volume 7, pages 50–55, 2007.
- 7 Christian Bessiere, Abderrazak Daoudi, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Younes Mechqrane, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. New approaches to constraint acquisition. In *Data mining and constraint programming*, pages 51–76. Springer, 2016.
- 8 Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O’Sullivan. Constraint acquisition. *Artificial Intelligence*, 244:315–342, 2017.
- 9 Alessandro Biso, Francesca Rossi, and Alessandro Sperduti. Experimental results on learning soft constraints. *KR*, 2:435–444, 2000.
- 10 Craig Boutilier, Relu Patrascu, Pascal Poupart, and Dale Schuurmans. Regret-based utility elicitation in constraint-based decision problems. In *IJCAI*, volume 9, pages 929–934, 2005.
- 11 Craig Boutilier, Relu Patrascu, Pascal Poupart, and Dale Schuurmans. Constraint-based optimization and utility elicitation using the minimax decision criterion. *Artificial Intelligence*, 170(8-9):686–713, 2006.
- 12 Céline Brouard, Simon de Givry, and Thomas Schiex. Pushing data into cp models using graphical model learning and solving. In *International Conference on Principles and Practice of Constraint Programming*, pages 811–827. Springer, 2020.
- 13 Bertrand Cabon, Simon De Givry, Lionel Lobjois, Thomas Schiex, and Joost P. Warners. Radio link frequency assignment. *Constraints*, 4(1):79–89, 1999.
- 14 Paolo Campigotto, Andrea Passerini, and Roberto Battiti. Active learning of combinatorial features for interactive optimization. In *International Conference on Learning and Intelligent Optimization*, pages 336–350. Springer, 2011.
- 15 Li Chen and Pearl Pu. Survey of preference elicitation methods. Technical report, EPFL, 2004.
- 16 Vincent Conitzer. Eliciting single-peaked preferences using comparison queries. *Journal of Artificial Intelligence Research*, 35:161–191, 2009.
- 17 Luc De Raedt, Anton Dries, Tias Guns, and Christian Bessiere. Learning constraint satisfaction problems: An ilp perspective. In *Data Mining and Constraint Programming*, pages 96–112. Springer, 2016.
- 18 Luc De Raedt, Andrea Passerini, and Stefano Teso. Learning constraints from examples. In *Proceedings in Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
- 19 Carmel Domshlak, Eyke Hüllermeier, Souhila Kaci, and Henri Prade. Preferences in ai: An overview, 2011.
- 20 Paolo Dragone, Stefano Teso, and Andrea Passerini. Constructive preference elicitation. *Frontiers in Robotics and AI*, 4:71, 2018.
- 21 Eugene C Freuder. Modeling: the final frontier. In *The First International Conference on The Practical Application of Constraint Technologies and Logic Programming (PACLP)*, London, pages 15–21, 1999.
- 22 Eugene C Freuder. Progress towards the holy grail. *Constraints*, 23(2):158–171, 2018.

- 23 Eugene C Freuder and Barry O’Sullivan. Grand challenges for constraint programming. *Constraints*, 19(2):150–162, 2014.
- 24 Eugene C Freuder and Richard J Wallace. Suggestion strategies for constraint-based match-maker agents. In *International Conference on Principles and Practice of Constraint Programming*, pages 192–204. Springer, 1998.
- 25 Mirco Gelain, Maria Silvia Pini, Francesca Rossi, K Brent Venable, and Toby Walsh. Elicitation strategies for soft constraint problems with missing preferences: Properties, algorithms and experimental studies. *Artificial Intelligence*, 174(3-4):270–294, 2010.
- 26 Shengbo Guo and Scott Sanner. Real-time multiattribute bayesian preference elicitation with pairwise comparison queries. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 289–296, 2010.
- 27 Samuel M Kolb. Learning constraints and optimization criteria. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.
- 28 Mohit Kumar, Samuel Kolb, Stefano Teso, and Luc De Raedt. Learning max-sat from contextual examples for combinatorial optimisation. In *AAAI*, pages 4493–4500, 2020.
- 29 Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. In *22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, volume 1, pages 45–52. IEEE, 2010.
- 30 Tom Michael Mitchell. Version spaces: an approach to concept learning. Technical report, Stanford Univ Calif Dept of Computer Science, 1978.
- 31 Barry O’Sullivan. Automated modelling and solving in constraint programming. In *AAAI Conference on Artificial Intelligence*, pages 1493–1497, 2010.
- 32 Steven D Prestwich. Robust constraint acquisition by sequential analysis. *Frontiers in Artificial Intelligence and Applications*, 325:355–362, 2020.
- 33 Jean-Francois Puget. Constraint programming next challenge: Simplicity of use. In *International Conference on Principles and Practice of Constraint Programming*, pages 5–8. Springer, 2004.
- 34 Francesca Rossi and Alessandro Sperduti. Learning solution preferences in constraint problems. *Journal of Experimental & Theoretical Artificial Intelligence*, 10(1):103–116, 1998.
- 35 Francesca Rossi and Allesandro Sperduti. Acquiring both constraint and solution preferences in interactive constraint systems. *Constraints*, 9(4):311–332, 2004.
- 36 Francesca Rossi, Kristen Brent Venable, and Toby Walsh. Preferences in constraint satisfaction and optimization. *AI magazine*, 29(4):58–58, 2008.
- 37 Kostyantyn Shchekotykhin and Gerhard Friedrich. Argumentation based constraint acquisition. In *Ninth IEEE International Conference on Data Mining*, pages 476–482. IEEE, 2009.
- 38 Atena M Tabakhi. Preference elicitation in dcops for scheduling devices in smart buildings. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.
- 39 Atena M Tabakhi, Tiep Le, Ferdinando Fioretto, and William Yeoh. Preference elicitation for dcops. In *International Conference on Principles and Practice of Constraint Programming*, pages 278–296. Springer, 2017.
- 40 Dimosthenis C Tsouros and Kostas Stergiou. Efficient multiple constraint acquisition. *Constraints*, 25(3):180–225, 2020.
- 41 Dimosthenis C Tsouros, Kostas Stergiou, and Christian Bessiere. Structure-driven multiple constraint acquisition. In *International Conference on Principles and Practice of Constraint Programming*, pages 709–725. Springer, 2019.
- 42 Dimosthenis C. Tsouros, Kostas Stergiou, and Panagiotis G. Sarigiannidis. Efficient methods for constraint acquisition. In *24th International Conference on Principles and Practice of Constraint Programming*, 2018.
- 43 Paolo Viappiani. Preference modeling and preference elicitation: An overview. In *DMRS*, pages 19–24, 2014.

- 44 Paolo Viappiani and Christian Kroer. Robust optimization of recommendation sets with the maximin utility criterion. In *International Conference on Algorithmic Decision Theory*, pages 411–424. Springer, 2013.
- 45 Xuan-Ha Vu and Barry O’Sullivan. Semiring-based constraint acquisition. In *19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007)*, volume 1, pages 251–258. IEEE, 2007.
- 46 Xuan-Ha Vu and Barry O’Sullivan. Generalized constraint acquisition. In *International Symposium on Abstraction, Reformulation, and Approximation*, pages 411–412. Springer, 2007.
- 47 Xuan-Ha Vu and Barry O’Sullivan. A unifying framework for generalized constraint acquisition. *International Journal on Artificial Intelligence Tools*, 17(05):803–833, 2008.

Parallelizing a SAT-Based Product Configurator

Nils Merlin Ullmann ✉

CAS Software AG, Karlsruhe, Germany

Tomáš Balyo ✉

CAS Software AG, Karlsruhe, Germany

Michael Klein ✉

CAS Software AG, Karlsruhe, Germany

Abstract

This paper presents how state-of-the-art parallel algorithms designed to solve the Satisfiability (SAT) problem can be applied in the domain of product configuration. During an interactive configuration process, a user selects features step-by-step to find a suitable configuration that fulfills his desires and the set of product constraints. A configuration system can be used to guide the user through the process by validating the selections and providing feedback. Each validation of a user selection is formulated as a SAT problem. Furthermore, an optimization problem is identified to find solutions with the minimum amount of changes compared to the previous configuration. Another additional constraint is deterministic computation, which is not trivial to achieve in well performing parallel SAT solvers. In the paper we propose five new deterministic parallel algorithms and experimentally compare them. Experiments show that reasonable speedups are achieved by using multiple threads over the sequential counterpart.

2012 ACM Subject Classification Applied computing → E-commerce infrastructure

Keywords and phrases Configuration, Satisfiability, Parallel

Digital Object Identifier 10.4230/LIPIcs.CP.2021.55

Related Version The paper is based on Nils Merlin Ullmann's Master's thesis.

Extended Version: <https://doi.org/10.5281/zenodo.5154040>

1 Introduction

Configuration systems [12] offer great benefits for the sales process and customers, while technical challenges rise to enable configuration models with increasingly large knowledge bases. Every valid configuration has to fulfill a set of propositional formulas. The configuration system's task is to find such an assignment for a given set of user requirements. Janota showed [25] that this problem can be expressed as a *Boolean Satisfiability Problem* (SAT) and consequently solved by a SAT-solver. Essentially, the configuration model together with the user requirements are transformed into conjunctive normal form (CNF). If the formula is satisfiable, then there exists at least one valid configuration for the underlying configuration model and the user's needs.

Parallel SAT-solvers have been mainly studied on very hard SAT problems. Configuration systems tend to have different requirements that exceed the common SAT problem. Generally, the created problem instances are smaller and less complex, due to the a step-by-step configuration process. However, new problems are introduced. Firstly, in case that the customer's latest selection in the interactive process combined with the current configuration are unsatisfiable, the configuration system should return an alternative solution that minimizes the number of changes with regard to the current configuration. This problem can be modeled as an optimization problem, for example as a *Maximum Satisfiability Problem* (MaxSAT) or *Minimum-Cost Satisfiability Problem* (MinCostSAT). The underlying cost function evaluates



© Nils Merlin Ullmann, Tomáš Balyo, and Michael Klein;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 55; pp. 55:1–55:18

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the changes of every assignment compared to the current configuration. Additionally a configuration system requires fully deterministic behavior and very low response time (in the order of hundreds of milliseconds).

The aim of this work is to report on how state-of-the-art parallel SAT solving techniques can be adapted and used in a commercial product configurator. Our main contribution is the development of parallel algorithms for problem instances created during interactive configuration processes¹. These instances are described by calculating the optimal valid solutions for a given user change, start-configuration, and Boolean formula. Several parallel algorithms are presented that fulfill the completeness and optimality criteria for assignments required by configuration systems. Experimental results show that the presented approaches can achieve significant improvements in response time by using multiple processor cores. Furthermore, the approaches fulfill the completeness, optimality, and determinism requirements. Especially the latter has not been widely researched in this domain, because many applications do not rely on reproducible results.

2 Related Work

SAT and two of its extensions (MaxSAT and MinCostSAT) have been applied to the field of product configuration [46, 47]. Methods were presented to check the general consistency of a product data base, by converting it into formulas of propositional logic. This was done in a real product configuration system for the automotive industry. A broader discussion of the applicability of SAT solvers for configuration systems is given in [25], where scalability is emphasized as a potential benefit. Early research on concepts focusing on the configuration process has been presented by Sabin and Weigel in [43]. A good overview of the *interactive configuration* process, in which a user specifies his requirements step-by-step with validation and feedback in between, is given in [27]. Different solving techniques for this process have been proposed. Batory and Freuder et al. present in [3] and [13] algorithms for a lazy approach. In this context lazy means that no precompilation is required, because all computations are performed during the configuration process. Non-lazy approaches mainly use binary-decision-diagrams (BDD) instead of SAT solvers, examples are [16] and [1]. Furthermore, Janota discusses many optimization techniques and algorithms to model the interactive configuration process lazily with the help of SAT [26]. Additionally, he elaborates on methods to improve the transparency of a configuration system. This is achieved by providing algorithms to generate comprehensible explanations using resolution trees and completing partial configurations.

Currently, most parallel SAT solvers are based on one of the two main approaches: *divide-and-conquer* (also called search space splitting) and *parallel portfolios*. Early parallel SAT solvers (PSATO [49], PSatz [30], PaSAT [45], GridSAT [7], MiraXT [34], PaMiraXT [44]) were based on the divide-and-conquer approach. The search space in these solvers is divided dynamically using *guiding-paths*. Another approach is to divide the search space statically at the beginning of the search using look-ahead techniques [22]. This paradigm is called

¹ We adapted already existing algorithms, however, they are designed for SAT solving, which still has several differences to our problem (decision vs. optimization problem, non-deterministic vs. deterministic behavior, focus on large difficult problems vs. real time response, non-interactive vs. interactive usage). Therefore our main contribution is the non-trivial adaptation of the SAT algorithms to product configuration. The second contribution is the evaluation of these algorithms on real industrial configuration problems (and some random problems) and identifying their strengths and weaknesses in that context.

cube-and-conquer. It is a two-phase approach that partitions the original problem into many subproblems (cubes) which are subsequently solved in parallel [23, 5]. Parallel portfolios, popularized by Hamadi et al. with the solver ManySAT [18], differ by starting several SAT solvers on the same formula but with different parameter settings in parallel. The solvers compete to find a solution to the input problem and terminate as soon as one has been found. More recent examples of portfolio solvers are Plingeling [4] or HordeSat [2]. Most parallel SAT solvers introduce non-deterministic behavior, which is not acceptable in some applications, such as ours for example. This problem has been acknowledged by Hamadi et al. [17], who proposed the first deterministic parallel SAT solver based on ManySAT. A recent result [40] shows that comparable performance to non-deterministic parallel SAT solvers is achievable by using techniques such as delayed clause exchange and accurate estimation of execution time of clause exchange intervals.

This paper is focused on the combination of SAT and optimization problems. Hence, we also discuss parallel best-first search algorithms. Approaches for parallel A* can be separated into two major categories, depending on the management of the OPEN list. A centralized parallel A* [24, 41] works on a shared central OPEN list [41]. To remove potential bottlenecks on the shared OPEN list, algorithms that use the so called decentralized approach have been developed. The algorithm PRA* (Parallel Retracting A*) assigns an OPEN list to each processor [10]. Every generated node is mapped to a processor using a hash function. Following work mainly concentrated on developing sophisticated hash functions to reduce overhead [31, 29].

3 Preliminaries

A *Boolean variable* has two possible values: *True* and *False*. A *literal* is a Boolean variable (positive literal) or a negation of a Boolean variable (negative literal). A *clause* is a disjunction (\vee) of literals and, finally, a CNF formula (or just formula) is a conjunction of clauses. A clause with only one literal is called a *unit clause*. A positive (resp. negative) literal is satisfied if the corresponding variable is assigned the value *True* (resp. *False*). A clause is satisfied, if at least one of its literals is satisfied and the formula is satisfied, if all its clauses are satisfied.

The satisfiability (SAT) problem is to determine whether a given formula has a satisfying assignment, and if so, also find it. Most complete SAT solvers are based on the DPLL algorithm [8] and its extension the CDCL algorithm [38, 39].

SAT only searches for an assignment that satisfies the Boolean formula. Optimality with regard to the assignment is not considered. Two approaches that introduce an optimization function are the Maximum Satisfiability Problem (MaxSAT) and the Minimum-Cost Satisfiability Problem (MinCostSAT). Both problems extend SAT by incorporating a cost function that evaluates assignments. The better known MaxSAT problem is to find an assignment that maximizes the number of satisfied clauses of a Boolean formula [35]. In the MinCostSAT problem we assign a non-negative cost to each variable to quantify an assignment. The problem is to find a variable assignment that satisfies F and minimizes the total cost of the variables set to *True*. The transformation between MaxSAT and MinCostSAT problems can be performed by adding auxiliary variables/clauses to the respective formulas [36].

The DPLL/CDCL algorithm can be extended to solve MinCostSAT instances. The cost for every variable assignment are accumulated during unit propagation. At every branching point a decision variable is chosen, partial assignments are calculated, and the assignment with the lowest costs is used. This amounts to a recursive branch-and-bound algorithm [14, 36].

A configuration task is a triplet (V, D, C) , where $V = \{v_1, v_2, \dots, v_n\}$ is a finite set of domain (feature) variables and $D = \{dom(v_1), dom(v_2), \dots, dom(v_n)\}$ represent the set of corresponding finite variable domains. Furthermore, $C = P_{KB} \cup C_R$ represent constraints, with P_{KB} being the product knowledge base (configuration model) and C_R a set of user requirements. The *solution* to a configuration task is called a *configuration*. A configuration is an instantiation (assignment) $I = \{v_1 = i_1, v_2 = i_2, \dots, v_n = i_n\}$, with each i_j being one of the elements of $dom(v_j)$. A configuration is called valid, if it is *complete* (every variable is assigned with a value) and *consistent* with all constraints [11]. Janota discusses in [25] whether SAT solvers can be used effectively in a configuration system (*configurator*). Janota shows that every configuration task can be translated into a Boolean formula. Consequently, by solving the Boolean formula in CNF by using a SAT solver, the initial configuration task is solved as well.

4 Problem definition

The term *configuration* can be interpreted as an assignment for an underlying Boolean formula F_{cnf} . During the interactive configuration process, a user selects or deselects attributes step-by-step to add or remove them from a configuration. The attributes are translated to corresponding Boolean variables of F_{cnf} , thus every attribute is also an atomic proposition. A selection expresses that the attribute must be part of the current configuration. Any selection or deselection can be reverted throughout the configuration process. A configuration is considered valid if it is consistent with all constraints (C), but not every variable (V) needs to be assigned. The user may start with an empty configuration which is progressing through user selections and deselections until it is complete. With respect to the underlying Boolean formula, an empty configuration contains every literal as a negative one. In the following, a user selection or deselection is also called a *user wish*.

During the configuration process, each user wish δ causes a *configuration step*. The step calculates a new valid configuration (solution) s using an existing assignment β (called *start-configuration*), by applying the user wish. A start-configuration describes the valid preexisting assignment for a configuration step. The user wish represents the desired change that should be applied to the existing configuration. Therefore, a user wish δ can be described as a set of literals.

For every configuration step, the configurator ensures the validity of the resulting configuration to prevent invalid user selections. Depending on the constraints, certain user wishes violate the configuration model which are resolved by the configurator through automatically selecting or deselecting attributes. To quantify the result of a configuration step, a cost function is introduced. The costs of the changes resulting from the user wish are calculated by analyzing the difference between the start-configuration β and the resulting configuration s :

$$\text{deltaCost}(\beta, s) = \sum_{l \in s} \begin{cases} l \in \beta \rightarrow 0 \\ l \notin \beta \rightarrow c(l) \in \mathbb{N}_{\geq 0} \end{cases} \quad (1)$$

The cost function $c(l)$ must be non-negative but can be domain specific. For example, literal changes from positive to negative can be more expensive to prefer keeping literals that the user already selected in the configuration process. A change from a positive literal to a negative one expresses a deselection of an attribute for the user. Each configuration step has two concrete requirements to fulfill.

1. Every configuration step has to apply the user wish δ to the start-configuration β , expressed as $\delta \subseteq s$.

2. The resulting configuration s has to be optimal, i.e., there is no other solution s' that results in lower costs (by applying the function $\text{deltaCosts}(\beta, s')$) and introduces the user wish δ to the start-configuration β .

The second requirement extends the SAT problem by incorporating an optimization problem of finding the minimal-cost configuration for each step. The problem of calculating a configuration step is similar to the *MinCostSAT* problem. The main difference is that positive variables do not inherently increase the costs. Costs are only accumulated by changes to the start-configuration, independent of the variable's truth value. The user can define a limit r on how many solutions should be returned at most in the case that multiple optimal valid configurations exist. All found solutions are represented by the set S . Furthermore, this set of solutions must be the same when repeating the same configuration step, i.e., we require fully deterministic calculation.

► **Definition 1** (*MinCostConf*). *The minimal-cost interactive configuration task (MinCostConf) is described by the 6-tuple:*

$$(A_p, C_p, \beta, \delta, c(l), r)$$

where A_p describes the set of attributes for a product p given its feature variables. The set of constraints for a specific product is given by C_p . The dynamic components are the start-configuration β and the user wish δ . Furthermore, a non-negative cost function $c(l)$ defines the cost for attribute $l \in A_p$, when l changes with respect to β . The maximum amount of returned solutions is limited to r . A solution s is valid if all constraints C_p are fulfilled and δ is part of each solution s ($\delta \subseteq s$). An optimal solution s is a minimal-cost assignment with respect to β ($\sum_{l \in s} c(l), \forall l \notin \beta$). A valid optimal solution s is called a configuration and the set of found solutions is denoted by S .

The task is to find all configurations. In case of more than r optimal configurations, the cardinality of S is limited to r . Repeatedly solving the same task must return the same set S , i.e., the process must be deterministic.

Regarding the complexity, *MinCostConf* (being an extension of *MinCostSAT*) obviously belongs to the class of NP-complete optimization problems.

5 Parallel Algorithms for Product Configuration

This Section contains the main contributions of the paper – the description of the parallel algorithms we developed for the *MinCostConf* problem. This paper is based on a Master Thesis [48], which contains a more detailed description of the described algorithms including pseudo-code, examples, and figures.

5.1 Baseline: Sequential A* Search for MinCostConf

The A* algorithm [19] is an extension of Dijkstra's algorithm [9]. The A* algorithm formulates its problem as a weighted directed graph and aims to find the minimal cost path from a source node to a goal node. In this process, the algorithm constructs a search tree and always expands the most promising node. Furthermore, the A* algorithm maintains an OPEN list of nodes that have not been expanded yet. The list is ordered by the cost accumulated from the root node to the current node and the heuristic estimation of the cost to reach a goal node. A second list, called CLOSED list contains all nodes that have been expanded. To decide which path to expand, the function $f(n) = g(n) + h(n)$ is minimized over all nodes,

where $g(n)$ is the path's cost from the source node to n , which is the next node on the path. Additionally, $h(n)$ estimates the cost of extending the path from node n to the goal node. The heuristic function $h(n)$ differentiates the A* algorithm from Dijkstra's algorithm. A heuristic function is called admissible if it does not overestimate the cost from node n to the goal node. For A* it holds in general, that if h is an admissible function, then A* search is optimal [19]. Consequently, utilizing this property allows us to terminate the search as soon as a solution has been found, because the estimated costs of all other nodes are larger than the actual cost of the found solution.

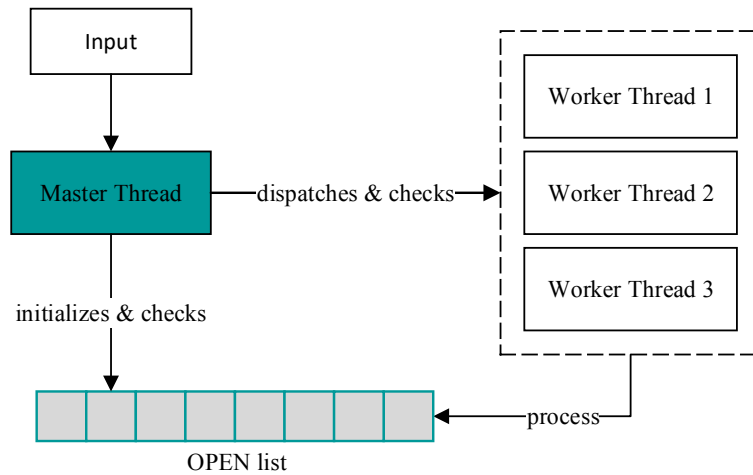
The A* algorithm can be combined with DPLL and applied to the MinCostConf problem, which is an optimization problem. The DPLL algorithm constructs a search tree which can be interpreted as a weighted directed graph. Every edge serves as a unit propagation of a decision literal. The edges' weights are given by the costs of the propagated literals, thus the weights are non-negative. This leads to a search tree in which the costs can only increase or remain unchanged with increasing tree depth. Finally, the goal node is the lowest cost valid assignment. In the domain of product configuration, multiple valid optimal solutions may exist, which represent potential alternatives for the user to choose from. Thus, several goal nodes may exist. In our case, the function $g(n)$ sums the costs of already assigned literals. The set L_n describes the literals that are assigned on the path from source node n_1 to node n_i . As a lower bound for the costs from n to the goal node, the heuristic function $h(n_i) = \text{costs}(U_{n_i})$ is used, where U_{n_i} is the set of unit clauses remaining in the node n_i . It accumulates the costs of all remaining unit clauses in F , that will result from the chosen literal, because their truth value is already defined but they have not been propagated yet. This obviously constitutes an admissible heuristic.

During the A* search, node evaluations are mainly based on the costs of propagated literals. Essentially, every change with regards to the start-configuration introduces costs. The start-configuration always describes the configuration prior to the current user wish. Having selected an attribute, the user wants as few changes to the prior configuration as possible, thus a cost function is utilized.

5.2 Centralized Parallel A* Search

An intuitive solution is to extend the sequential A* algorithm to perform it in a multithreaded environment. In parallel versions of the centralized A* approach, the single OPEN list is kept [24, 41]. Therefore, k threads work concurrently on a shared OPEN list. Each thread retrieves nodes from that data structure in order of their f -values, expands them, and inserts the successors of the expanded nodes (states) into the OPEN list. Re-expansions of nodes (contrary to the sequential algorithm) are possible, because a state may not have the optimal g -value when taken from the list and being expanded. However, this issue is not applicable to the DPLL algorithm, since nodes in the search tree are only reached by one specific path. This property leads to two improvements. Firstly, the g -value of a node cannot be updated by processing another node first. Secondly, duplicate detection of states is not required, because only one path leads to every state, thus two threads cannot arrive at the same state. Therefore, when using parallel A* search for the DPLL algorithm, the CLOSED list is not required. Nevertheless, the concurrent work can lead to search overhead by expanding suboptimal nodes that would not have been expanded by a sequential version of A*. Besides search overhead, expanding nodes in parallel easily leads to nondeterministic behavior. This critical problem as well as solutions are discussed thoroughly at the end of this section.

In a multithreaded environment, a coordination mechanism for all threads is required. We use the *Master-Worker* paradigm, with a single master thread and a set of worker threads. The latter can scale from 1 to k threads. The master maintains a overview of the procedure,



■ **Figure 1** An explanatory thread overview of the centralized parallel A* approach using a single master thread and three worker threads (Master-Worker paradigm). All threads are accessing one central OPEN list, using a shared-memory architecture.

while the workers process tasks in parallel. The master initializes the `MinCostConf` instance resulting in the root node and dispatches the k threads to start processing. Subsequently the workers remove nodes from the OPEN list, which is implemented as a priority queue. Each task consists of the unit propagation of a decision literal (except for the root node) and resulting unit clauses. Communication between threads is performed by using shared data, e.g. for thread state information and all currently known optimal solutions. Access to the shared data is required for workers as well as for the master. Workers need to know the current optimal solutions to decide whether their task (node) needs to be processed or can be skipped. While all workers are processing nodes, the master checks whether the OPEN list is empty, because an empty list indicates that the search is finished. Additionally, the master checks the state of all worker threads by accessing shared data, to decide whether the search can be terminated or not, for instance in case that the optimal solution has been found. See Figure 1 for an overview diagram.

The centralized parallel A* approach uses two shared primitive variables, the Boolean `searchFinished` and the integer `minCost`. Especially the latter is updated several times during a configuration step, because many suboptimal solutions may be found during the parallel search process. Thus, multiple threads try to retrieve and update the value of this variable simultaneously. To avoid memory inconsistency, all updates to these variables use optimistic locking in form of atomic compare-and-swap instructions. Moreover, several complex data structures are utilized. The most central data structure is the priority queue Q , representing the OPEN list by maintaining all nodes that may be expanded during the tree search. It is based on a binary heap and uses locks as a synchronization mechanism to allow parallel work of k threads on a single queue. An increasing number of threads (k) can result in high lock contention on Q . However, the assumption is that the number of operations on Q is relatively low in the domain of product configuration, due to solving smaller problem instances. Moreover, the node expansion is expensive, because it involves performing compute-intensive unit propagation which further limits the lock contention. For

search termination detection, two arrays thread-waiting states (TWS) and running-thread-costs (RTC) are used. Both enforce locking mechanisms to restrict concurrent thread access, for instance of the master thread and worker threads. While a worker thread retrieves a node from Q , the master thread has to wait until the worker thread has updated its waiting state as well as running costs. The described locking techniques are also applied in all other approaches discussed below.

5.3 Decentralized Parallel A* Search

In a decentralized parallel A* search, each thread maintains its own local OPEN list to perform the search. Initially, the root node is assembled and processed by a thread. Subsequently generated nodes are distributed among all threads to achieve good load-balancing with low idle time. Especially the load balancing strategy is a deciding factor for computation time, thus many different ideas have been developed. Kumar, Ramesh, and, Rao were among the first to utilize a distributed strategy in [33]. Their approach generates an initial amount of nodes and distributes them to all OPEN lists. The different threads expand the nodes in parallel. To avoid threads working on suboptimal parts of the search tree, since the thread is limited to its own OPEN list, they introduced a communication strategy to share nodes. The goal is to have all threads working on promising sections of the search space. The communication to distribute newly spawned nodes is performed by choosing another thread *randomly* and inserting the node to the target's local OPEN list.

To apply the decentralized parallel A* search to DPLL, each of the k threads maintains a local priority queue as an OPEN list. Moreover, a distribution strategy is required to attain good load balancing. Besides additional priority queues, the overall coordination mechanism is similar to the presented centralized parallel A* search. Therefore, a master thread and k worker threads are used. In the following, the responsibilities of the different threads are explained. Being a variation of parallel A* search, only key differences to the centralized approach are elaborated.

The master thread's function is the search initialization and termination detection. Being an extension of the centralized A* search algorithm, only a few changes have been made for the master thread's logic. For initialization, the root node is inserted into the queue of the first thread. Successive nodes are distributed by the worker threads. Having multiple OPEN lists changes the search termination detection. Firstly, we must test whether all queues in Q are empty and all threads are on a waiting state. Secondly, we test whether all remaining nodes in all OPEN lists are more expensive or at least equal-cost and r solutions have been found. Thus, the search termination detection is more complex, to accommodate the increased number of priority queues.

The worker thread only processes nodes from its local task pool to reduce lock contention. One major difference between the centralized and decentralized strategy is the handling of successor nodes. Every successor node is assigned to one specific thread, determined by a random distribution function.

Comparing properties of the decentralized and centralized parallel A*, the main advantage of the former is the reduced lock contention on the shared OPEN list. With increasing number of threads, the contention on the central data structure surges (synchronization overhead). Hence, the decentralized parallel A* search has the potential to scale better with more processing units. However, distributing nodes across multiple OPEN lists brings disadvantages as well. Firstly, communication overhead is increased, because nodes are exchanged across threads to reduce idle time and the number of suboptimal nodes expanded. Secondly, search overhead is increased compared to the centralized approach. In general, it

can be measured by comparing the expanded nodes in parallel to the number of its sequential implementation. Reason for this growth is that threads potentially work on suboptimal parts of the search tree while waiting for more promising nodes from other threads. In a centralized A* search, all threads work on one data structure, thus promising nodes are expanded first and threads often work on similar sections of the search tree.

5.4 Parallel Cube-and-Conquer

Cube-and-Conquer (C&C) is a two-step approach to solve the SAT instances. First, the problem is divided into a large number of subproblems (cubes) which are subsequently solved in parallel. Regarding thread management, the Master-Worker paradigm is used, comparable to previous parallel A* search algorithms. The master thread is entirely responsible for the first phase, thus generating the desired amount of cubes. Afterwards, the master distributes the cubes among all k available worker threads which process them independently in parallel. The generation of cubes is performed by best-first search until the required amount of cubes is found. In case the set of optimal solutions is found during this phase, the second phase is not performed.

In phase two each worker thread is assigned a list of subproblems (called *cubes*), that have to be processed. For each cube, a local priority queue Q is initialized with the cube, a partial assignment, as the root node. Subsequently, a best-first search is executed for the current cube constructing a sub-tree of the original search space. Thus, nodes are removed from the priority queue iteratively and successor nodes are generated by unit propagation and the choice of the next decision literal resulting in up to two child nodes. Furthermore, to reduce search overhead, cube termination detection is performed. If the head of the priority queue has accumulated costs that exceed the costs of any found solution then the cube can be aborted.

Considering cube-and-conquer, it is a two-phase method that is suited for hard SAT instances. For such CNF formulas, cube-and-conquer approaches can generate between thousand and a million cubes, evaluated extensively in [23] and [5]. The first phase is executed sequentially, followed by parallel processing of subproblems. Distributing cubes in phase two among all worker threads results in a relatively clear search space partitioning and thus little communication overhead. Worker threads only check the currently known minimum cost solution to decide whether a cube can be aborted. Using local information also reduces the synchronization overhead, especially by not using a global OPEN list. Overall, these advantages lead to a simpler approach with less complexity. However, the used method partitions the original search tree into sub-trees. These cubes are processed iteratively, which potentially leads to larger search overhead. The reduced communication and iterative processing of cubes can increase the time threads spend on expanding suboptimal parts of the overall search space. According to our observations, this disadvantage is enhanced in SAT instances that are not very hard, for instance a configuration step within a configuration system. Due to the underlying optimization problem (MinCostConf) and lower instance hardness, the search trees usually have a small height and width. Two reasons are that many paths can be pruned and only few conflicts are encountered. Having hard SAT instances, search trees tend to be of greater height and width, reducing potential search overhead.

In contrast to this, the parallel A* Search uses more complex search space splitting strategies, by extensively sharing data. On the one hand, regarding communication and synchronization overhead, Cube-and-conquer introduces the smallest efforts. However, this benefit leads to the drawback of higher search overhead, especially compared to the centralized A* Search approach. These properties affect the performance depending on the considered

formula’s complexity. Summarized, cube-and-conquer potentially performs well on hard SAT instances, for example for Random SAT and Random 3-SAT benchmarks. Regarding industry cases, the performance is dependent on the size of the constructed search tree. Full evaluation, analysis and comparison of the different methods is presented in Chapter 6.

5.5 Parallel Portfolio Solver

Our first portfolio approach combines several instances of a sequential solver (A* base solver). Each instance has its own set of configuration settings, consisting of several parameters such as:

- The branching heuristic. Different score-based branching heuristics are used which are comparable to the One-Sided (OS) and Two-Sided (TS) Jeroslow-Wang Rules [28]. Furthermore, a second component is added as a parameter which influences the priority of choosing variables that have a positive cost (i.e. are not special variables with zero cost). Due to the underlying optimization problem, it can be beneficial to preferably branch over variables with positive cost to limit the search tree growth.
- Tie-breaking criteria of the OPEN list. We can use different strategies to prioritize nodes with the same f value within the OPEN list, such as comparing the number of unsatisfied clauses.
- Clause Learning. Clause learning is not always beneficial in MinCostConf, due the introduced overhead. Therefore some instances will utilize clause learning and others not.

With various defined parameters, a portfolio can be constructed by combining solvers with different configuration sets. The diversification’s purpose is to have a portfolio with solvers that have little search space overlap to reduce search overhead and the solver’s sensitivity to parameters.

Comparing parallel portfolios to divide-and-conquer approaches, several distinctions can be drawn. A major advantage is the robustness of parallel portfolios against the impact of configuration parameters that SAT solvers generally face. However, considerable disadvantages exist, especially for industrial use cases. The main problem is that the parallel portfolio does not partition the search space into distinct portions, but rather duplicates the Boolean formula for each instance. As a result, the memory consumption increases approximately proportionally to the number of cores. For industry cases and production systems, resources are limited.

5.6 Parallel Portfolio A*

An alternative strategy is to adapt the parallel A* algorithm. The main motivation is to avoid the Boolean formula duplication that is performed by the previously shown parallel portfolio. Instead, the search space is divided but multiple threads work cooperatively on the same instance. To adopt the benefit of being less sensitive to parameter tuning, each thread uses different settings but work on the same formula. Regarding parameters, changing the branching heuristic and related settings such as the “Prefer Cost Literals” can be changed for each processing unit. Hence, when a thread processes a node and has to choose the next decision literal for that path, a thread specific heuristic is utilized. Usually, these parameters have a strong impact on the search behavior with respect to the order of path expansions. This strategy can be applied to both, centralized and decentralized parallel A* search. Despite using varying parameters, no other changes are required for these algorithms.

5.7 Making the Search Deterministic

Most current parallel SAT solvers are not capable of producing stable results, due to their architectures relying on weak synchronization [17]. Nevertheless, reproducibility of a configuration process is a key requirement for a configurator.

The following criteria are used to order configurations as the foundation to achieve reproducible configuration steps: (1) cost of the configuration using the delta-cost function, (2) number of decision literals (tree-depth), (3) number of positive decision literals (left branches), and (4) hash of literals contained in the solution. During parallel search in the search tree, solutions can be found in varying order across several executions. To accommodate this instability, the termination criteria must be adapted to ensure that previously returned solutions are not skipped through early termination in another execution of the same configuration step. Therefore, two alterations are presented in the following, both aiming at ensuring deterministic behavior as well as minimizing the search overhead.

5.7.1 Tree-depth Depending Termination

The first approach exploits the number of decision variables in any obtained solution. As soon as the requested amount of alternative configurations r has been found, the maximum tree-depth (number of decision variables) j is extracted from all found solutions. This indicates the solution that is ordered last among all solutions, thus it can be used to prune unexplored nodes. Consequently, only paths within the search tree that either have lower costs than already known valid assignments or are equal-cost and have a tree-depth less or equal to j are expanded. For subsequently found equal-cost solutions, the maximum j may be updated to reflect the new upper bound, in case it has fewer decision variables. This procedure incrementally reduces the depth of j and consequently the number of paths that may be expanded. As soon as no viable nodes can be processed, because all are more expensive or equal-cost and at deeper levels, the search can be terminated. This strategy ensures that always the same m solutions are returned, because all paths are expanded that contain the m optimal solutions with the smallest number of decision literals.

5.7.2 Advanced Tree-depth Depending Termination

The previous strategy can further be improved by not only utilizing the tree-depth (vertical), but also the position within one level of the search tree (horizontal). For instance, all z optimal solutions share the same tree-depth, but only at most r alternative configurations are requested. Using the presented *tree-depth depending termination*, all z solutions have to be computed, since the sequence of finding them is unstable using multiple threads. The larger z is, the higher is the search overhead (in worst case $z - r$ avoidable nodes are expanded). To circumvent this, the multiple-criteria order of solutions described above is adapted by adding the number of left edges in the path from root to the node n . The number of left edges in a path is abbreviated with “left-branches” in the following. After having found r solutions, the minimum tree-depth as well as the maximum number of left-branches on that level are shared with all threads to prune paths. The value can be updated after having found another solution on a smaller or equal level. In the former case, the value is always updated because the new solution has a smaller tree-depth. In the latter case, the value is only updated if the number of left-branches is larger than the minimum of all currently hold solutions on that level. This strategy reduces the number of expanded nodes, when several solutions are on the same level.

5.7.3 Node Expansion Termination

For the presented cube-and-conquer approach (5.4), a simpler deterministic search termination detection can be used. The distribution of cubes after the first phase consistently assigns a set of work to each thread. Each worker thread processes the cubes in the same order without exchanging them, hence the initial search space partitioning is consistent for repeating configuration steps. This property can be exploited to terminate cubes early, by limiting the amount of nodes each thread is processing after r solutions have been found. For this, the sorting criteria is changed. Having two equal-cost solutions, the one that originated from an earlier cube is preferred.

6 Experimental Evaluation

This Section presents an evaluation and comparison of the developed algorithms in Chapter 5. As a baseline, the commercial product configurator *CAS Merlin* [6] is used. The evaluated algorithms are:

1. *Centralized Parallel A* Search* (CA*): Extension of the sequential A* search using multiple threads on a single OPEN list.
2. *Decentralized Parallel A* Search* (DA*): Extending the A* search by assigning an OPEN list to each thread. Nodes are exchanged among the workers.
3. *Parallel Cube-and-Conquer* (C&C): Two-phase approach. Firstly, the problem is decomposed into subproblems. Afterwards, the cubes are processed in parallel.
4. *Parallel Portfolio* (PP): Combining multiple sequential base solvers with different configurations.
5. *Parallel Portfolio A** (PPA*): Extending parallel A* search by using different settings for each worker thread.

All the algorithms are implemented in Java 11 and executed on the application server WildFly 15. The server has an Intel Core i7-7820HQ CPU and 16 GB RAM. We did experiments with up to 4 threads per SAT solve. This is not an impressive amount when compared to recent work in the area of parallel SAT solvers, nevertheless, we identified 4 as the maximum amount of threads that can be allocated to solving one configuration click (calculation of the next valid configuration) in order to use the available server capacities reasonably and economically in a commercial setting in our case.

We do not compare our algorithms to any existing academic implementations for similar problems in this paper for a few reasons. To facilitate a meaningful evaluation we would need to integrate state-of-the-art academic parallel PBO [42] or PMaxSAT [37] solvers into the configurator. This is difficult, since these solvers are written in C/C++ while our application is in Java. According to our preliminary evaluations, the translation and execution overhead is too big, especially for easy problem instances, which constitute the majority in our application. Additionally, our configuration problems also contain numeric constraints (like in SMT) which cannot be handled by PBO/PMaxSat. We tried integrating a MaxSMT solver but we could not obtain satisfactory results that way either. Not even for the sequential computation, let alone in parallel. Another reasons are the special requirements in our application such as deterministic calculation and enumeration of multiple optimal solutions.

■ **Table 1** Comparison of different rule sets (Boolean formulae). RS1 to RS3 show formulae of industry cases. R3SAT are random 3-SAT instances and RSAT stands for random SAT instances. Ratio describes the clause to variable ratio of the resulting Boolean formulas.

Rule Set	Domain Complexity	# Clauses	# Variables	Ratio	Avg. clause length
RS1	High	17157	8483	2.02	3.98
RS2	Medium	4803	8318	0.58	3.34
RS3	Low	8023	1722	4.66	4.70
R3SAT	–	449	100	4.49	3
RSAT	–	850	100	8.50	5.8

The data used for the experiments consists of various real industry cases as well as random SAT and random 3-SAT instances². The industry cases are divided into three groups (RS1, RS2, RS3) of various complexity, where RS1 is the most complex and RS3 the least complex³. The properties of the benchmarks are given in Table 1. The formula sizes are not as high as what we usually see in say SAT competition benchmarks. On the other hand, one must consider, that the time limit at the SAT competition is 5000 seconds while we want solutions in milliseconds. This is also related to using only 4 threads. Using many threads increases the overhead, which we cannot afford in our setting.

The entire system is tested using automated simulation of configuration processes. Each test for a rule set consists of n configuration steps. Every configuration step is repeated three times to have more accurate mean wall clock times. Therefore, for each rule set $3n$ data points are available. The usage of simulated configuration steps yields a mixture of small and larger configuration changes with varying response times. Considering randomly generated tests, 23 random SAT and 57 random 3-SAT instances are used (hence 80 data points). For each tested algorithm configuration, a warm-up phase is executed for the Java Virtual Machine (JVM). Despite measuring different metrics, the usage of automated tests also ensures the correctness of all implemented algorithms.

Our main goal was to conceptualize approaches that scale well with increasing numbers of processing units. In Table 2 the speedup of the different parallel algorithms is displayed using varying numbers of threads. We report relative speedup rather than absolute run-times, since we believe it is more representative and the standard way to evaluate parallel algorithms.

Largest improvements are attained on RS1 with an overall speedup of 2.5 for CA*. On the easier problem domains RS2 and RS3, the improvements decrease to 2.06 and 1.13. Especially the latter is explained by the numerous short running configuration tasks in RS3 that lead to overhead using CA* search. Results of DA* search show very similar effects, scaling well on complex rule sets but suffering from overhead on simpler ones like RS3. On RS3, the search algorithms perform better with two threads (0.91) than with four (0.90) due to the added synchronization and communication overhead. C&C's results do not show a clear pattern, it scales best on RS1 and RS3, less so on the medium rule set RS2. However, the addition of more threads shows larger diminishing returns. On RS1 the difference between three and four threads is only a speedup of 0.06 compared to the baseline. Furthermore, the table shows that the portfolio solver scales negatively in many cases with the number of processing units. An exception is the less complex rule set RS3. Hence, the scaling issues of the PP solver is related to the complexity of the problem instance.

² The JNH and CBS benchmarks from SATLIB <https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html>

³ The complexity of the groups RS1, RS2, and RS3 is defined based on the average runtime performance of the baseline (single threaded) algorithm on these benchmark sets.

■ **Table 2** Comparison of overall speedup for parallel algorithms using varying numbers of threads. PPA* was evaluated only with 4 threads.

Rule Set		CA*	DA*	C&C	PP	PPA*
RS1	2 Threads	1.63	1.58	1.22	0.91	–
	3 Threads	2.07	1.98	1.37	0.88	–
	4 Threads	2.50	2.08	1.44	0.83	2.40
RS2	2 Threads	1.58	1.40	0.94	0.86	–
	3 Threads	2.03	1.43	1.01	0.72	–
	4 Threads	2.06	1.55	1.05	0.63	1.94
RS3	2 Threads	0.89	0.91	1.12	1.07	–
	3 Threads	1.01	0.90	1.27	1.15	–
	4 Threads	1.13	0.90	1.33	1.24	1.00
R3SAT/RSAT	2 Threads	1.40	1.49	1.91	1.02	–
	3 Threads	1.55	1.86	2.26	0.99	–
	4 Threads	1.72	2.06	2.46	0.97	1.72

All before presented results include the deterministic versions of the algorithms. To determine the approximate impact of the stricter search termination criteria, the best performing algorithm is considered: the centralized parallel A* search. For the evaluation, the most complex benchmark RS1 is used which contains a wide variety of configuration tasks. The impact is measured by the overall speedup of the non-deterministic version over the deterministic version of CA* search. We measured an overhead of approximately 2-3% on RS1. The overall performance cost is 2.4%, if only long running tasks (>100 ms) are considered.

A general observation is the increasing performance improvement on more complex rule sets. A main reason for this is the share of short running configuration tasks on less complex formulas (e.g. RS3). Having instances that can be solved very fast (in less than 50 ms), it is difficult to achieve significant improvements through additional processing units. The added overhead due to initialization and synchronization outweighs the benefits of solving the instance in parallel. Regarding the parallel portfolio, the solver performs worst on many industry cases. Moreover, it partly scales negatively with increasing thread pools. For each sequential base solver, the Boolean formula is duplicated. Performing k separate A* searches, with k being the number of threads, escalates the memory consumption due to the search tree construction. In most tests, the increased load on the system outweighs the benefits of having separate solver configurations that reduce the sensitivity to parameter tuning. Thus, the alternative portfolio approach PPA*, which applies different sets of parameters to parallel A* search, achieves more consistent and better results.

Further data (plots and tables) and discussion about the experimental evaluation are available in the main author's master's thesis [48].

7 Conclusion

Our goal was to find search algorithms that can exploit the capabilities of common multi-core processors while maintaining their completeness and determinism with respect to found solutions. To define the problem occurring in interactive configuration, the MinCostConf problem was introduced which extends the SAT and MinCostSAT problems and belongs

to the class of NP-hard problems. It describes the task of finding minimal-cost solutions given a start-configuration and user wish. Additionally, different custom cost functions were shown to model distinctive behaviors that evaluate configuration changes with respect to the start-configuration.

We presented various parallel algorithms to solve the MinCostConf problem. As a baseline, the existing sequential A* search algorithm was introduced with a custom cost function that prefers to keep prior user selections. Three major strategies for parallel search were implemented. Firstly, two versions of parallel A* search were conceptualized. Secondly, a parallel cube-and-conquer algorithm was presented and lastly, a parallel portfolio approach was proposed. To avoid search space duplication, an alternative was shown which applies portfolio concepts to parallel A* search. The introduction of parallel algorithms for MinCostConf added nondeterminism with regard to the order of found solutions. This has been addressed by designing robust search termination detection strategies which ensure that the search is only terminated when the expected configurations are found.

We performed experiments to compare the implemented algorithms using real industry cases as well as random SAT instances. The achieved speedup varied depending on the rule set, but for critical configuration tasks with longer response times, a consistent speedup between 2 and 3 was attained utilizing 4 worker threads. Lastly, the experiments also showed that deterministic behavior can be achieved with a reasonable overhead.

7.1 Future Work

There are several aspects in this paper that can be extended and further improved upon. Firstly, the evaluation was performed on a limited selection of rule sets using up to four threads. Thus, the presented algorithms can be optimized to utilize a larger number of processing units, although diminishing returns are expected.

Secondly, the presented algorithms are only a subset of possible approaches. Other algorithms that have been used in the literature can be adopted and changed to fit the presented problem. For instance, to limit the memory footprint the iterative deepening A* (IDA*) algorithm can be adapted ([32]). This can also improve the presented parallel portfolio approach which is limited by its resource consumption. Furthermore, the presented MinCostConf problem defines solutions as minimal-cost configurations. In some domains with very complex configuration models, this criteria may be loosened and only good but suboptimal solutions are requested. This could be performed for example with a parallel and deterministic version of beam search.

A reviewer of this paper suggested, that the methods used for robust and super solutions [21, 20, 15] in constraint programming bear a lot of similarity to our methods. In future work we would like to examine these similarities.

Lastly, parallel SAT related algorithms may also be used in other areas of interactive product configuration. Besides a valid configuration, additional information can be calculated, for example the attributes that are not possible to select without changing the pinned attributes. Therefore, these attributes may be grayed out for the user. To accelerate this calculation, a parallel algorithm can be applied. Another area of interest is multi-product configuration. Given several loosely coupled products, a user wish in one product can cause changes in other dependent ones, possibly causing a chain reaction. The calculation of this impact can be performed in parallel, by analyzing the impact of the user with the help of a dependency graph and performing independent sub-configurations in parallel.

References

- 1 Henrik Reif Andersen, Tarik Hadzic, and David Pisinger. Interactive cost configuration over decision diagrams. *Journal of Artificial Intelligence Research*, 37:99–139, 2010.
- 2 Tomáš Balyo, Peter Sanders, and Carsten Sinz. Hordesat: A massively parallel portfolio sat solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 156–172, 2015.
- 3 Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20, 2005.
- 4 Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. *Proceedings of SAT competition*, 2013:1, 2013.
- 5 Csaba Biro, Gergely Kovasznai, Armin Biere, Gábor Kusper, and Gábor Geda. Cube-and-conquer approach for sat solving on grids. In *Annales Mathematicae et Informaticae*, pages 9–21, 2013.
- 6 CAS Software AG. Cas configurator merlin, 2020. URL: <https://www.cas.de/en/products/configurator/cas-configurator-merlin.html>.
- 7 Wahid Chrabakh and Richard Wolski. Gridsat: A chaff-based distributed sat solver for the grid. In *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 37, 2003.
- 8 Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
- 9 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
- 10 Matthew Evett, James Hendler, Ambuj Mahanti, and Dana Nau. Pra*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing*, 25(2):133–143, 1995.
- 11 Andreas Falkner, Alexander Felfernig, and Albert Haag. Recommendation technologies for configurable products. *Ai Magazine*, 32(3):99–108, 2011.
- 12 Alexander Felfernig, Lothar Hotz, Claire Bagley, and Juha Tiuhonen. *Knowledge-based configuration: From research to business cases*. Newnes, 2014.
- 13 Eugene C. Freuder, Chavalit Likitvivatanavong, and Richard J. Wallace. Explanation and implication for configuration problems. In *IJCAI 2001 workshop on configuration*, pages 31–37, 2001.
- 14 Zhaohui Fu and Sharad Malik. Solving the minimum-cost satisfiability problem using sat based branch-and-bound search. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 852–859, 2006.
- 15 Begum Genc, Mohamed Siala, Barry O'Sullivan, and Gilles Simonin. Finding robust solutions to stable marriage. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 631–637. ijcai.org, 2017. doi:10.24963/ijcai.2017/88.
- 16 Tarik Hadzic, Sathiamoorthy Subbarayan, Rune M. Jensen, Henrik R. Andersen, Jesper Møller, and Henrik Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. *small*, 10(1):3, 2004.
- 17 Youssef Hamadi, Said Jabbour, Cédric Piette, and Lakhdar Sais. Deterministic parallel dpll. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):127–132, 2011.
- 18 Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: A parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2010.
- 19 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.
- 20 Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. Robust solutions for constraint satisfaction and optimization. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 186–190. IOS Press, 2004.

- 21 Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. Super solutions in constraint programming. In Jean-Charles Régin and Michel Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*, volume 3011 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2004. doi:10.1007/978-3-540-24664-0_11.
- 22 Marijn Heule and Hans van Maaren. Look-ahead based sat solvers. *Handbook of satisfiability*, 185:155–184, 2009.
- 23 Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In *Haifa Verification Conference*, pages 50–65, 2011.
- 24 KEKIB IRANI and YI-FON SHIH. Parallel a* and ao* algorithms: An optimality criterion and performance evaluation. In *1986 International Conference on Parallel Processing, University Park, PA*, pages 274–277, 1986.
- 25 Mikolas Janota. Do sat solvers make good configurators? In *SPLC (2)*, pages 191–195, 2008.
- 26 Mikoláš Janota. *SAT solving in interactive configuration*. PhD thesis, Citeseer, 2010.
- 27 Mikoláš Janota, Goetz Botterweck, Radu Grigore, and Joao Marques-Silva. How to complete an interactive configuration process? In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 528–539, 2010.
- 28 Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1(1-4):167–187, 1990.
- 29 Yuu Jinnai and Alex Fukunaga. Abstract zobrist hashing: An efficient work distribution method for parallel best-first search. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.
- 30 Bernard Jurkowiak, Chu Min Li, and Gil Utard. A parallelization scheme based on work stealing for a class of sat solvers. *Journal of Automated Reasoning*, 34(1):73–101, 2005.
- 31 Yoshikazu Kobayashi, Akihiro Kishimoto, and Osamu Watanabe. Evaluations of hash distributed a* in optimal sequence alignment. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.
- 32 Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.
- 33 Vipin Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel best-first search of state-space graphs: A summary of results. In *AAAI*, volume 88, pages 122–127, 1988.
- 34 Matthew Lewis, Tobias Schubert, and Bernd Becker. Multithreaded sat solving. In *Asia and South Pacific Design Automation Conference, 2007*, pages 926–931, Piscataway, NJ, 2007. IEEE Operations Center. doi:10.1109/ASPDAC.2007.358108.
- 35 Chu Min Li and Filip Manyá. Maxsat, hard and soft constraints. *Handbook of satisfiability*, 185:613–631, 2009.
- 36 Xiao Yu Li. *Optimization algorithms for the minimum-cost satisfiability problem*. PhD thesis, North Carolina State University, 2004. URL: <https://repository.lib.ncsu.edu/handle/1840.16/4594>.
- 37 Inês Lynce, Vasco M. Manquinho, and Ruben Martins. Parallel maximum satisfiability. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 61–99. Springer, 2018. doi:10.1007/978-3-319-63516-3_3.
- 38 Joao P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
- 39 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.
- 40 Hidetomo Nabeshima and Katsumi Inoue. Reproducible efficient parallel sat solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 123–138. Springer, 2020.

- 41 Mike Phillips, Maxim Likhachev, and Sven Koenig. Pa* se: Parallel a* for slow expansions. In *Twenty-Fourth International Conference on Automated Planning and Scheduling*, 2014.
- 42 Ted K. Ralphs, Yuji Shinano, Timo Berthold, and Thorsten Koch. Parallel solvers for mixed integer linear optimization. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 283–336. Springer, 2018. doi:10.1007/978-3-319-63516-3_8.
- 43 Daniel Sabin and Rainer Weigel. Product configuration frameworks-a survey. *IEEE Intelligent Systems and their applications*, 13(4):42–49, 1998.
- 44 Tobias Schubert, Matthew Lewis, and Bernd Becker. Pamiraxt: Parallel sat solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):203–222, 2010.
- 45 Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin. Pasat – parallel sat-checking with lemma exchange: Implementation and applications. *Electronic Notes in Discrete Mathematics*, 9:205–216, 2001.
- 46 Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Detection of inconsistencies in complex product configuration data using extended propositional sat-checking. In *FLAIRS conference*, pages 645–649, 2001.
- 47 Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Formal methods for the validation of automotive product configuration data. *Ai Edam*, 17(1):75–97, 2003.
- 48 Nils Merlin Ullmann. Parallel sat-solving for product configuration. Master’s thesis, KIT, Karlsruhe, Germany, 2021. doi:10.5281/zenodo.5154040.
- 49 Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. Psato: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4-6):543–560, 1996.

Solution Sampling with Random Table Constraints

Mathieu Vavrille ✉

Laboratoire des Sciences du Numérique de Nantes, 44322 Nantes, France

Charlotte Truchet ✉🏠

Laboratoire des Sciences du Numérique de Nantes, 44322 Nantes, France

Charles Prud'homme ✉

TASC, IMT-Atlantique, LS2N-CNRS, F-44307 Nantes, France

Abstract

Constraint programming provides generic techniques to efficiently solve combinatorial problems. In this paper, we tackle the natural question of using constraint solvers to sample combinatorial problems in a generic way. We propose an algorithm, inspired from Meel's ApproxMC algorithm on SAT, to add hashing constraints to a CP model in order to split the search space into small cells of solutions. By sampling the solutions in the restricted search space, we can randomly generate solutions without revamping the model of the problem. We ensure the randomness by introducing a new family of hashing constraints: randomly generated tables. We implemented this solving method using the constraint solver Choco-solver. The quality of the randomness and the running time of our approach are experimentally compared to a random branching strategy. We show that our approach improves the randomness while being in the same order of magnitude in terms of running time.

2012 ACM Subject Classification Computing methodologies → Randomized search

Keywords and phrases solutions, sampling, table constraint

Digital Object Identifier 10.4230/LIPIcs.CP.2021.56

Supplementary Material *Software (Source Code):*

<https://github.com/MathieuVavrille/tableSampling>

archived at `swb:1:dir:63a03fba176c348c1f9d698bda1b484957b6b5ce`

1 Introduction

Using constraint satisfaction as a core technique, constraint solvers have been enriched with different additional properties, such as optimisation (even with multiple objectives [8]), user preferences [18], diverse solutions [10], robust solutions [9], etc. In this article, we propose a method to sample solutions of a constraint problem, without modifying its model. This work is motivated by many situations where a user wants randomized solutions: to ease user feedback and decision making, to ensure equity (for instance in planning problems), to guarantee solution coverage (for instance in test generation problems).

Currently, a straightforward way to randomly sample solutions with a CP solver is to use `RANDOMVARDOM`; that is, randomly picking a variable and a value as an enumeration strategy. However this strategy does not return uniformly drawn solutions (uniformly within the solution set), and also replaces the strategy that may have been chosen or built for the problem. Our approach is inspired from `UNIGEN` [13], a near-uniform sampling algorithm for SAT, adapted to the CP framework. The idea is to divide the search space by adding random hashing constraints, until only a small, tractable number of solutions remain. No replacement of the strategy is needed and the sampling can be done among these solutions. Our algorithm also features a dichotomic variation which accelerates the whole process.

This algorithm needs to be fed with random hashing constraints. In order to maintain the running time reasonable, we choose to randomly generate table constraints [5], which are implemented in all constraint solvers. We rely on their extensional representation of valid tuples to produce, at cheap cost, a multivariate uniform distribution.



© Mathieu Vavrille, Charlotte Truchet, and Charles Prud'homme;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 56; pp. 56:1–56:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We implemented our proposal on top of Choco-solver [16] and compare it to RANDOMVARDOM on various types of problems. We show that our approach improves, in practice, the randomness compared to RANDOMVARDOM. In addition, the running time increase is limited in practice. This shows that adding a better level of randomization can be done at a low computation cost.

1.1 Related works

As we previously said, our work is inspired from Meel’s work on UNIGEN [13], an algorithm to sample SAT problems. UNIGEN is a two-step algorithm. The first step consists in running APPROXMC, an algorithm for SAT model counting. XOR constraints are added to the model until there are less than a given number of solutions. The solutions are counted within the smaller space marked by the additional constraints, and this number, multiplied by a ratio between the volume of the smaller space and the volume of the real search space, gives a first value for the solution number. Applying this algorithm multiple times gives an approximation of the number of solutions. In the second step, this approximation is used in UNIGEN to sample the problem. The resulting distribution is near the uniform distribution. The idea used in APPROXMC on SAT (divide to count) has also recently been used in a CP context for model counting [15]. Our algorithm uses the same idea of additional constraints to divide the search space, within the CP framework, for solution sampling.

Among the broad literature of SAT solution sampling, we also want to mention [6], which is close to our approach. The authors sample partial solutions, and iteratively extend them by instantiating a fixed number of variables. This approach works well because of the binary domains, but in CP the possible large domains would be an issue. This would force to do more iterations, which in the worst case would lead to an algorithm close to RANDOMVARDOM.

Solution sampling in constraint programming was first studied in [4] and [7], using bayesian networks. These approaches allow to have a uniform sampling, or to choose the distribution of the solutions, but are exponential in the induced width of the constraint graph. This complexity prevents the approach from being used on big instances, and forces the use of approximations. We took the opposite view of designing a fast sampling method, knowing that we would not be able to guarantee the uniformity of the sampling.

Other approaches improve the diversity of solutions, a different task from sampling. It consists in finding solutions far from each other, for a given metric (edit distance for instance). In [10] the model is re-written to search for distant solutions. In [20] solutions are returned in an online fashion; search strategies are designed to search in spaces far from the solutions previously found. Diversification and sampling are linked but remain two very different goals. On one hand, sampling multiple solutions will necessarily return diverse solutions, but it is very unlikely to be the most distant solutions. On the other hand, diversification does not give any guarantee of randomness, as solutions close to other ones may never be returned.

1.2 Outline

Section 2 gives the notations and recalls the definitions that are needed afterwards. Section 3 presents our approach to sample solutions, and section 4 describes our experimental evaluation.

2 Preliminaries

2.1 Constraint programming

In this article we are interested in constraint satisfaction problems (CSPs). A CSP \mathcal{P} is a triple $\langle \mathcal{X}, \mathcal{C}, \mathcal{D} \rangle$ where

- $\mathcal{X} = \{X_1, \dots, X_n\}$ is a set of variables;
- \mathcal{D} is a function associating a domain to every variable;
- \mathcal{C} is a set of constraints, each constraint $C \in \mathcal{C}$ consists of:
 - a tuple of variables called *scope* of the constraint $scp(C) = (X_{i_1}, \dots, X_{i_r})$, where r is the arity of the constraint
 - a relation, i.e. a set of instantiations

$$rel(C) \subseteq \prod_{k=1}^r \mathcal{D}(X_{i_k})$$

A constraint is said to be satisfied if every variable $X_{i_k} \in scp(C)$ is instantiated to a value of its domain $x_{i_k} \in \mathcal{D}(X_{i_k})$, and $(x_{i_1}, \dots, x_{i_r}) \in rel(C)$. The constraints can be defined in extension (called table constraints [5]) by giving explicitly $rel(C)$, or in intension with an expression in a higher level language. For example, the expression $X_1 + X_2 \leq 1$ for X_1, X_2 on domains $\{0, 1\}$, represents $rel(C) = \{(0, 0), (0, 1), (1, 0)\}$.

CSP solving is the search for one, some or all solutions, i.e. assignments of value to every variable such that all the constraints are satisfied. Optimisation problems (COPs) are CSPs where an objective function *obj* to minimise (or maximise) has been added.

Notations

Let a problem $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, and C a constraint, we write $\mathcal{P} \wedge C$ for the CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{C\} \rangle$. We note $Sols(\mathcal{P})$ the set of solutions of problem \mathcal{P} .

In the following, we only consider satisfaction problems. It is also possible to deal with optimisation problems, up to an approximation, by turning them into a satisfaction problem. Let a COP (\mathcal{P}, obj) to minimise (resp. maximise), and let *opt* be the minimum value (resp. maximum) of the objective *obj*. Let $\epsilon \geq 0$, we transform the problem into a CSP $\mathcal{P} \wedge (obj \leq opt + \epsilon)$ (resp. $\mathcal{P} \wedge (obj \geq opt - \epsilon)$). As the gap ϵ increases, the solutions searched will be further from the optimal value.

2.2 Chi squared test

Evaluating the randomness of a system is a hard task because random systems can take surprising values without being biased (for example, a fair coin does, occasionally, land ten times in a row on heads). The chi squared (or χ^2) test allows to compare the result of a random experiment to an expected probability distribution. It comes from a convergence result to the χ^2 law, stated in [14] and recalled here. Let Y be a random variable on a finite set, taking the value k with probability p_k for $1 \leq k \leq d$. Let Y_1, \dots, Y_n be independent random variables of the same law as Y . Let $N_n^{(k)}$ the number of variables $Y_i, 1 \leq i \leq n$ equal to k .

► **Theorem 1** ([14]). *When n tends to infinity, the cumulative distribution function of the random variable*

$$Z_n = \sum_{k=1}^d \frac{\left(N_n^{(k)} - n \cdot p_k\right)^2}{n \cdot p_k}$$

tends to the cumulative distribution function of the law of the χ^2 with $(d - 1)$ degrees of freedom (noted χ_{d-1}^2).

The χ^2 test comes down to randomly picking values by making the assumption that they follow the law of Y , compute the experimental value z_n^{exp} of Z_n , and compute the probability (called p-value)

$$\mathbb{P}(Z_n \geq z_n^{exp}) \approx \mathbb{P}(\chi_{d-1}^2 \geq z_n^{exp})$$

If this probability is close to zero, then, having a more extreme result than the one obtained is very unlikely. It means that the hypothesis under which the experimental values follow the same law as Y can be confidently rejected.

2.3 Random search strategy

The search algorithm for a constraint problem alternates:

- a depth-first search, where the search space is reduced by adding constraints (called decisions), for example, an assignment of a variable to a value in its domain;
- a phase of propagation that checks the satisfiability of the constraints of the CSP.

A natural search strategy to add randomness is the strategy `RANDOMVARDOM` that picks randomly (uniformly) a variable X among all the non instantiated variables, a value $x \in \mathcal{D}(X)$, and applies the decision $X = x$. This strategy has the advantage to be easily implemented in any constraint solver. However it prevents from using an other more efficient exploration strategy, and besides, the distribution of the solutions may be far from uniform. For example on the problem $\mathcal{P} = \langle \{X_1, X_2\}, \{X_1 \mapsto \{0, 1\}, X_2 \mapsto \{0, 1\}\}, \{X_1 + X_2 > 0\} \rangle$, let s the solution returned by a solver configured to use `RANDOMVARDOM`, then we have,

$$\begin{aligned} \mathbb{P}(s = \{X_1 \mapsto 0, X_2 \mapsto 1\}) &= \frac{3}{8} \\ \mathbb{P}(s = \{X_1 \mapsto 1, X_2 \mapsto 0\}) &= \frac{3}{8} \\ \mathbb{P}(s = \{X_1 \mapsto 1, X_2 \mapsto 1\}) &= \frac{1}{4} \end{aligned}$$

The implications on the running time to find solutions and the quality of the randomness on different problems are discussed in section 4.

3 New sampling approach

We present here a new approach to sample solutions. This approach is twofold: first we present a way to generate random tables, and we then present an algorithm to sample solutions using these generated constraints.

■ **Algorithm 1** Random table constraint generation algorithm.

```

1 Function RANDOMTABLE( $\mathcal{P}, v, p$ )
   Data: A CSP  $\mathcal{P} = \langle \{X_1, \dots, X_n\}, \mathcal{D}, \mathcal{C} \rangle$ ,  $v > 0$ ,  $0 < p < 1$ 
   Result: A random table constraint
2    $T \leftarrow \{\}$ ;
3    $i_1, \dots, i_v \leftarrow \text{GETINDICES}(\mathcal{P}, v)$ ;
4   foreach  $(x_{i_1}, \dots, x_{i_v}) \in \prod_{k=1}^v \mathcal{D}(X_{i_k})$  do
5     if RANDOM()  $< p$  then
6        $T.add((x_{i_1}, \dots, x_{i_v}))$ ;
7   return TABLE( $(X_{i_1}, \dots, X_{i_v}), T$ );

```

3.1 Random table constraints

The algorithm to generate random table constraints is presented in Algorithm 1. We suppose available the functions *RANDOM*() that returns a random floating point number between 0 and 1, *GETINDICES*(\mathcal{P}, v) that returns v indices i_1, \dots, i_v such that $|\mathcal{D}(X_{i_k})| \neq 1$, $1 \leq k \leq v$ (if there are less than v such indices, they are all returned), and *TABLE*(\mathcal{X}', T) that creates a table constraint C such that $scp(C) = \mathcal{X}'$ and $rel(C) = T$. The two parameters of the algorithm are: v the number of variables in the table, and p the probability to add a tuple in the table. The algorithm first randomly chooses v variables among the variables whose domains are not reduced to a singleton, and then runs through all the instantiations of these v variables and adds each instantiation in the table with probability p .

The goal of these tables is to restrict the solution space to a smaller sub-space. The following theorem shows that in average, the number of solutions of the problem is reduced by a factor p .

► **Theorem 2.** *Let \mathcal{P} be a CSP, and T a table constraint randomly generated with probability p . Then*

$$\mathbb{E}(|Sols(\mathcal{P} \wedge T)|) = p|Sols(\mathcal{P})|$$

Proof. For $\sigma \in Sols(\mathcal{P})$, let γ_σ a random variable equal to 1 if and only if $\sigma \in Sols(\mathcal{P} \wedge T)$. $\mathbb{P}(\gamma_\sigma)$ is the probability that σ satisfies T . Let X_{i_1}, \dots, X_{i_v} the variables chosen in T . Each instantiation of these variables has been added in the table with probability p , including the instantiation $(\sigma(X_{i_1}), \dots, \sigma(X_{i_v}))$. It means that σ satisfies the table constraint T with probability p . We then have $p = \mathbb{P}(\gamma_\sigma = 1) = \mathbb{E}(\gamma_\sigma)$. It follows:

$$\begin{aligned}
 \mathbb{E}(|Sols(\mathcal{P} \wedge T)|) &= \mathbb{E}\left(\sum_{\sigma \in Sols(\mathcal{P})} \gamma_\sigma\right) \\
 &= \sum_{\sigma \in Sols(\mathcal{P})} \mathbb{E}(\gamma_\sigma) \\
 &= \sum_{\sigma \in Sols(\mathcal{P})} p \\
 &= p|Sols(\mathcal{P})|
 \end{aligned}$$

◀

The purpose of Theorem 2 is the following: by adding table constraints, we decrease the size of the solution set, and we can control how much, in average.

■ **Algorithm 2** Sampling algorithm by adding table constraints.

```

1 Function TABLESAMPLING( $\mathcal{P}, \kappa, v, p$ )
  Data: A CSP  $\mathcal{P}, \kappa \geq 2, v > 0, 0 < p < 1$ 
  Result: A solution to the problem  $P$ 
2    $S \leftarrow \text{FINDSOLUTIONS}(\mathcal{P}, \kappa);$ 
3   if  $|S| = 0$  then
4     return "No solution";
5   while  $|S| = 0 \vee |S| = \kappa$  do
6      $T \leftarrow \text{RANDOMTABLE}(\mathcal{P}, v, p);$ 
7      $S \leftarrow \text{FINDSOLUTIONS}(\mathcal{P} \wedge T, \kappa);$ 
8     if  $|S| \neq 0$  then
9        $\mathcal{P} \leftarrow \mathcal{P} \wedge T;$ 
10  return RANDELEMENT( $S$ );

```

3.2 Sampling algorithm

First, the auxiliary functions used in the sampling algorithm are presented. The first one is RANDELEMENT(S) that returns a random element taken uniformly in S . The second function is FINDSOLUTIONS(\mathcal{P}, s) that enumerates the solutions of \mathcal{P} until s solutions have been found, and returns them. Notice that, if this function returns s solutions, then $|\text{Sols}(\mathcal{P})| \geq s$, and if it returns less than s solutions then all the solutions have been found. The depth first search in constraint solvers makes the implementation of such a function easy.

The sampling algorithm works in the following manner: table constraints are added to the problem to reduce the number of solutions. When there are less solutions than a given pivot value, a solution is randomly returned among the remaining solutions. The algorithm is presented in details in Algorithm 2. A value κ for the pivot is chosen to bound the number of solutions enumerated in the intermediate problems, as well as the number of variables per table v and the probability p to add a tuple in the table.

The algorithm first enumerates κ solutions and immediately stops if there are no solutions, or less than κ solutions. If the problem has more than κ solutions a new table constraint is randomly generated. If the problem with this constraint still has solutions, the constraint is definitively added to the problem. The algorithm stops when there are less than κ solutions. Finally, a solution is randomly chosen from all the solutions remaining, and returned.

3.2.1 Proof of termination

When creating random algorithms, one has to be particularly careful about the termination. We show here that Algorithm 2 terminates with probability 1. A discussion about the experimental behaviour is done in section 4.3.

We fix values for $\kappa \geq 2, v > 0$ and $0 < p < 1$. The case of the initial problem not being satisfiable is caught at the beginning of the algorithm (line 3).

The following lemmas shows that there always exists a table that reduces the number of solutions of the problem without making it inconsistent, and this table is chosen with a non-zero probability. Without loss of generality, we suppose that there are always v variables in the tables. If less than v variables are not instantiated, we pick some of the already instantiated variables and use their current values to complete the instantiations.

► **Lemma 3.** *Let \mathcal{P} be a problem with at least two solutions. In our framework, there exists a random table constraint T_0 such that*

$$0 < |\text{Sols}(\mathcal{P} \wedge T_0)| < |\text{Sols}(\mathcal{P})|$$

Proof. Let σ_1 and σ_2 two distinct solutions of the problem \mathcal{P} . Let i_1 such that $\sigma_1(X_{i_1}) \neq \sigma_2(X_{i_1})$. Let i_2, \dots, i_v other indices such that $|\mathcal{D}(X_{i_k})| \neq 1, 2 \leq k \leq v$. Let us define the table

$$T_0 = \text{TABLE}((X_{i_1}, \dots, X_{i_v}), \{(\sigma_1(X_{i_1}), \dots, \sigma_1(X_{i_v}))\})$$

Then $\sigma_1 \in \text{Sols}(\mathcal{P} \wedge T_0)$ so $\text{Sols}(\mathcal{P} \wedge T_0) \neq \emptyset$, and $\sigma_2 \notin \text{Sols}(\mathcal{P} \wedge T_0)$ so $\text{Sols}(\mathcal{P} \wedge T_0) \neq \text{Sols}(\mathcal{P})$. Since we add a constraint to \mathcal{P} to build $\mathcal{P} \wedge T_0$, we have $\text{Sols}(\mathcal{P} \wedge T_0) \subseteq \text{Sols}(\mathcal{P})$, thus $0 < |\text{Sols}(\mathcal{P} \wedge T_0)| < |\text{Sols}(\mathcal{P})|$. ◀

► **Lemma 4.** *There exists a constant $\rho > 0$, depending only on the initial problem, such that, for T a randomly chosen table constraint with v variables:*

$$\mathbb{P}(0 < |\text{Sols}(\mathcal{P} \wedge T)| < |\text{Sols}(\mathcal{P})|) \geq \rho$$

Proof. We know from Lemma 3 that there is at least one table constraint T_0 such that $0 < |\text{Sols}(\mathcal{P} \wedge T_0)| < |\text{Sols}(\mathcal{P})|$. Let d be the maximum size of the domains of the initial problem. We bound the probability of $\text{RANDOMTABLE}(v, p)$ to pick exactly T_0 (up to ordering of the scope of the constraints). Let T be a random table returned by $\text{RANDOMTABLE}(v, p)$. We want to bound

$$\begin{aligned} \mathbb{P}(T = T_0) &= \mathbb{P}(\text{scp}(T) = \text{scp}(T_0) \wedge \text{rel}(T) = \text{rel}(T_0)) \\ &= \mathbb{P}(\text{scp}(T) = \text{scp}(T_0)) \cdot \mathbb{P}(\text{rel}(T) = \text{rel}(T_0) | \text{scp}(T) = \text{scp}(T_0)) \end{aligned}$$

There is $\binom{n}{v}$ ways to choose the v variables appearing in the table (the ordering does not matter), so $\mathbb{P}(\text{scp}(T) = \text{scp}(T_0)) = 1/\binom{n}{v}$. Let k the number of tuples in T_0 . There are at most d^v possible tuples in total. The probability to choose every tuple in T_0 and not the others is $p^k(1-p)^{d^v-k}$. As $k \leq d^v$ we have the lower bound $\mathbb{P}(\text{rel}(T) = \text{rel}(T_0) | \text{scp}(T) = \text{scp}(T_0)) \geq p^k(1-p)^{d^v-k} \geq \min(p, 1-p)^{d^k}$. By defining $\rho = \frac{1}{\binom{n}{v}} \min(p, 1-p)^{d^k}$ we have the desired bound, and $\rho > 0$ because $0 < p < 1$. ◀

We proved that during an iteration of the loop, there is a probability strictly greater than 0 to remove solutions without making the problem inconsistent. We can now prove that the algorithm terminates with probability 1. The proof is similar to the one showing that tossing a fair coin, until tails comes up, ends with probability 1.

► **Theorem 5.** *Algorithm 2 terminates with probability 1.*

Proof. For some $k > |\text{Sols}(\mathcal{P})| - \kappa$, we want to find an upper bound of the probability that the algorithm has not stopped after k iterations. In some cases, an iteration reduces the number of solutions to the problem without making it inconsistent. There can be at most $|\text{Sols}(\mathcal{P})| - \kappa$ such iterations, because the algorithm stops if there is less than κ solutions (condition of the while line 5). For the other iterations, the condition of the while loop ensures that: either the (most recently added) table made the problem inconsistent, or it did not reduce the number of solutions. The probability for this to happen is less than $1 - \rho$, as stated in Lemma 4. Thus, the probability that, after k iterations, the algorithm did not stop, is less than $(1 - \rho)^{k - |\text{Sols}(\mathcal{P})| + \kappa}$. This probability tends to zero when k tends to infinity. This proves that the algorithm stops with probability 1. ◀

■ **Algorithm 3** Algorithm of dichotomic addition of tables.

```

1 Function DICHOTOMICTABLEADDITION( $\mathcal{P}, nbTables, \kappa, v, p$ )
   Data: A CSP  $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ ,  $nbTables > 0, \kappa \geq 2, v > 0, 0 < p < 1$ 
   Result:  $\mathcal{P}$  with the new table constraints, and the number of added tables
2    $\mathcal{T} \leftarrow$  array of size  $nbTables$ ;
3   for  $i = 0$  to  $nbTables - 1$  do
4      $\mathcal{T}[i] \leftarrow \text{RANDOMTABLE}(\mathcal{P}, v, p)$ ;
5    $S \leftarrow \text{FINDSOLUTIONS}(\mathcal{P} \wedge \bigwedge_{t \in \mathcal{T}} t, \kappa)$ ;
6   while  $|S| = 0 \wedge |\mathcal{T}| > 0$  do
7      $\mathcal{T} \leftarrow \mathcal{T}[0 : |\mathcal{T}|/2]$ ;
8      $S \leftarrow \text{FINDSOLUTIONS}(\mathcal{P} \wedge \bigwedge_{t \in \mathcal{T}} t, \kappa)$ ;
9   return  $\mathcal{P} \wedge \bigwedge_{t \in \mathcal{T}} t, |\mathcal{T}|$ ;

```

This proof is built with an upper bound, and considers the worst case (when solutions are eliminated slowly), but in practice there is more than one table satisfying Lemma 3. The solving time will be studied in practice in Section 4.5.

3.3 Dichotomic table addition

It is possible to improve the efficiency of the algorithm by increasing the number of tables added at each step. At the beginning of the search, a table has a small probability to make the problem inconsistent, so it is wiser to add more constraints to reduce the number of calls to the solver. This algorithm is inspired from the unbounded dichotomic search: first, find i such that the value we want to guess is between 2^i and 2^{i+1} , and then, run a usual dichotomic search between 2^i and 2^{i+1} .

The algorithm of dichotomic table addition is presented in Algorithm 3, and should replace lines 6 to 9 of Algorithm 2. Let τ be the number of tables added at the previous step, we choose $nbTables = 1$ if $\tau = 0$ or $nbTables = 2\tau$ otherwise, and $nbTables$ tables are generated and stored in an array \mathcal{T} . The algorithm then enumerates κ solutions to the problem where the tables in \mathcal{T} have been added. If there is no solutions, it deletes half of the constraints in \mathcal{T} . The procedure stops when the problem is satisfiable or $|\mathcal{T}| = 0$.

Theorem 5 can be extended to the case of the dichotomic table addition, because line 6 in Algorithm 3 ensures that the problem does not become inconsistent.

3.4 Discussion

In this section, we discuss the algorithmic choices we have made in Algorithm 2.

3.4.1 Quality of the division by the tables

In the proof of Theorem 2 the random variables $(\gamma_\sigma)_{\sigma \in \text{Sols}(\mathcal{P})}$ are not independent. For example, let σ_1 and σ_2 two solutions to the problem that only differ on one variable X , then

$$\mathbb{P}(\gamma_{\sigma_2} = 1 | \gamma_{\sigma_1} = 1) = \mathbb{P}(X \in \text{scp}(T)) \cdot p + \mathbb{P}(X \notin \text{scp}(T)) \quad (1)$$

Indeed, if the variable X appears in T , then σ_2 will be kept with probability p , but if X is out of the scope of T , then σ_2 will always be kept. If the table does not have all the variables in its scope, then it may not split the clusters of solutions which take the same values on

multiple variables. This notion of independence is central in the approaches of Kuldeep Meel [13] to show the uniformity of the sampling. Contrarily to this approach, our sampling is not uniform. We choose to have tables of a controlled size for sake of efficiency.

Formula 1 showing the non independence also shows that increasing the number of variables in the table makes the random variables γ_σ more independent, hence the whole sampling process closer to uniformity. Tables containing all the variables of the problem would make the random variables γ_σ fully independent, since in this case $\mathbb{P}(X \notin \text{scp}(T)) = 0$. This would give a theoretical guarantee on the sampling, but is impossible to generate in practice.

3.4.2 Influence of the parameters

Three parameters have to be chosen to run the algorithm. We can already estimate the impact of the parameters on the running time and on the quality of the randomness.

- As seen in the previous subsection, increasing the number of variables in the tables should improve the randomness, but will also exponentially increase the number of tuples in the table, with a negative impact on the running time.
- Reducing the probability of adding a tuple in a table should improve the running time because the tables will be smaller, so the propagation will be faster, and the number of added tables will be lower because the problem will be more quickly reduced.
- The impact of the pivot on the running time is unclear. Having a higher pivot means that more solutions have to be enumerated at each step, but it also means that the algorithm will stop after adding fewer constraints.

These hypotheses will be experimentally verified in section 4.

4 Experiments

This section presents the experiments done to test our approach. First, we evaluate the behaviour in term of randomness. Then, we compare the running time of our approach to the strategy RANDOMVARDOM. The code is available online ¹, along with all the scripts to generate the figures presented in this article.

4.1 Implementation

The implementation has been done in Java 11 using the constraint solver `choco-solver` version 4.10.6 [16]. It is possible to create a model directly in Java using the `choco-solver` library, or by giving a file in the FlatZinc format (generated from the MiniZinc format). Unless the FlatZinc file defines a strategy, the solver default strategy is used (`dom/Wdeg` [3] and `lastConflict` [12]).

A technical improvement has been done, by adding a propagation step before the generation of a table (before line 6 of algorithm 2). This avoids enumerating some tuples that would be immediately deleted by propagation.

In the following, the algorithm used is TABLESAMPLING with DICHOTOMICTABLEADDITION.

¹ <https://github.com/MathieuVavrille/tableSampling>

Management of randomness

The random number generator used is the default one in Java : `java.util.Random`. This generator uses a formula of linear congruence to modify a seed on 48 bits given as input. The Java documentations points to [11], see section 3.2.1 for more information. This randomness generator has flaws (notably a period of 2^{48}), but is sufficient to our needs (as shown in [2]).

The implementation uses a single instance of the random number generator, passed as argument to every function needing it. This avoids a non independence behaviour due to a bad generation of random seeds.

4.2 Problems

The approach is independent from the constraints of the problem, so we were able to apply it on four different problems, including three real life problems. We present here the models and their characteristics.

N-queens

The first problem is the *N*-queens problem, which consists of placing *N* queens on an $N \times N$ chessboard such that no queen attacks an other one (queens attack in every 8 directions, as far as possible). We implemented it with the usual model with *N* variables with domain $[1, N]$, an `all_different` constraint and inequality binary constraints (for diagonal attacks).

Renault Mégane Configuration

This is the problem of configurations of the Renault Mégane introduced in [1] and already used in [10] for the search of diverse solutions. There are 101 variables with domains containing up to 43 values, and the 113 constraints are modeled by table constraints, the majority of them are non binary. This problem is loosely constrained, hence having more than $1.4 \cdot 10^{12}$ solutions.

On Call Rostering

This problem models the system of duty, notably used by healthcare workers. This problem is available in the MiniZinc benchmarks ² and contains different constraint types, such as linear constraints, global constraints `count`, absolute values, implications and table constraints. Many datasets are available but only the smallest (`4s-10d.dzn`) has been used here. It is an optimisation problem (minimization), so it was necessary to transform this problem into a satisfaction problem by bounding the objective function. The optimal value is 1:

- There are 136 solutions with $obj \leq 1$
- There are 2,099 solutions with $obj \leq 2$
- There are more than 10,000 solutions with $obj \leq 3$

By randomly sampling the solutions, the solver can be used as a tool to help people creating plannings to decide on (giving them multiple plannings to compare), and brings a form of equity between the workers. Indeed, oriented search methods could favor some workers at the expenses of others.

² <https://github.com/MiniZinc/minizinc-benchmarks/tree/master/on-call-rostering>

Feature Models

These are problems of software management, helping to decide on the order of implementation of software features. The problem is specified in the MiniZinc format in [17] using the data in [19]. Again, it is an optimisation problem (maximization), the optimal value is 20,222. We add the constraint $obj \geq 17,738$ to make it a satisfaction problem with 95 solutions.

4.3 Experimental behaviour

We discuss here the experimental behaviour of TABLESAMPLING. In practice, we see that most of the computations are done at the beginning of the algorithm. At the beginning, most of the tables added do not make the problem inconsistent. We really benefit from the dichotomic addition of tables. Most of the time is spent finding κ solutions, because as only few tables are added to the problem, the search space is not reduced much, so searching for solutions is not sped up.

At the end of the algorithm, when there are only few solutions remaining (but still more than κ), there is a higher probability to make the problem inconsistent by adding a table. Actually, this is not an issue, because it becomes really fast to find the solutions (or to prove that the problem is inconsistent). This is due to the fact that all the tables added previously really restrict the search space and are quickly propagated.

4.4 Quality of the randomness

The first goal of the experiments is to evaluate the quality of the randomness, i.e. knowing if the solutions are sampled randomly and uniformly. The following results show that even if the solutions are not sampled uniformly, the approach using table constraints is *more uniform* than the strategy RANDOMVARDOM.

4.4.1 Evaluation of the uniformity

To have a numerical measure of the uniformity of the sampling, we used the χ^2 test. Knowing the number $nbSols$ of solutions of a problem (and numbering these solutions), $nbSamples$ samples are drawn and the number of occurrences $nbOcc_i$ of each solution $i \in \{1, \dots, nbSols\}$ is counted. We compute the value of the variable

$$z_{exp} = \sum_{k=1}^{nbSols} \frac{(nbOcc_k - nbSamples/nbSols)^2}{nbSamples/nbSols}$$

and then the p-value of the test³ (i.e. the probability that the χ^2 law takes a more extreme value than z_{exp}). This p-value gives a numerical value of the quality of the randomness. More specifically, a large number of samples are drawn (more than the number of solutions) and the evolution of the p-value depending on the number of samples is plotted. In our case, as the sampling is not uniform, the p-value will tend to 0 when the number of samples increases, but we remark that the sampling using tables has a p-value which tends slower to 0 than the default sampling using RANDOMVARDOM.

³ We use the library “Apache Commons Mathematics Library” (<https://commons.apache.org/proper/commons-math/>) for the probability computations

56:12 Solution Sampling with Random Table Constraints

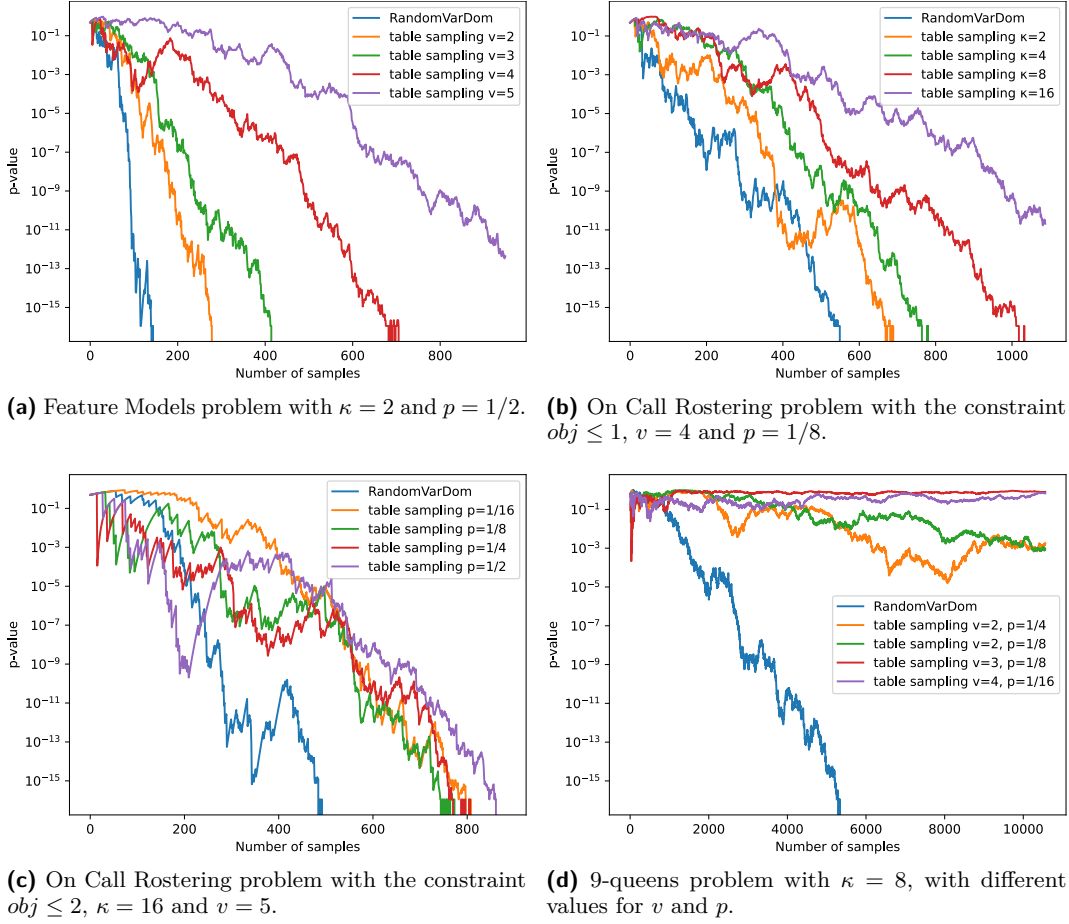


Figure 1 Evolution of the p-value on different problems, with different parameters. Plotting of graphs using the table sampling and RANDOMVARDOM.

To do this test we need to know the number of solutions $nbSols$ and to sample multiple times $nbSols$ solutions, so the evaluation of the randomness can only be done on small instances. These instances are the 9-queens (352 solutions), the Feature Models with the constraint $obj \geq 17,738$ (95 solutions) and the On Call Rostering problem with the constraints $obj \leq 1$ and $obj \leq 2$ (136 and 2,099 solutions).

We want to evaluate the impact of the evolution of a parameter (number of variables, pivot, or probability) on the randomness of the algorithm of sampling by tables. To do so, in the figures that follow, we plot the evolution of the p-value for the strategy RANDOMVARDOM, as well as for different values of parameters for the sampling by tables, by changing one parameter at a time. The legend gives the parameters associated to each execution (v for the number of variables, κ for the pivot and p for the probability).

► **Remark 6.** The figures show the p-value in a logarithmic scale, because it tends to 0. Moreover, as the computations are done using floating point representation, a p-value smaller than 10^{-16} will be considered to be equal to 0.

4.4.2 Impact of the number of variables

We first vary the number of variables used in the generated tables. Fig. 1a shows the evolution of the p-value on the Feature Models problem with parameters $\kappa = 2$ and $p = 1/2$. We remark that increasing the number of variables in the table makes the p-value tend to zero slower, meaning that the sampling is closer to uniformity. As we remarked earlier, this is due to a better independence in the probability that two solutions will satisfy the table constraints (see section 3.4.1).

4.4.3 Impact of the pivot

On Fig. 1b, we vary the pivot on the problem On Call Rostering, with the constraint $obj \leq 1$ and parameters $v = 4$ and $p = 1/8$. Here we observe that increasing the pivot improves the randomness. Indeed, when the pivot is high, at each step a lot of solutions are enumerated, and in the end a random solution will be picked among a lot of other solutions, leading to a better randomness. The extreme case is the perfect (but costly) sampling process, when the pivot is higher than the number of solutions: the algorithm is then simply an enumeration of all the solutions and returns a random solution.

4.4.4 Impact of the probability

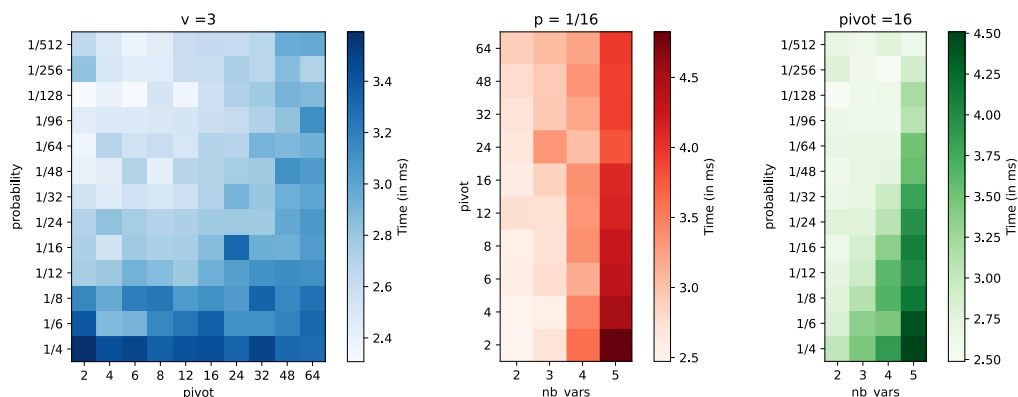
Fig. 1c shows the p-value for different values of the table probability p , on the On Call Rostering problem, with the constraint $obj \leq 2$ and the parameters $\kappa = 16$ and $v = 5$. There is no clear influence of the probability to add tuples in the tables to the quality of the randomness. This allows, when choosing the probability, to focus on the running time (as we will see in section 4.5.1).

4.4.5 Quality of the randomness

The last test was done on the 9-queens problem. Fig. 1d shows the evolution of the p-value with $\kappa = 8$ and different values for v and p . On this particular problem (and on the N -queens problem for any N), the sampling using the table constrains is actually uniform in practice (the p-value tends to 1). As we already said, we have no theoretical guarantee with our approach, but it seems that, on some problems, we may achieve uniformity in practice. We believe that it is due to the structure of the solution space, because the N -queens problem is a very structured problem with many symmetries. Thus, it is likely that the solutions are properly spread on the search space.

4.4.6 Comparison with RandomVarDom

On every graph, we also plotted the evolution of the p-value for the sampling using the search strategy RANDOMVARDOM. We see on the three first graphs that our approach tends to zero after significantly more samples than RANDOMVARDOM. Using TABLESAMPLING makes the sampling more uniform. For example, after sampling 50 solutions, the p-value of RANDOMVARDOM is 0.004, but the p-value of TABLESAMPLING (with parameters $\kappa = 2, v = 2, p = 1/2$) is 0.1. On the N -queens problem, RANDOMVARDOM is not uniform but our approach is, meaning that it really improved the quality of the randomness.



■ **Figure 2** Heat maps of the time to sample one solution, by fixing different parameters, on the On Call Rostering problem with the constraint $obj \leq 3$.

4.5 Running time

The evaluation of the running time is done in two parts. The parameters may have an important impact on the running time: we experimentally investigate in Section 4.5.1 which parameters have a positive impact. Then, we compare the running time of our method and that of using RANDOMVARDOM as a randomization strategy.

In the section, we use problems with more than 10,000 solutions. This eliminates the Feature Models problem, which has too few solutions to provide a meaningful statistical test. For each method, 50 samples are done, in order to average the running time.

4.5.1 Impact of the parameters

To show the impact of the parameters on the time to sample one solution, we fix one parameter, and vary the two other parameters, to plot a heat map of the time, in function of the two parameters. Fig. 2 shows the three heat maps obtained by fixing the number of variables, the pivot or the probability. The hypotheses done in section 3.4.2 are verified here experimentally:

- decreasing the number of variables in the tables, decreases the running time;
- decreasing the probability of adding a tuple in the table decreases the running time.

As we previously saw, increasing the number of variables improves the randomness. There is a compromise to do between the running time (having few variables in the tables) and the quality of the randomness (having many variables in the tables).

We also saw that the probability to add a tuple in the tables does not have a clear impact on the quality of the randomness, so it is best to choose small probabilities to have smaller tables, hence giving a faster propagation, as well as fewer tables added during the computations.

The number of variables in the tables should be chosen as a trade-off between the desired quality of randomness and the running time. It will depend on the application: instances with big domains may require smaller v not to have too big tables (for example, $v = 4$ for domains of size 100 would have to enumerate 10^8 tuples). From our experiments, we suggest as a baseline to use the parameters $\kappa = 16$ and $p = 1/32$.

■ **Table 1** Comparison of the time to sample a solution between RANDOMVARDOM and table sampling.

Problem	RANDOM- VARDOM	Table sampling				Ratio
		v	κ	p	Time	
Renault Mégane Configuration	32 ms	2	16	1/16	55 ms	1.7
				1/32	59 ms	1.8
			32	1/16	89 ms	2.8
				1/32	86 ms	2.7
		3	16	1/16	74 ms	2.3
				1/32	67 ms	2.1
			32	1/16	83 ms	2.6
				1/32	65 ms	2.0
On Call Rostering	38 ms	2	16	1/16	14 ms	0.36
				1/32	14 ms	0.38
			32	1/16	15 ms	0.4
				1/32	18 ms	0.46
		3	16	1/16	18 ms	0.47
				1/32	15 ms	0.4
			32	1/16	19 ms	0.5
				1/32	17 ms	0.44
12-queens	2 ms	2	16	1/16	4 ms	2.4
				1/32	4 ms	2.3
			32	1/16	7 ms	4.1
				1/32	7 ms	3.9
		3	16	1/16	10 ms	5.6
				1/32	8 ms	4.3
			32	1/16	12 ms	6.9
				1/32	11 ms	6.0

4.5.2 Comparison to RandomVarDom

In this Section, we compare the running time to RANDOMVARDOM. Table 1 shows the running time to sample one solution for 8 different sets of parameters of table sampling. To take into account the variability of the solving time, we measure it on 50 samplings, and report the average time to get one sample. The ratio between the time of TABLESAMPLING and the time of RANDOMVARDOM is also given.

We showed that the quality of the randomness of TABLESAMPLING is better than the one of RANDOMVARDOM, and we would expect to pay a price in running time in return. But, in practice, the running times are still within the same order of magnitude. On the On Call Rostering problem, it is even two to three time faster to use TABLESAMPLING instead of RANDOMVARDOM. On the other two problems, RANDOMVARDOM is faster. This behaviour can be explained quite easily: the On Call Rostering problem is very sparse, and there are many values in the domains of the variables that do not lead to solutions. Thus, the RANDOMVARDOM strategy provokes a lot of fails during the search, because it often picks values that do not appear in solutions. In comparison, on the problems of Renault Mégane Configuration and N -queens, a lot of values in the domains of the variables may lead to solutions, so the probability of failing because of a bad choice is low. Our approach also allows to use a different and more powerful strategy, since it can be combined with any search strategy. We can thus take advantage of all the progress made in the design of search strategies.

5 Conclusion

We presented an algorithm using table constraints to randomly sample solutions of a problem. We improved this algorithm by increasing the number of tables added at each step. We experimented our approach on four different problems, involving different types of constraints and different domains of variables. We showed that the sampling is closer to uniformity than a sampling using the search strategy RANDOMVARDOM. Even with this improved randomness, the running time remains comparable to RANDOMVARDOM. Our approach uses the solver as a black box, hence can be applied to a wide range of problems.

In the future, we plan to study structured problems, and investigate how to improve the sampling using the structure of the problem. Optimisations problems are also an other research direction. Instead of artificially turning them into satisfaction problems, we plan to use the samples previously found to directly search for close to optimal solutions.

References

- 1 Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic cps—application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.
- 2 Eric Bach. Realistic analysis of some randomized algorithms. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 453–461, 1987.
- 3 Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.
- 4 Rina Dechter, Kaley Kask, Eyal Bin, Roy Emek, et al. Generating random solutions for constraint satisfaction problems. In *AAAI/IAAI*, pages 15–21, 2002.
- 5 Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In *International Conference on Principles and Practice of Constraint Programming*, pages 207–223. Springer, 2016.
- 6 Stefano Ermon, Carla P Gomes, and Bart Selman. Uniform solution sampling using a constraint solver as an oracle. *arXiv preprint*, 2012. [arXiv:1210.4861](https://arxiv.org/abs/1210.4861).
- 7 Vibhav Gogate and Rina Dechter. A new algorithm for sampling csp solutions uniformly at random. In *International Conference on Principles and Practice of Constraint Programming*, pages 711–715. Springer, 2006.
- 8 Renaud Hartert and Pierre Schaus. A support-based algorithm for the bi-objective pareto constraint. 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 2674–2679. AAAI Press, 2014. URL: <http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8337>.
- 9 E. Hebrard. *Robust Solutions for Constraint Satisfaction and Optimisation under Uncertainty*. PhD thesis, University of New South Wales, 2006.
- 10 Emmanuel Hebrard, Brahim Hnich, Barry O’Sullivan, and Toby Walsh. Finding diverse and similar solutions in constraint programming. In *AAAI*, volume 5, pages 372–377, 2005.
- 11 Donald E Knuth. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.
- 12 Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Last conflict based reasoning. 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 133–137. IOS Press, 2006. URL: <http://www.booksonline.iospress.nl/Content/View.aspx?piid=1661>.

- 13 Kuldeep S Meel. Constrained counting and sampling: bridging the gap between theory and practice. *arXiv preprint*, 2018. [arXiv:1806.02239](#).
- 14 Karl Pearson. X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, 1900.
- 15 Gilles Pesant, Kuldeep S. Meel, and Mahshid Mohammadalitajrishi. On the usefulness of linear modular arithmetic in constraint programming. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 18th International Conference, CPAIOR 2021, Vienna, Austria, September 21-24, 2021, Proceedings*, Lecture Notes in Computer Science. Springer, 2021.
- 16 Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Solver Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. URL: <http://www.choco-solver.org>.
- 17 Björn Regnell and Krzysztof Kuchcinski. Exploring software product management decision problems with constraint solving-opportunities for prioritization and release planning. In *2011 Fifth International Workshop on Software Product Management (IWSPM)*, pages 47–56. IEEE, 2011.
- 18 Francesca Rossi, Kristen Brent Venable, and Toby Walsh. Preferences in constraint satisfaction and optimization. *AI Mag.*, 29(4):58–68, 2008. doi:10.1609/aimag.v29i4.2202.
- 19 Günther Ruhe and Moshood Omolade Saliu. The art and science of software release planning. *IEEE software*, 22(6):47–53, 2005.
- 20 Yevgeny Schreiber. Value-ordering heuristics: Search performance vs. solution diversity. In *International Conference on Principles and Practice of Constraint Programming*, pages 429–444. Springer, 2010.

Making Rigorous Linear Programming Practical for Program Analysis

Tengbin Wang ✉

College of Computer, National University of Defense Technology, Changsha, China

Liqian Chen¹ ✉

College of Computer, National University of Defense Technology, Changsha, China

Taoqing Chen ✉

State Key Laboratory of High Performance Computing, College of Computer,
National University of Defense Technology, Changsha, China

Guangsheng Fan ✉

State Key Laboratory of High Performance Computing, College of Computer,
National University of Defense Technology, Changsha, China

Ji Wang¹ ✉

State Key Laboratory of High Performance Computing, College of Computer,
National University of Defense Technology, Changsha, China

Abstract

Linear programming is a key technique for analysis and verification of numerical properties in programs, neural networks, etc. In particular, in program analysis based on abstract interpretation, many numerical abstract domains (such as Template Constraint Matrix, constraint-only polyhedra, etc.) are designed on top of linear programming. However, most state-of-the-art linear programming solvers use floating-point arithmetic in their implementations, leading to an approximate result that may be unsound. On the other hand, the solvers implemented using exact arithmetic are too costly. To this end, this paper focuses on advancing rigorous linear programming techniques based on floating-point arithmetic for building sound and efficient program analysis. Particularly, as a supplement to existing techniques, we present a novel rigorous linear programming technique based on Fourier-Mozkin elimination. On this basis, we implement a tool, namely, RlpSolver, combining our technique with existing techniques to lift effectiveness of rigorous linear programming in the scene of analysis and verification. Experimental results show that our technique is complementary to existing techniques, and their combination (RlpSolver) can achieve a better trade-off between cost and precision via heuristic rules.

2012 ACM Subject Classification Theory of computation → Linear programming; Software and its engineering → Formal methods

Keywords and phrases Linear programming, rigorous linear programming, abstract interpretation, program analysis, Fourier-Mozkin elimination

Digital Object Identifier 10.4230/LIPIcs.CP.2021.57

Funding This work is supported by the National Key R&D Program of China (No. 2017YFB1001802), and the National Natural Science Foundation of China (Nos. 61872445, 62032024).

1 Introduction

Linear programming [23] is increasingly widespread in the field of analysis and verification, such as program analysis, neural network verification, etc. Up to now, there have emerged a wide variety of high-efficiency and scalable linear programming solvers. One common trait of

¹ Corresponding authors



© Tengbin Wang, Liqian Chen, Taoqing Chen, Guangsheng Fan, and Ji Wang;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 57; pp. 57:1–57:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

these state-of-the-art solvers is that they are based on floating-point arithmetic and can only produce an approximate solution which may be far away from the optimal solution or even outside the feasible domain. On the other hand, the solvers based on exact arithmetic can get exact optimal solutions, but are time-consuming and have poor scalability in practice.

In this paper, we particularly consider the scene of applying linear programming in program analysis based on numerical abstract interpretation [6], in the context of which it is important to guarantee the soundness of the analysis. Linear programming is one of the basic operations of many numerical abstract domains, such as Template Constraint Matrix [21], constraint-only polyhedra [7], etc. To achieve high efficiency while guaranteeing soundness, this paper considers the scene of using low-cost rigorous linear programming techniques based on floating-point arithmetic to implement numerical abstract domains.

Rigorous linear programming has received much attention in the last two decades in the field of mathematics. In 2003, Neumaier and Shcherbina [19] propose a method that computes safe bounds of the objective function of the primal problem by solving the dual problem using floating-point linear programming. Jansson [11] proposes another technique which focuses on linear programming problem with uncertain input data and infinite bounds of decision variables. Although the above techniques produce general solution for rigorous linear programming problems, they may provide too conservative results for some linear programming problems in practice, especially when variables involve infinite bounds or problems are ill-conditioned. Hence, they may cause much precision loss when being used to implement program analysis.

Our approach. To make rigorous linear programming more practical for program analysis, we propose a novel rigorous linear programming technique based on Fourier-Mozkin elimination and interval arithmetic. This new technique is complementary to existing rigorous linear programming techniques and their combination via some heuristic rules can achieve a better trade-off between cost and precision than the identical ones.

The main contributions of this paper are as follows:

- We introduce a new rigorous linear programming technique based on Fourier-Mozkin elimination and interval arithmetic.
- We conduct an experimental evaluation of the usefulness and effectiveness of our technique for solving linear programming problems. The results show the availability of our technique.
- We develop a tool called RlpSolver that wraps existing rigorous linear programming techniques together, and provide heuristic rules to help choosing proper solvers in different cases.

For the sake of brevity, we use the following abbreviations throughout the paper:

- LP: linear programming.
- RLP: rigorous linear programming.
- FME: Fourier-Mozkin elimination.

The rest of the paper is organized as follows. In Section 2, we review some preliminaries of FME and RLP. Section 3 presents an overview of our work via a motivating example. In Section 4, we illustrate our RLP approach based on FME in detail. Section 5 presents the framework and heuristic rules of our tool named RlpSolver. Section 6 presents experimental results. Section 7 discusses some related work. Finally, conclusions and future work are given in Section 8.

2 Preliminaries

In this section, we first review some basic concepts and notations of linear system with interval coefficients. After that, we illustrate two existing representative RLP techniques, i.e., SafeBound [19] and ErrorBound [11].

2.1 Interval Linear System

Let $\underline{A}, \overline{A} \in \mathbb{R}^{m \times n}$ be real matrices with $\underline{A} \leq \overline{A}$. We define the following set of real matrices:

$$\mathbf{A} = [\underline{A}, \overline{A}] = \{A \in \mathbb{R}^{m \times n} : \underline{A} \leq A \leq \overline{A}\}$$

We call \mathbf{A} an interval matrix, whose lower and upper bounds are \underline{A} and \overline{A} respectively. An interval vector is a special interval matrix with one column, i.e., $\mathbf{d} = \{d \in \mathbb{R}^m : \underline{d} \leq d \leq \overline{d}\}$. Assume that $\mathbf{A} \in \mathbb{IR}^{m \times n}$ (where \mathbb{IR} denotes the set of intervals whose bounds are real numbers) is a interval matrix whose dimension is $m \times n$ and $b \in \mathbb{R}^m$ is a m-dimension real vector. We define

$$\mathbf{A}x \leq b \tag{1}$$

as an interval linear inequality system, representing a set of linear inequality systems constituted by all $Ax \leq b$ where $A \in \mathbf{A}$.

2.2 Linearization Technique

Now, we describe a method (called Orthant Reduction in this paper) to transform interval linear inequalities into linear inequalities [4]. Considering any interval linear inequality $\sum_i [a_i, b_i] x_i \leq c$ ($0 \leq i \leq n$), where the variable x_k ($0 \leq k \leq n$) is always non-negative or always non-positive. When $x_k \geq 0$, it always holds that $a_k x_k \leq b_k x_k$. Hence, in this case, $\sum_i [a_i, b_i] x_i \leq c$ is equivalent to $\sum_{i \neq k} [a_i, b_i] x_i + a_k x_k \leq c$. Similarly, when $x_k \leq 0$, it always holds that $a_k x_k \geq b_k x_k$. Hence, in this case, $\sum_i [a_i, b_i] x_i \leq c$ is equivalent to $\sum_{i \neq k} [a_i, b_i] x_i + b_k x_k \leq c$.

2.3 SafeBound

Neumaier and Shcherbina [19] propose a method (called SafeBound in this paper) that derives the safe bounds of the objective function by post-processing on the approximate result produced by a general floating-point LP solver.

Consider a LP model represented in the following form:

$$\begin{aligned} & \min c^T x \\ & s.t. Ax \leq b \end{aligned} \tag{2}$$

the dual of which is

$$\begin{aligned} & \max b^T y \\ & s.t. \begin{cases} A^T y = c \\ y \leq 0 \end{cases} \end{aligned} \tag{3}$$

Suppose y is an approximate result of (3). By introducing interval arithmetic, we have $r := A^T y - c \in \mathbf{r} = [\underline{r}, \bar{r}]$. Remind that $Ax \leq b$ and $y \leq 0$, and thus we have $c^T x = (A^T y - r)^T x = y^T Ax - r^T x \geq y^T b - r^T x \in y^T b - \mathbf{r}^T \mathbf{x}$. Hence, $\mu := \inf(y^T b - \mathbf{r}^T \mathbf{x})$ is the safe lower bound for $c^T x$, which can be calculated via floating-point arithmetic as follows:

rounddown:

$$\underline{r} = A^T y - c;$$

$$t = y^T b;$$

roundup:

$$\bar{r} = A^T y - c;$$

$$\mu = \max\{\underline{r}^T \underline{x}, \underline{r}^T \bar{x}, \bar{r}^T \underline{x}, \bar{r}^T \bar{x}\} - t;$$

$$\mu = -\mu;$$

where **rounddown** (**roundup**) denotes that we set rounding mode to $-\infty$ ($+\infty$).

2.4 ErrorBound

Jansson [11] proposes a method (called ErrorBound in this paper) to derive rigorous error bounds for the optimal value from boxes that are verified to contain feasible points. And Keil implements this method in Lurupa [14]. The method is described as follows (we refer the proofs to [11]).

Consider the following LP model:

$$\begin{aligned} f &:= \min c^T x \\ \text{s.t. } &\begin{cases} Ax \leq a \\ \underline{x} \leq x \leq \bar{x} \end{cases} \end{aligned} \quad (4)$$

where the simple bounds of x , i.e., \underline{x} and \bar{x} , may be infinite (i.e., $\underline{x} = -\infty$ or $\bar{x} = +\infty$), which can lead to uncertainties.

LP model (4) can be formally represented by the parameter tuple $P := (A, a, c)$. To cope with uncertainties of the input data, we introduce interval arithmetic by rewriting P with corresponding interval parameter tuple $\mathbf{P} := (\mathbf{A}, \mathbf{a}, \mathbf{c})$. Then we focus on this resulting interval LP problem \mathbf{P} .

The dual of interval LP problem \mathbf{P} is depicted as follows:

$$\begin{aligned} f &:= \max \mathbf{a}^T \mathbf{y} + \underline{\mathbf{x}}^T \mathbf{u} + \bar{\mathbf{x}}^T \mathbf{v} \\ \text{s.t. } &\begin{cases} \mathbf{A}^T \mathbf{y} + \mathbf{u} + \mathbf{v} = \mathbf{c} \\ \mathbf{y} \leq 0, \mathbf{u} \geq 0, \mathbf{v} \leq 0 \end{cases} \end{aligned} \quad (5)$$

► **Theorem 1 (Lower Bound).** *Consider an interval linear program \mathbf{P} with simple bounds $\underline{x} \leq \bar{x}$. Suppose interval vectors $\mathbf{y} \leq 0$ satisfy*

1. *for all free x_j and all $A \in \mathbf{A}$, there exists $y \in \mathbf{y}$ such that*

$$c_j - (A_{:j})^T y = 0$$

holds, and

2. *for all variables x_j bounded on one side only the defects*

$$\mathbf{d}_j := c_j - (A_{:j})^T \mathbf{y}$$

are non-negative if the variable is bounded from below and non-positive if it is bounded from above.

Then a rigorous lower bound for the optimal value can be computed as

$$\inf_{P \in \mathcal{P}} f(P) \geq f := \inf \left(\mathbf{a}^T \mathbf{y} + \sum_{\underline{x}_j \neq -\infty} \underline{x}_j \mathbf{d}_j^+ + \sum_{\bar{x}_j \neq +\infty} \bar{x}_j \mathbf{d}_j^- \right) \quad (6)$$

► **Theorem 2 (Upper Bound).** Consider an interval linear program \mathbf{P} with simple bounds $\underline{x} \leq \bar{x}$. Suppose interval vector \mathbf{x} satisfies

$$\mathbf{A}\mathbf{x} \leq \mathbf{a}, \underline{x} \leq \mathbf{x} \leq \bar{x} \quad (7)$$

Then a rigorous upper bound for the optimal value can be computed as

$$\sup_{P \in \mathcal{P}} f(P) \leq \bar{f} := \max\{\mathbf{c}^T \mathbf{x}\} \quad (8)$$

3 Overview

In this section, we provide a simple but typical example to illustrate our motivation. Consider the following LP problem:

$$\begin{aligned} \min z &= -9x_0 + 6x_1 - 4x_2 \\ \text{s.t. } \begin{cases} 3x_0 - 8x_1 + 5x_2 & \leq -14 \\ -5x_0 - 2x_1 + 6x_2 & \leq 17 \\ 5x_0 + 2x_1 - 6x_2 & \leq -17 \\ -2x_0 + 4x_1 & \leq 19 \\ x_0, x_1, x_2 & \geq 0 \end{cases} \end{aligned} \quad (9)$$

Table 1 shows the results and execution time of solving the above LP problem via different LP and RLP techniques. The first row of Table 1 shows the exact result given by *glp_exact* which is based on exact arithmetic from GLPK [16], a linear programming kit maintained by GNU. The existing RLP technique proposed by Neumaier and Shcherbina [19] (called SafeBound in Sect. 2.4) only produces minus infinity which is sound but too conservative. The reason lies in that the bounds of most variables in problem (9) involve infinity. Lurupa [14] which implements Jansson's method [11] (we call ErrorBound in Sect. 2.4), produces a finite lower bound. Compared with Lurupa and SafeBound, our FME-based RLP produces a more precise result which is also a rigorous finite lower bound. Besides, from the third column of Table 1, we can see that among these techniques, our FME-based RLP has the best performance.

■ **Table 1** Results of motivating example.

approaches	results	time(s)
<i>glp_exact</i>	6.04166666666666785090	0.000292
Lurupa [14]	6.04166582676214503067	0.000063
SafeBound [19]	$-\infty$	0.000160
FME-based RLP	6.04166666666666607454	0.000014

4 RLP based on Fourier-Mozkin Elimination

In this section, we will present our RLP approach based on Fourier-Mozkin elimination (FME). First, we review how to solve LP problem using FME. Then, we introduce how to derive a sound floating-point FME to construct a RLP approach. After that, we introduce techniques to improve efficiency of FME, so as to improve efficiency of RLP.

4.1 LP via Fourier-Mozkin Elimination

Fourier-Mozkin elimination is a general technique to perform variable elimination from a system of linear inequalities. Solving LP problems mathematically via FME has been discussed by Williams [26]. In this subsection, we will introduce the principle of applying FME to solve the following LP problem:

$$\begin{aligned} & \max c^T x \\ & s.t. \begin{cases} Ax \leq b \\ x \geq 0 \end{cases} \end{aligned} \quad (10)$$

To solve LP problem (10) by FME, we need to reconstruct the objective function by generating a new linear inequality, namely, $x_{n+1} - c^T x \leq 0$ and then we get a new linear inequality system:

$$\begin{cases} Ax \leq b \\ x_{n+1} \leq c^T x \\ x \geq 0 \end{cases} \quad (11)$$

For (11), we can get the upper bound of newly added variable, i.e., x_{n+1} , by applying FME to eliminate all the other variables in (11). Obviously, the upper bound of x_{n+1} is the maximum value of $c^T x$. Computing the minimum value of $c^T x$ can be reformulated as the following problem:

$$\min c^T x = -(\max -c^T x) \quad (12)$$

Following the above process, if applying FME based on exact arithmetic, we can get the exact maximum (minimum) value of the LP problem. However, if we conduct FME using floating-point arithmetic, the result may be unsound. In other words, we may get a smaller (larger) value than the exact maximum (minimum) value.

To this end, in this paper, we propose a RLP approach based on FME using floating-point arithmetic. The key idea is to use interval arithmetic and linearization techniques to derive a sound floating-point FME. In the following subsections, we first describe how to get a sound floating-point FME and then describe techniques to improve the precision and efficiency.

4.2 Sound Floating-Point FME

The key idea of constructing sound floating-point FME is to use interval arithmetic with outward rounding, that is, rounding up for computing upper bound and rounding down for computing lower bound. With interval arithmetic, we will get a new form of linear inequality in which all coefficients of variables are intervals.

For the sake of presentation, we introduce the following notations to denote floating-point operations:

- \oplus_r : floating-point addition.
- \ominus_r : floating-point minus.
- \otimes_r : floating-point multiplication.
- \oslash_r : floating-point division.

where $r \in \{+\infty, -\infty\}$ represents rounding mode in which $+\infty$ means upward and $-\infty$ means downward.

Assume we want to eliminate variable x_i ($i \leq n$) from the following two inequalities:

$$\begin{cases} a_i^+ x_i + \sum_{k \neq i, k \leq n} a_k^+ x_k + [\underline{a_{n+1}^+}, \overline{a_{n+1}^+}] x_{n+1} \leq c^+ & \text{if } a_i^+ > 0 \end{cases} \quad (13)$$

$$\begin{cases} a_i^- x_i + \sum_{k \neq i, k \leq n} a_k^- x_k + [\underline{a_{n+1}^-}, \overline{a_{n+1}^-}] x_{n+1} \leq c^- & \text{if } a_i^- < 0 \end{cases} \quad (14)$$

where only the coefficient for variable x_{n+1} (that is introduced in (11) to denote the objective value of the original LP problem) is an interval and all coefficients for other variables x_k ($k \leq n$) are scalars. After dividing (13) and (14) respectively by the absolute value of the coefficient of x_i using interval arithmetic with outward rounding, we have

$$\begin{aligned} x_i + \sum_{k \neq i, k \leq n} [a_k^+ \oslash_{-\infty} a_i^+, a_k^+ \oslash_{+\infty} a_i^+] x_k \\ + [\underline{a_{n+1}^+} \oslash_{-\infty} a_i^+, \overline{a_{n+1}^+} \oslash_{+\infty} a_i^+] x_{n+1} \leq c^+ \oslash_{+\infty} a_i^+ \end{aligned} \quad (15)$$

where $a_i^+ > 0$ and

$$\begin{aligned} -x_i + \sum_{k \neq i, k \leq n} [a_k^- \oslash_{-\infty} (\ominus a_i^-), a_k^- \oslash_{+\infty} (\ominus a_i^-)] x_k \\ + [\underline{a_{n+1}^-} \oslash_{-\infty} (\ominus a_i^-), \overline{a_{n+1}^-} \oslash_{+\infty} (\ominus a_i^-)] x_{n+1} \leq c^- \oslash_{+\infty} (\ominus a_i^-) \end{aligned} \quad (16)$$

where $a_i^- < 0$. By adding (15) and (16), we have

$$\begin{aligned} \sum_{k \neq i, k \leq n} [(a_k^+ \oslash_{-\infty} a_i^+) \oplus_{-\infty} (a_k^- \oslash_{-\infty} (\ominus a_i^-)), (a_k^+ \oslash_{+\infty} a_i^+) \oplus_{+\infty} (a_k^- \oslash_{+\infty} (\ominus a_i^-))] x_k + \\ [(\underline{a_{n+1}^+} \oslash_{-\infty} a_i^+) \oplus_{-\infty} (\underline{a_{n+1}^-} \oslash_{-\infty} (\ominus a_i^-)), (\overline{a_{n+1}^+} \oslash_{+\infty} a_i^+) \oplus_{+\infty} (\overline{a_{n+1}^-} \oslash_{+\infty} (\ominus a_i^-))] x_{n+1} \\ \leq (c^+ \oslash_{+\infty} a_i^+) \oplus_{+\infty} (c^- \oslash_{+\infty} (\ominus a_i^-)) \end{aligned} \quad (17)$$

Then according to the LP model (10), we have $x_k \geq 0$ when $k \leq n$. Hence, via Orthant Reduction technique described in Sect. 2.2, (17) can be linearized into the following form (where all coefficients for x_k ($k \leq n$) are scalars):

$$\sum_{k \neq i, k \leq n} d_k x_k + [\underline{d_{n+1}}, \overline{d_{n+1}}] x_{n+1} \leq c' \quad (18)$$

where

$$\begin{aligned} d_k &= ((a_k^+ \oslash_{-\infty} a_i^+) \oplus_{-\infty} (a_k^- \oslash_{-\infty} (\ominus a_i^-))) \quad \text{when } k \leq n \\ \underline{d_{n+1}} &= (\underline{a_{n+1}^+} \oslash_{-\infty} a_i^+) \oplus_{-\infty} (\underline{a_{n+1}^-} \oslash_{-\infty} (\ominus a_i^-)) \\ \overline{d_{n+1}} &= (\overline{a_{n+1}^+} \oslash_{+\infty} a_i^+) \oplus_{+\infty} (\overline{a_{n+1}^-} \oslash_{+\infty} (\ominus a_i^-)) \end{aligned}$$

and

$$c' = (c^+ \oslash_{+\infty} a_i^+) \oplus_{+\infty} (c^- \oslash_{+\infty} (\ominus a_i^-))$$

We use the above process to eliminate a variable x_i ($i \leq n$) from a system of inequalities where only the coefficient for variable x_{n+1} is an interval and the coefficients for other variables x_k ($k \leq n$) are scalars, which results in a system of the same form. Obviously, from the implementation point of view, we can skip the calculation of upper bound of interval coefficient of x_k (when $k \leq n$) in (15) (17), which can reduce some unnecessary calculations.

The division operation during converting (13) ~ (14) into (15) ~ (16) may introduce many interval coefficients for other variables, and converting interval coefficients into scalar ones using linearization may introduce precision loss. Considering that integer values of normal (not too large) magnitude can be represented exactly in floating-point representation, we also consider implementing FME using multiplication described as follows.

Assume $a_i^+ \otimes_{-\infty} a_i^- = a_i^+ \otimes_{+\infty} a_i^-$ can be represented exactly by floating-point representation (e.g., an integer). Then, after multiplying (13) by the minus of coefficient of x_i in (14) and multiplying (14) by the coefficient of x_i in (13) using interval arithmetic with outward rounding, we have

$$\begin{aligned} -a_i^+ a_i^- x_i + \sum_{k \neq i, k \leq n} [a_k^+ \otimes_{-\infty} (\ominus a_i^-), a_k^+ \otimes_{+\infty} (\ominus a_i^-)] x_k \\ + [\underline{a_{n+1}^+} \otimes_{-\infty} (\ominus a_i^-), \overline{a_{n+1}^+} \otimes_{+\infty} (\ominus a_i^-)] x_{n+1} \leq c^+ \otimes_{+\infty} (\ominus a_i^-) \end{aligned} \quad (19)$$

$$\begin{aligned} a_i^+ a_i^- x_i + \sum_{k \neq i, k \leq n} [a_k^- \otimes_{-\infty} a_i^+, a_k^- \otimes_{+\infty} a_i^+] x_k \\ + [\underline{a_{n+1}^-} \otimes_{-\infty} a_i^+, \overline{a_{n+1}^-} \otimes_{+\infty} a_i^+] x_{n+1} \leq c^- \otimes_{+\infty} a_i^+ \end{aligned} \quad (20)$$

where $a_i^+ > 0$ and $a_i^- < 0$. By adding (19) and (20), we have

$$\begin{aligned} \sum_{k \neq i, k \leq n} [(a_k^+ \otimes_{-\infty} (\ominus a_i^-)) \oplus_{-\infty} (a_k^- \otimes_{-\infty} a_i^+), (a_k^+ \otimes_{+\infty} (\ominus a_i^-)) \oplus_{+\infty} (a_k^- \otimes_{+\infty} a_i^+)] x_k + \\ [(\underline{a_{n+1}^+} \otimes_{-\infty} (\ominus a_i^-)) \oplus_{-\infty} (\underline{a_{n+1}^-} \otimes_{-\infty} a_i^+), (\overline{a_{n+1}^+} \otimes_{+\infty} (\ominus a_i^-)) \oplus_{+\infty} (\overline{a_{n+1}^-} \otimes_{+\infty} a_i^+)] x_{n+1} \\ \leq (c^+ \otimes_{+\infty} (\ominus a_i^-)) \oplus_{+\infty} (c^- \otimes_{+\infty} a_i^+) \end{aligned} \quad (21)$$

Then we can linearize (21) into scalar form similarly as linearizing (17). Similarly, we can skip the calculation of upper bound of interval coefficient of x_k in (21).

Finally, after we eliminate all variables x (consisting of x_k 's ($k \leq n$) from the system (11), we will result in an one-variable interval linear system over x_{n+1} :

$$\begin{cases} [\underline{a_1}, \overline{a_1}] x_{n+1} \leq b_1 \\ \dots \\ [\underline{a_m}, \overline{a_m}] x_{n+1} \leq b_m \end{cases}$$

which is equivalent to the disjunction of the following two linear systems:

$$\begin{cases} -x_{n+1} \leq 0 \\ \underline{a_1} x_{n+1} \leq b_1 \\ \dots \\ \underline{a_m} x_{n+1} \leq b_m \end{cases} \quad \vee \quad \begin{cases} x_{n+1} \leq 0 \\ \overline{a_1} x_{n+1} \leq b_1 \\ \dots \\ \overline{a_m} x_{n+1} \leq b_m \end{cases}$$

from which we can easily drive the upper bound of variable x_{n+1} .

To sum up, overall, we can derive FME-based RLP by substituting the process of FME described in Sect. 4.1 with sound floating-point FME described in this subsection. As floating-point FME is sound, so the result of FME-based RLP is also rigorous.

4.3 Redundancy Removal

It is known that FME for eliminating multiple variables may introduce a large number of redundant constraints during the process, which may lead to extra space and time cost. Hence, redundancy removal is a significant point to make FME practical.

In this paper, we utilize bit vector to implement two techniques, that is, Chernikov's rule [5] and Kohler's rule [15], to remove redundant constraints. The main ideas of these two techniques are described in the following subsections.

To simplify the descriptions of redundancy removal techniques, we rewrite linear inequality system (11) as

$$A'x' \leq b' \quad (22)$$

where A' , x' and b' respectively are

$$A' = \begin{pmatrix} A & 0 \\ -I & 0 \\ -c^T & 1 \end{pmatrix}, \quad x' = \begin{pmatrix} x \\ x_{n+1} \end{pmatrix}, \quad b' = \begin{pmatrix} b \\ 0 \\ 0 \end{pmatrix} \quad (23)$$

Note that sometimes we also represent linear system (22) via the parameter tuple (A', b') .

4.3.1 Bit Vector

We associate each inequality respectively with an index set which consists of integers representing the row index of the inequality in the original system (22). For efficiency, we encode the index set via a bit vector. The main idea is that if one inequality generated during FME is a combination result of some original inequalities, then in the bit vector of the generated inequality, the bits corresponding to the combined original inequalities will be set to 1, while the remaining bits are set to 0. E.g., if an inequality φ is a combination result of the first and third inequalities of the original inequality system, then the bit vector of φ is $0 \cdots 00101$ (the lowest bit from right corresponds to the first inequality in the system).

4.3.2 Chernikov's Rule

In [5], Chernikov proposes a heuristic rule to avoid generating some redundant constraints during FME by restricting the length of the index set associated with each inequality. To simplify description, we write q_i to denote the index set of inequality i . Before the starting of elimination, the index sets of each constraint q_i in the original inequality system (22) are initialized as singleton sets which consist of the row index i of the corresponding constraint q_i in the constraint matrix A' (i.e., q_i is one of $\{1, 2, \dots, m\}$, where m is the number of rows in A'). Assume that we want to make a combination between inequalities i and j . Before doing combination, we can calculate the index set of this combination, that is, $q_{ij} = q_i \cup q_j$. If the size of q_{ij} is strictly greater than $s + 1$, where s means the combination is to be conducted during the process of eliminating the s -th variable (after eliminating $s - 1$ variables), then we can skip this combination since the resulting constraint is definitely redundant [5].

Algorithm 1 depicts the procedure of FME integrated with Chernikov's rule.

■ **Algorithm 1** Procedure of FME with Chernikov's rule.

```

1: /*  $(A', b')$  denotes the linear inequality system (22) */
2: /*  $bv'$  denotes the set of bit vectors corresponding to the inequalities in  $(A', b')$  */
3: /*  $j$  means the index of the variable to be eliminated, also corresponding to the  $j$ -th
   column of  $A'$  */
4: procedure FME_Iteration( $A', b', bv', j$ )
5:   //  $i$  means the  $i$ -th row of  $A'$  ( $i = 1, 2, \dots, m$ )
6:    $I_j^+ \leftarrow \{i : a_{ij} > 0\}$ ;    $I_j^- \leftarrow \{i : a_{ij} < 0\}$ ;    $I_j^0 \leftarrow \{i : a_{ij} = 0\}$ 
7:   // Extracting inequalities and their corresponding bit vectors of  $A'$  with indices in  $I_j^0$ 
8:    $(A'', b'', bv'') \leftarrow (A', b', bv') \mid_{I_j^0}$ 
9:   for  $k^+ \in I_j^+$  do
10:    for  $k^- \in I_j^-$  do
11:      // bv_1bits() derives the number of one's in a bit vector
12:       $ij\_1bits \leftarrow bv\_1bits(bv_{k^+} \mid bv_{k^-})$  // Here  $\mid$  denotes bitwise OR operation
13:      // Check Chernikov's rule
14:      if  $ij\_1bits > j + 1$  then
15:        continue
16:      end if
17:      // FME_combine() combines of two inequalities, as described in Sect. 4.2
18:      // FME_add() adds one inequality together with its corresponding
19:      // bit vector into the resulting inequality system
20:       $(a_c, b_c) \leftarrow FME\_combine((A'_{k^+}, b'_{k^+}), (A'_{k^-}, b'_{k^-}))$ 
21:       $(A'', b'', bv'') \leftarrow FME\_add((A'', b'', bv''), (a_c, b_c, bv_{k^+} \mid bv_{k^-}))$ 
22:    end for
23:  end for
24:  return  $(A'', b'', bv'')$ 
25: end procedure

```

■ **Algorithm 2** Procedure of redundancy removal with Kohler's Rule.

```

1: procedure RMR_ByKohler( $A', b', bv'$ )
2:    $(A'', b'') \leftarrow \emptyset$ ;    $bv'' \leftarrow \emptyset$ 
3:   // nb_rows() derives the number of rows for the parameter matrix
4:   for  $i \leftarrow 0$  to  $nb\_rows(A') - 1$  do
5:      $flag = false$ 
6:     for  $j \leftarrow i + 1$  to  $nb\_rows(A')$  do
7:       if  $(bv_i \mid bv_j) == bv_i$  then
8:          $flag = true$ 
9:         break
10:      end if
11:    end for
12:    if  $flag == false$  then
13:       $(A'', b'', bv'') \leftarrow FME\_add((A'', b'', bv''), (A'_i, b'_i, bv'_i))$ 
14:    end if
15:  end for
16:  return  $(A'', b'', bv'')$ 
17: end procedure

```

4.3.3 Kohler's Rule

Kohler's rule is another well-known technique for removing redundant constraints during FME. In our design, after doing FME in one iteration step (i.e., eliminating one variable), we can derive a new inequality system, (A', b') , together with its bit vectors, bv' . Then we check each inequality in (A', b') with others to find out whether the subset relation exists between them, to determine the redundant inequality, more clearly, superset being redundant and subset not. Algorithm 2 depicts the procedure of Kohler's rule to remove redundant constraints.

Finally, we integrate Chernikov's rule and Kohler's rule into the process of FME. Algorithm 3 depicts the procedure of our FME-based RLP, with initial inequality system as input and maximum value of objective function as output.

■ **Algorithm 3** Procedure of FME-based RLP.

Input: Initial linear inequality system (A', b')

Output: The maximum value of x_{n+1}

```

1: // bv_initial() initializes bit vectors for initial system, as described in Sect. 4.3.1
2:  $bv' \leftarrow bv\_initial(A', b')$ 
3: for  $j \leftarrow 1$  to  $nb\_columns(A')$  do
4:    $(A', b', bv') \leftarrow FME\_Iteration(A', b', bv', j)$ 
5:    $(A', b', bv') \leftarrow RMR\_ByKohler(A', b', bv')$ 
6: end for
7: // max() derives the maximum value for objective function (noting
8: //   that at this location,  $(A', b')$  only involves one variable, i.e.,  $x_{n+1}$ )
9: return  $max(A', b')$ 

```

4.4 Optimization Considering Sparsity

In the field of program analysis, the objective function and constraint system are usually sparse [24, 25], i.e., they contain mostly zeros. Hence, when we solve RLP problems encountered during program analysis, we may make use of the sparsity in the LP problem to accelerate the solving process.

Consider the LP problem encoded in (22), we say two variables x_i and x_j are *relevant*, if there exists a constraint ϕ in $A'x' \leq b'$ of (22) such that the coefficients of x_i and x_j in ϕ are not zero. The defined *relevant* relation is an equivalence relation on the set of variables in x' , and the collection of its equivalence classes forms a partition of set of variables in x' . Let S denote the equivalence class that variable x_{n+1} belongs in, and let \bar{S} denote the set of variables not in S . Then the LP problem encoded in (22) can be reformulated as

$$A'x' \leq b'$$

where

$$A' = \begin{pmatrix} A_S & 0 \\ 0 & A_{\bar{S}} \end{pmatrix}, \quad x' = \begin{pmatrix} x_S \\ x_{\bar{S}} \end{pmatrix}, \quad b' = \begin{pmatrix} b_S \\ b_{\bar{S}} \end{pmatrix} \quad (24)$$

Thus, the LP problem encoded via

$$A'x' \leq b'$$

can be reduced to the following equivalent LP problem:

$$A_S x_S \leq b_S \quad (25)$$

Note that to derive the variable bound for x_{n+1} , (25) is equivalent to (22). It is worth mentioning that solving (25) will be more efficient than solving (22), since solving (25) involves less variables to be eliminated from a linear system with less constraints.

5 Integrating RLP Techniques

Our proposed FME-based RLP, and two existing RLP techniques (i.e., SafeBound and ErrorBound) have their own advantages and disadvantages. E.g., FME-based RLP has high precision and specializes in small-scale LP problems but degrades a lot in efficiency when the scale of LP problems increases. SafeBound and ErrorBound can handle large-scale LP problems but have poor accuracy. To make RLP more practical and effective, we implement a tool called RlpSolver, combining our FME-based RLP together with two existing RLP techniques, that is, SafeBound [19], ErrorBound [11] (implemented in Lurupa [14]). Fig. 1 provides an overview of RlpSolver.

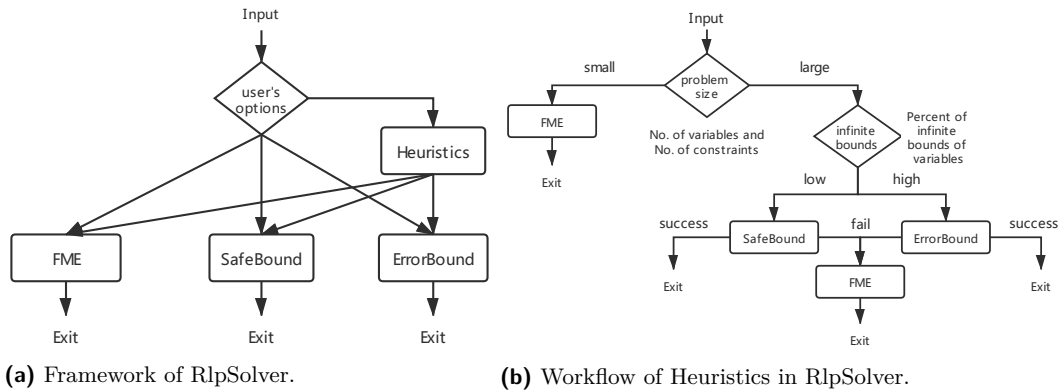


Figure 1 Overview of RlpSolver.

As shown in Fig. 1a, in RlpSolver, we can choose any of the three techniques to solve one LP problem. Moreover, RlpSolver provides an option to automatically choose a proper technique to solve a LP problem via heuristic rules. The workflow of the heuristics we design is depicted in Fig. 1b, which automatically chooses a proper RLP technique for a given LP problem. The details of heuristic rules are as follows:

- FME-based RLP is often more precise than SafeBound and ErrorBound in many cases, but may cost more time when the number of constraints is greater than some threshold. Hence, by default, RlpSolver will choose FME-based RLP when the number of constraints is less than some threshold. In other cases, we will try SafeBound or ErrorBound.
- When the number of constraints is greater than some threshold, we will choose between ErrorBound and SafeBound. Specifically, when many variables' bounds are infinite (i.e., when the percent of infinite bounds exceeds some threshold), we will use ErrorBound since SafeBound may often give infinity as results and ErrorBound behaves better than SafeBound for infinite bounds.
- Since ErrorBound and SafeBound depend upon floating-point simplex or other LP algorithms, they may encounter numerical instability when solving LP problems and may not fit for dealing with ill-conditioned (dual) problems, while FME-based RLP is more robust than ErrorBound and SafeBound, thus we will try FME-based RLP if ErrorBound or SafeBound fails.

6 Implementation and Experiments

To evaluate the precision and efficiency of our proposed FME-based RLP and the integration tool RlpSolver, we conduct experiments over randomly generated LP problems. We apply `glp_exact`, `SafeBound`, `Lurupa` (which implements `ErrorBound`), and FME-based RLP to our benchmark respectively. We use the exact LP API from GLPK (a linear programming kit maintained by GNU) [16], i.e., `glp_exact` (which is implemented via exact arithmetic), as our baseline.

By setting thresholds for the number of variables and constraints (10 and 15 respectively), we split our benchmark into four categories, that is, small number of variables with small number of constraints (SV_SC), small number of variables with large number of constraints (SV_LC), large number of variables with small number of constraints (LV_SC), and large number of variables with large number of constraints (LV_LC). In our benchmark, the ranges of sizes of constraints (and variables) in SV_SC, SV_LC, LV_SC and LV_LC are 6~14 (4~9), 15~25 (4~9), 12~14 (10~12), 15~25 (10~18) respectively. Fig. 2 shows the log of execution time. From Fig. 2, we can see that when the number of constraints is small (as shown in Fig. 2 (a) and (c)), the performance of FME-based RLP is almost at the same level as `Lurupa` and `SafeBound`, even better in many cases. When the number of constraints is large (as shown in Fig. 2 (b) and (d)), FME-based RLP costs more time than `SafeBound` and `Lurupa`, but its performance is mostly better than that of `glp_exact`.

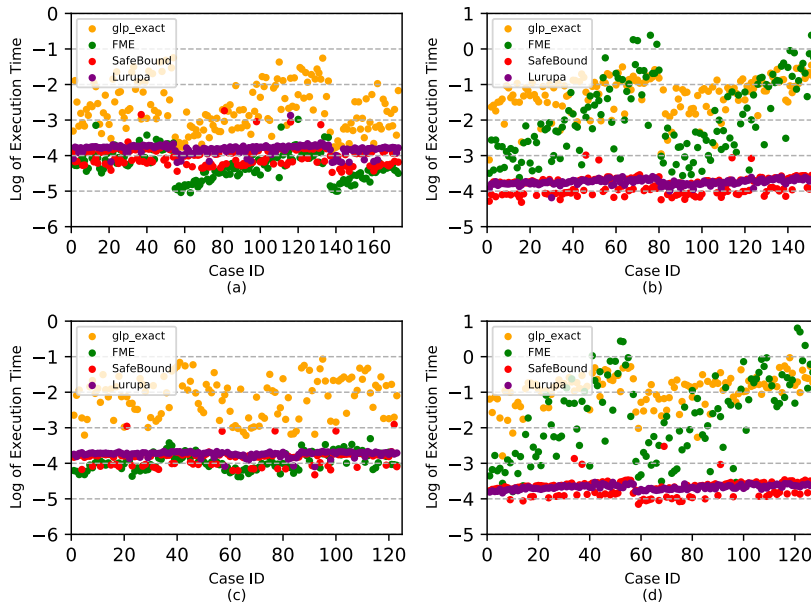


Figure 2 Execution time over benchmark SV_SC (a), SV_LC (b), LV_SC (c), and LV_LC (d).

For the precision, as shown in Table 2, we provide the statistics of the number of instances that FME-based RLP, `SafeBound` and `Lurupa` respectively provide the highest precision in benchmark. For example, in SV_LC category, there are 149 out of 152 cases where FME-based RLP obtains the highest precision. Experimental results show that FME-based RLP is mostly more precise than `SafeBound` and `Lurupa`. Thus, when the number of constraints is small, FME-based RLP is the best choice and also a good choice in the cases where the number of constraints is large but requiring high precision. Moreover, during experiments,

we find that there are many cases where SafeBound outputs too conservative results (i.e., infinity as the objective value) while FME-based RLP can output bounded (finite) results which are close to that of `glp_exact`. In other words, FME-based RLP can be used in these cases where SafeBound or Lurupa provide too conservative results, to improve the precision of program analysis. The last column of Table 2 shows that there are 272 out of 575 cases where integration tool RlpSolver can derive the highest precision.

■ **Table 2** The number of cases where each technique provides the highest precision.

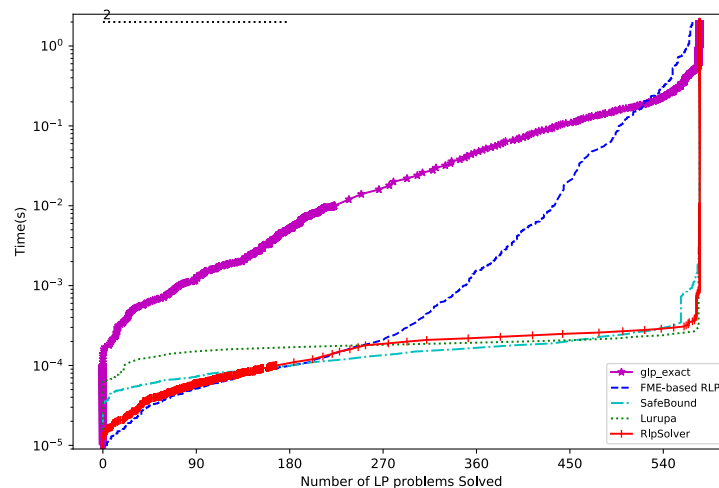
Categories	Total	FME-based RLP	SafeBound	Lurupa	Equal ¹⁾	RlpSolver
SV_SC	173	157 ²⁾	8	24	7	156
SV_LC	152	149	2	1	0	1
LV_SC	123	115 ³⁾	1	10	1	115
LV_LC	127	127	0	0	0	0

1) “Equal” means the results of FME-based RLP, SafeBound and Lurupa are equal.

2) There is 1 case where FME-based RLP is equal to only SafeBound and 1 case where FME-based RLP is equal to only Lurupa in this category.

3) There is 1 case where FME-based RLP is equal to only Lurupa in this category.

For the execution time, Fig. 3 shows the cumulative time of different techniques for solving LP problems in benchmark. The dashed line at the top left corner of Fig. 3 means the time-out period. The curves of FME-based RLP and RlpSolver show that FME-based RLP and RlpSolver take less execution time over the left half part than Lurupa and SafeBound. Over the right half part (which consists of problems with large number of constraints), the performance of FME-based RLP degrades a lot, while the execution times of RlpSolver, SafeBound and Lurupa are still in the same order of magnitude. On the whole, via heuristic rules, RlpSolver can achieve a good balance in terms of time cost and precision by choosing a proper technique for different cases.



■ **Figure 3** Execution time of several techniques.

7 Related Work

FME and FME-based LP. Fourier-Mozkin elimination is a general technique to perform variable elimination from a system of linear inequalities. Kohler [15] proposes a heuristic rule (called Kohler’s rule now) for removing redundant constraints. Chernikov [5] proposes additional rules to avoid generating some redundant combinations during elimination. Basttrakov et al. [1] propose a new way of checking Chernikov rules using bit pattern trees as an accelerating data structure to avoid extensive enumeration. Maréchal et al [17] present a raytracing algorithm that replaces most LP problem resolutions by distance computations to efficiently eliminate redundancies in polyhedra. Solving LP problems via FME has been continuously studied. Williams [26] first adapts FME to solve LP problems mathematically. Kannappan et al. [12] propose a modified FME method of solving LP problems which can reduce the number of additional constraints to a considerable extent.

RLP. Rigorous linear programming has received much attention in the recent two decades. In the 2003 seminal paper, Neumaier and Shcherbina [19] propose a technique that computes safe bounds of objective value of primal problem by solving the dual problem with floating-point linear programming. At almost the same time, Jansson [11] proposes another method for LP with uncertain input data and infinite bounds. Keil implements Jansson’s method [11] in a tool, named Lurupa [14]. Guilbeau et al. [10] review the technique proposed in [19] and point out typographical errors in original publication and give some advice for other implementers. Rump [20] gives some details on how to obtain mathematically rigorous results for global optimization implemented in floating-point arithmetic.

LP and RLP in Analysis and Verification. LP is widely-used in analysis and verification of programs, neural networks, etc. Sankaranarayanan et al. [22] use standard LP in their Template Constraint Matrix (TCM) domain to compute the right-hand constants for templates. Chen et al. [3] use RLP in their floating-point polyhedra domain [3] and interval polyhedra domain [4] to support domain operations. David Monniaux [18] proposes a simple but sound and complete preprocessing phase which can be adapted to existing SMT solvers via floating-point computations to help an exact linear arithmetic decision procedure. Besson [2] explains how to design a sound procedure for linear arithmetic built on an inexact floating-point LP solver. Dillig et al. [8] propose a sound and complete simplex-based algorithm for solving linear inequalities over integers which can be viewed as a semantic generalization of the branch-and-bound technique. It is also worth mentioning that LP and mixed integer LP (MILP) are used in recent neural network verification [9, 13].

8 Conclusion

Rigorous linear programming is an important technique to make implementations of program analysis and verification techniques sound and efficient in practice. Existing RLP techniques sometimes produce too conservative (even unbounded) results, especially when many bounds of variables in the problem are infinite or when the LP problem is ill-conditioned. To address this problem, we propose a new technique, FME-based RLP, which can be treated as a supplement to existing RLP techniques. On this basis, we implement a tool, RlpSolver, wrapping FME-based RLP and existing RLP techniques together, and propose heuristics to select a proper technique for different LP problems. Experimental results show that our FME-based RLP is complementary to existing RLP techniques and provides more precise

results than existing RLP techniques for many cases. Experimental results also show that our RLP integration tool, i.e., RlpSolver, achieves a good performance in terms of precision and efficiency by choosing a proper technique for different cases.

For future work, we plan to make use of more techniques to expedite removing redundancy during the process of FME. We also plan to conduct experiments in the context of using RLP in program analysis and verification.

References

- 1 SI Bastrakov, AV Churkin, and N Yu Zolotykh. Accelerating fourier–motzkin elimination using bit pattern trees. *Optimization Methods and Software*, pages 1–14, 2020.
- 2 Frédéric Besson. On using an inexact floating-point lp solver for deciding linear arithmetic in an smt solver. In *8th International Workshop on Satisfiability Modulo Theories*, 2010.
- 3 Liqian Chen, Antoine Miné, and Patrick Cousot. A sound floating-point polyhedra abstract domain. In *Asian Symposium on Programming Languages and Systems*, pages 3–18. Springer, 2008.
- 4 Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In *International Static Analysis Symposium*, pages 309–325. Springer, 2009.
- 5 Sergei Nikolaevich Chernikov. The convolution of finite systems of linear inequalities. *USSR Computational Mathematics and Mathematical Physics*, 5(1):1–24, 1965.
- 6 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.
- 7 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978.
- 8 Isil Dillig, Thomas Dillig, and Alex Aiken. Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In *International Conference on Computer Aided Verification*, pages 233–247. Springer, 2009.
- 9 Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 269–286. Springer, 2017.
- 10 Jared T Guilbeau, Md Istiaq Hossain, Sam D Karhbet, Ralph Baker Kearfott, Temitope S Sanusi, and Lihong Zhao. A review of computation of mathematically rigorous bounds on optima of linear programs. *Journal of Global Optimization*, 68(3):677–683, 2017.
- 11 Christian Jansson. Rigorous error bounds for the optimal value of linear programming problems. In *International Workshop on Global Optimization and Constraint Satisfaction*, pages 59–70. Springer, 2002.
- 12 P Kanniappan and K Thangavel. Modified fourier’s method of solving linear programming problems. *Opsearch*, 35(1):45–56, 1998.
- 13 G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, 2017.
- 14 Christian Keil. Lurupa-rigorous error bounds in linear programming. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.
- 15 David A Kohler. Projections of convex polyhedral sets. Technical report, California Univ Berkeley Operations Research Center, 1967.
- 16 Andrew Makhorin. Gnu linear programming kit. *Moscow Aviation Institute, Moscow, Russia*, 38, 2001.

- 17 Alexandre Maréchal and Michaël Périn. Efficient elimination of redundancies in polyhedra by raytracing. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 367–385. Springer, 2017.
- 18 David Monniaux. On using floating-point computations to help an exact linear arithmetic decision procedure. In *International Conference on Computer Aided Verification*, pages 570–583. Springer, 2009.
- 19 Arnold Neumaier and Oleg Shcherbina. Safe bounds in linear and mixed-integer linear programming. *Mathematical Programming*, 99(2):283–296, 2004.
- 20 Siegfried M Rump. Mathematically rigorous global optimization in floating-point arithmetic. *Optimization Methods and Software*, 33(4-6):771–798, 2018.
- 21 Sriram Sankaranarayanan, Thao Dang, and Franjo Ivančić. Symbolic model checking of hybrid systems using template polyhedra. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 188–202. Springer, 2008.
- 22 Sriram Sankaranarayanan, Henny B Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 25–41. Springer, 2005.
- 23 Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.
- 24 Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Making numerical program analysis fast. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 303–313. ACM, 2015.
- 25 Gagandeep Singh, Markus Püschel, and Martin T. Vechev. A practical construction for decomposing numerical abstract domains. *Proc. ACM Program. Lang.*, 2(POPL):55:1–55:28, 2018.
- 26 H Paul Williams. Fourier’s method of linear programming and its dual. *The American mathematical monthly*, 93(9):681–695, 1986.

Engineering an Efficient PB-XOR Solver

Jiong Yang

School of Computing, National University of Singapore, Singapore

Kuldeep S. Meel

School of Computing, National University of Singapore, Singapore

Abstract

Despite the NP-completeness of Boolean satisfiability, modern SAT solvers are routinely able to handle large practical instances, and consequently have found wide ranging applications. The primary workhorse behind the success of SAT solvers is the widely acclaimed Conflict Driven Clause Learning (CDCL) paradigm, which was originally proposed in the context of Boolean formulas in CNF. The wide ranging applications of SAT solvers have highlighted that for several domains, CNF is not a natural representation and the reliance of modern SAT solvers on resolution proof system limit their ability to efficiently solve several families of constraints. Consequently, the past decade has witnessed the design of solvers with native support for constraints such as Pseudo-Boolean (PB) and CNF-XOR.

The primary contribution of our work is an efficient solver engineered for PB-XOR formulas, i.e., formulas consisting of a conjunction of PB and XOR constraints. We first observe that a simple adaption of CNF-XOR architecture does not provide an improvement over baseline; our analysis highlights the need for careful engineering of the order of propagations. To this end, we propose three different tactics, all of which achieve significant performance improvements over the baseline. Our work is motivated by applications arising from binarized neural network verification where the verification of properties such as robustness, fairness, trojan attacks can be reduced to model counting queries; the state of the art model counters reduce counting to polynomially many SAT queries over the original formula conjuncted with randomly generated XOR constraints. To this end, we augment ApproxMC with LinPB and we call the resulting counter as ApproxMCPB. In an extensive empirical comparison over 1076 benchmarks, we observe that ApproxMCPB can solve 912 instances while the baseline version of ApproxMC4 (augmented with CryptoMiniSat) can solve only 802 instances.

2012 ACM Subject Classification Theory of computation; Computing methodologies → Artificial intelligence

Keywords and phrases PB-XOR Solving, Pseudo-Boolean, XOR, Gauss Jordan Elimination, SAT-Solving, Model Counting

Digital Object Identifier 10.4230/LIPIcs.CP.2021.58

Supplementary Material The open source tools and benchmarks are available at:

Software (Source Code LinPB): <https://github.com/meelgroup/linpb>

Software (Source Code ApproxMCPB): <https://github.com/meelgroup/approxmcpb>

Dataset (BNN benchmarks & raw data of experiments): <https://doi.org/10.5281/zenodo.5526835>

Funding This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004] and AI Singapore Programme [AISG-RP-2018-005], and NUS ODPRT Grant [R-252-000-685-13]. The computational work for this article was fully performed on resources of the National Supercomputing Centre, Singapore (<https://www.nscg.sg>).

Acknowledgements We are grateful to the anonymous reviewer for pointing out a subtle bug in the presentation of Algorithm 1. We are thankful to Priyanka Golia and Yang Suwei for their detailed feedback on the early drafts of the paper.



© Jiong Yang and Kuldeep S. Meel;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 58; pp. 58:1–58:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Given a Boolean formula F , the problem of satisfiability (SAT) is to determine whether there is an assignment σ to the set of variables such that F evaluates to True. The celebrated work of Cook and Levin (independently) established the NP-completeness of SAT [6, 21] and thereby establishing SAT at the core of the fundamental question of whether $P=NP$? From the practical perspective, the past three decades have been witness to unprecedented performance improvements in SAT solvers, which largely owes to the Conflict Driven Clause Learning (CDCL) paradigm, owing to seminal work of Marques-Silva and Sakallah [22], which has seen been combined with careful software engineering along with rigorous theoretical advances. Quoting Knuth: “The story of satisfiability is a tale of the triumph of software engineering blended with rich doses of beautiful mathematics.”

From a theoretical perspective, the breakthrough performance improvements of SAT solvers can be cast as surprising given the reliance of CDCL solvers on the resolution as a proof system. Resolution can be characterized as a weak proof system with strong lower bounds for simple formulas such as those based on Pigeon Hole Principle [15, 38]. The weakness of resolution as a proof system is well known to the SAT community, and consequently there have been efforts since the early 2000s in the design of solvers that can perform reasoning more powerful than resolution [10, 3, 33, 20, 12].

The CDCL solver’s reliance on resolution contributed to the rise of Conjunctive Normal Form (CNF) to be the input representation for modern SAT solvers. While Tseitin encoding provides an efficient method to convert an arbitrary Boolean formula into CNF with only a linear overhead [37], such an encoding deprives the solver of the natural representation of the problem. Several problems arising from practice can be naturally represented constraints using XORs and Pseudo Boolean (PB), which provided an impetus to the design of solvers with native support for such representations. It is worth remarking that for representations such as XORs and PBs, proof systems such as Gaussian Elimination and cutting planes [7] are known to be exponentially more powerful than resolution.

To summarize, the weakness of resolution and availability of instances arising from practice with natural representation in forms other than CNF have led to the design of solvers such as CryptoMiniSat [36] and RoundingSat [12] with native support for XORs and PB constraints respectively. While the design of CryptoMiniSat was originally motivated by applications in cryptanalysis, its availability served as a bedrock to the development of approximate model counting techniques over the past decade [14, 4, 13, 5, 25, 24, 1]. The current state-of-the-art approximate model counter is ApproxMC [4], which is in its fourth version [34]. ApproxMC takes in a CNF formula and then relies on hashing-based techniques to reduce counting to polynomially many SAT queries over the formulas represented as a conjunction of the original CNF formula and randomly generated XOR constraints. The past three years have witnessed the power of tight integration of CryptoMiniSat and ApproxMC [35, 34].

Akin to applications relying on SAT queries, for several applications of counting, CNF is not the natural representation. Of particular interest to us are applications arising from verification of neural network [27]. Baluta et al. proposed the framework of quantitative verification, called NPAQ, which reduces the verification of properties such as robustness, fairness, trojan attacks over Binarized Neural Networks (BNNs) to counting queries [2]. It is worth observing that the natural representation of BNNs is a conjunction of PB constraints and the counting framework of ApproxMC introduces randomly generated XOR constraints; therefore, each of the underlying SAT queries can be represented as a conjunction of PB and XOR constraints. The current implementation of ApproxMC is built on top

of CryptoMiniSat due to its native support of XORs and therefore, NPAQ employs CNF encoding of PB constraints into CNF. While NPAQ was shown to scale to large instances, the scalability remains a major challenge. In this context, one wonders whether it is possible to address the scalability challenge of hashing-based approximate model counting when the instances have their natural representation in PB via the design of an efficient model counter that has native support for both PB and XOR constraints.

Given the availability of the state-of-the-art cutting plane proof system-based PB solver, RoundingSat [12], a straightforward first step would be to integrate the easily portable Gauss-Jordan elimination module in CryptoMiniSat [36] into RoundingSat. Our initial foray, surprisingly, yielded little to no significant improvement in comparison to the current approach of invoking CryptoMiniSat over PB constraints encoded into CNF. We denote this approach by Lazy-GJE.

The primary contribution of this work is an efficient satisfiability solver, called LinPB, for PB-XOR formulas. Our design of LinPB is based on our identification of the key performance bottleneck in the aforementioned approach: the presence of *redundant* propagation. In LinPB, we propose three novel strategies for propagation: Shared-Watches, Eager-GJE, and Mixed-Watches. To evaluate the empirical effectiveness of our proposed techniques, we integrate LinPB with the ApproxMC algorithm; we call the resulting tool ApproxMCPB. We perform an empirical comparison of ApproxMCPB vis-a-vis ApproxMC4 tool and other state-of-the-art counters on over 1076 benchmarks arising from binarized neural network verification for diverse properties [2]. Our empirical comparison shows that while ApproxMC can solve only 802 instances, ApproxMCPB can solve 912 instances, thereby achieving a gain of over 100 instances. Furthermore, the PAR-2 score for ApproxMC is 3305 seconds while the PAR-2 score for ApproxMCPB is 1822 seconds, thereby achieving an almost 50% decrease in PAR-2 score. Among different strategies, we observe that usage of Lazy-GJE leads to ApproxMCPB solving 804 instances while usage of Shared-Watches, Eager-GJE, and Mixed-Watches leads to solving 892, 892, and 912 instances.

The rest of the paper is organized as follows: We discuss notations and preliminaries in Section 2 and introduce the background of PB and XOR solving in Section 3. In Section 4, We focus on core technical contributions for PB-XOR solving. We then present an extensive experimental evaluation in Section 5 and finally conclude in Section 6.

2 Notations and Preliminaries

Let $X = \{x_1, x_2, \dots, x_n\}$ be the set of Boolean variable. A literal is a variable or its negation. A clause is a disjunction of literals.

For a Boolean formula φ , we use $\text{Vars}(\varphi)$ to denote the set of variables involved in φ . If an assignment σ of truth values to all the variables in $\text{Vars}(\varphi)$ makes formula φ evaluate to True, it's called a *solution* or *witness* of φ . We use $\text{sol}(\varphi)$ to denote the set of all witnesses of φ . Given a set of variables $\mathcal{P} \subseteq \text{Vars}(\varphi)$, we denote the projection of R_F on \mathcal{P} by $\text{sol}(\varphi)_{\downarrow \mathcal{P}}$.

In the context of *propositional model counting*, we aim to compute the number of solutions, i.e. $|\text{sol}(\varphi)|$, for a given Boolean formula φ . A *probably approximately correct* (PAC) counter denotes a probabilistic algorithm $\text{ApproxCount}(\cdot, \cdot, \cdot)$ that takes as inputs a formula φ , a tolerance $\epsilon > 0$ and a confidence $1 - \delta \in (0, 1]$, and returns a count c with (ϵ, δ) -guarantees, i.e., $\Pr[|\text{sol}(\varphi)|/(1 + \epsilon) \leq c \leq (1 + \epsilon)|\text{sol}(\varphi)|] \geq 1 - \delta$. Similarly, projected model counting is to compute $|\text{sol}(\varphi)_{\downarrow \mathcal{P}}|$ instead of $|\text{sol}(\varphi)|$ for a given sampling set $\mathcal{P} \subseteq \text{Vars}(F)$.

A (linear) pseudo-Boolean (PB)-constraint is represented as $\sum_{i \in S} w_i x_i \geq k$ where, $S \subseteq [n]$, $w_i, k \in \mathbb{Z}$. An XOR-constraint is represented as $\oplus_{i \in S} x_i = b$ for $S \subseteq [n]$ and $b \in \{0, 1\}$ where \oplus represents XOR operator. A formula is in CNF if it can be represented as conjunction

of clauses. Similarly, a formula is PB form (resp. XOR form) if it can be represented as conjunction of PB (resp. XOR) constraints. Furthermore, a formula is in PB-XOR (resp. CNF-XOR) form if it can be represented as $\phi \wedge \psi$ where ϕ is a formula in PB (resp. CNF) form and ψ is a formula in XOR form.

PB Encoding of XOR

Our work focuses on the efficient handling of PB-XOR formulas. A simple baseline approach would be to express XOR constraints as PB constraints, and in this context, one wonders whether there is an efficient method to encode PB constraints. We now state a well-known encoding of XOR into PB constraints via the introduction of additional auxiliary variables.

► **Observation 1** (folklore). *Given a XOR constraint: $\bigoplus_{i=1}^{i=n} x_i = b$, and \oplus denotes exclusive disjunction operation, we introduce auxiliary Boolean variables $\{t_i\}, i = 1, 2, \dots, \lfloor \log_2(n) \rfloor$. Then, the XOR constraint is logically equivalent to the following pseudo-Boolean constraint:*

$$\sum_{i=1}^n x_i - \sum_{i=1}^{\lfloor \log_2(n) \rfloor} 2^i \cdot t_i = b \quad (1)$$

Applying the encoding in Definition 1, we achieve a one-to-one mapping between XOR constraint and its PB encoding.

3 Background

In order to put our contributions in context, we provide a brief discussion about the workings of the current state-of-the-art implementations of Gauss-Jordan elimination procedures in modern SAT solvers such as CryptoMiniSat [36].

3.1 Gauss-Jordan Elimination

Gauss-Jordan Elimination (GJE) is an efficient algorithm for solving systems of linear equations. Since XOR constraints are considered as linear equations modulo two, Gauss-Jordan Elimination (GJE) can be used to solve systems of XOR constraints. CryptoMiniSat [36] was the first SAT solver with deep integration of Gauss-Jordan Elimination into CDCL framework. Later, Han and Jiang proposed a new framework [16] building on Simplex-like techniques that performs Gauss-Jordan elimination, i.e., using reduced row echelon form instead of row echelon form. They used a two-watched variable scheme to detect propagations and conflicts in XOR constraints. Meel and Soos integrated Han and Jiang’s framework into their proposed architecture BIRD that sought to take advantage of both in-processing techniques and GJE. Recently, Soos, Gocht, and Meel [34] achieved acceleration in XOR unit propagation via exploiting bit-level parallelism offered in modern CPUs. In particular, they employed bit-packed integers to represent XOR rows in a matrix and apply bitwise operations, such as and, inverse, hamming weight, to quickly detect propagations and conflicts in XORs.

Lazy Reason Clause Generation

During XOR propagation, a reason clause will be generated to be used in future conflict analysis. However, during profiling the runtime of SAT solver, the generation process is quite time-consuming if the size of the XOR constraint involves thousands of variables. Furthermore, a large portion of reason clauses are never used during conflict analysis as not all assigned variables will be involved in the conflict as we apply the 1UIP policy. To reduce the overhead from the generation of useless reason clauses, Soos, Gocht, and Meel

proposed [34] a lazy generation method, which was based on the observation that once a literal is propagated by XOR propagation, the row of the XOR will preserve the propagated state until backtracking to the previous level. Therefore, the lazy method keeps an index of the row and the propagating literal but does not compute the reason clause eagerly. Whenever a reason clause is requested by conflict analysis, the reason clause is computed from the recorded row.

3.2 Conflict-Driven Pseudo-Boolean Solving

The past two decades have witnessed a rich array of techniques proposed in the context of PB solving (MiniSat+ [11], Open-WBO [23, 18], NaPS [30], Sat4j [20], PRS [10], Galena [3], Pueblo [33], RoundingSat [12]). Given the space considerations, we refer the reader to [28] for a detailed discussion on PB solvers and we will focus on providing a brief overview of the underlying PB solver, RoundingSat, in our work. RoundingSat employs a Conflict-Driven framework similar to the conflict-driven clause learning (CDCL) framework in CNF solving. The framework primarily extends conflict analysis and unit propagation from CNF to pseudo-Boolean constraints. RoundingSat employs cutting-planes based generalized resolution [7, 17, 9, 12] to resolve two PB constraints, which is exponentially stronger than resolution from a theoretical standpoint. In contrast to the two-watched literal scheme for CNF solving, RoundingSat employs a three-tiered approach where clauses, cardinality constraints, and general PB constraints were handled with different watched propagation techniques [3, 32, 20, 12, 8].

4 LinPB: An Efficient PB-XOR Solver

We now turn to the primary technical contribution of this work, our solver, LinPB, for PB-XOR formulas. As mentioned in Section 1, our first step (Section 4.1) was to lift the Lazy-GJE module inside CryptoMiniSat to RoundingSat. Observing that such a process did not yield any dividends compared to the baseline, we sought to investigate the key performance bottlenecks and accordingly propose three strategies: Shared-Watches, Eager-GJE, and Mixed-Watches, which seek to optimize the interaction between PB and XOR constraints. Since we keep the internal components of PB and GJE intact, our discussion in the rest of the section will focus on the interactions between the two components. In the rest of the section, we will use the term *PB propagation* to refer to propagations due to PB constraints and *XOR propagations* to refer to unit propagations due to XOR constraints via GJE.

4.1 Lazy Gauss-Jordan Elimination

We present the high-level overview of Lazy-GJE in Algorithm 1. We assume that the formula ϕ corresponds to PB constraints while the formula ψ corresponds to XOR constraints. Following CryptoMiniSat, we keep separate propagation indices for PB (q_ϕ) and XOR constraints (q_ψ). Trail ν represents the current assignment queue. The while loop at lines 3–7 performs PB propagation until we detect a conflict at line 6 or go through all assignments in ν . The while loop at lines 8–12 executes the similar procedure for XOR propagation. If neither PB nor XOR propagation detects a conflict, the unit propagation returns NULL at line 13. We refer to Algorithm 1 as a lazy method because GJE is invoked lazily, i.e., it is invoked only after all the unit propagations from PB constraints are processed.

As mentioned earlier, we observed that augmenting RoundingSat with Lazy-GJE did not lead to performance improvements over the baseline, CryptoMiniSat (wherein PB constraints are encoded into CNF). Upon further investigation, we observed a considerable performance

■ **Algorithm 1** Lazy Gauss Jordan Elimination.

```

1 Function propagationLazyGJE()
  Data: PB constraints  $\phi$ , XOR constraints  $\psi$ ,
  trail  $\nu$ , PB propagation index  $q_\phi$ , XOR propagation index  $q_\psi$ 
2 while  $q_\phi < \text{size}(\nu)$  or  $q_\psi < \text{size}(\nu)$  do
3   while  $q_\phi < \text{size}(\nu)$  do
4      $l_\phi \leftarrow \nu[q_\phi]$ 
5      $q_\phi \leftarrow q_\phi + 1$ 
6     if propagatePB( $\phi, l_\phi$ ) == conflict then
7       return conflict
8   while  $q_\psi < \text{size}(\nu)$  do
9      $l_\psi \leftarrow \nu[q_\psi]$ 
10     $q_\psi \leftarrow q_\psi + 1$ 
11    if propagateXOR( $\psi, l_\psi$ ) == conflict then
12      return conflict
13 return NULL

```

drop with the increase in the number of XOR constraints. A plausible primary reason for the behavior is that the Lazy-GJE delays conflict detection arising from XOR constraints, and the delay may lead to many redundant PB (unit) propagations. We illustrate such a scenario via an example.

► **Example 2. Hard instance for Lazy-GJE.** $\bigwedge_{i \in [1..n]} (x_i + \neg x_{i+1} \geq 1) \wedge \bigwedge_{j \in [0..\lfloor \frac{n}{3} \rfloor]} (x_{3j+1} \oplus x_{3j+2} \oplus x_{3j+3} = 1), n \gg 1$.

Suppose we select decision variables sequentially from x_1 to x_n and prefer negative polarity for the decision literal. Table 1 shows the procedure for a PB-XOR solver employing Lazy-GJE to solve Example 2. At level 1, the solver performs $O(n)$ PB propagations and produces $O(n)$ assignments. Then, the XOR propagation immediately detects a conflict, which, however, only involves the decision variable and first two variables implied at current level, while the rest of PB propagations are irrelevant to the conflict. The redundant PB propagations are reproduced every time the solver reaches level 1. In summary, the usage of Lazy-GJE leads to LinPB processing $O(n^2)$ redundant PB propagations. The scenario described above is reminiscent of the motivation of chronological backtracking [26], in which a solver may reassign many variables that are irrelevant to the conflict after non-chronological backtracking.

4.2 Eager-GJE

Table 1 demonstrates that lazy invocation of GJE may lead the solver to perform many redundant PB propagations in PB-XOR solving. Furthermore, Gauss Jordan Elimination is sound and complete, i.e., all unit propagations and conflicts implied by the given set of XORs would be discovered by a GJE-based decision procedure. Therefore, a natural reaction would be to invoke GJE in an eager fashion.

Algorithm 2 presents the propagation routine for Eager-GJE. Like Lazy-GJE, we use independent indexes for PB and XOR to track trail. Lines 2–6 perform PB propagation for literal l_ϕ . After each PB propagation, lines 7–11 go through all assignments in ν to detect all possible propagations and conflicts in XOR constraints via (incremental) GJE. We denote Algorithm 2 as an eager method because of the aggressive invocation of XOR propagations.

■ **Table 1** Procedure to solve Example 2 by Lazy-GJE. Column Level denotes the current decision level. Column Decision presents the decision literal at the current level. Column PB-propagation and XOR-propagation show the inferred assignments or the detected conflict by propagations. The last column specifies the conflict constraint, resolvents, and learned constraints in conflict analysis. Finally, jump to the next level if no conflict; otherwise, backtrack.

Level	Decision	PB propagation	XOR propagation	Conflict analysis
0	NIL	NIL	NIL	jump to level 1
1	$\neg x_1$	$\neg x_2, \neg x_3, \dots, \neg x_{n+1}$	conflict at $x_1 \oplus x_2 \oplus x_3 = 1$	conflict constraint $x_1 + x_2 + x_3 \geq 1$ resolve with $x_2 + \neg x_3 \geq 1$ and $x_1 + \neg x_2 \geq 1$ learn $x_1 \geq 1$ backtrack to level 0
0	NIL	x_1	NIL	jump to level 1
1	$\neg x_2$	$\neg x_3, \neg x_4, \dots, \neg x_{n+1}$	conflict at $x_4 \oplus x_5 \oplus x_6 = 1$	conflict constraint $x_4 + x_5 + x_6 \geq 1$ resolve with $x_5 + \neg x_6 \geq 1$ and $x_4 + \neg x_5 \geq 1$ learn $x_4 \geq 1$ backtrack to level 0
0	NIL	x_4, x_3, x_2	NIL	jump to level 1
repeat $k = 2, 3, \dots, \lfloor \frac{3}{n} \rfloor$				
1	$\neg x_{3k-1}$	$\neg x_{3k}, \neg x_{3k+1}, \dots, \neg x_{n+1}$	conflict at $x_{3k+1} \oplus x_{3k+2} \oplus x_{3k+3} = 1$	conflict constraint $x_{3k+1} + x_{3k+2} + x_{3k+3} \geq 1$ resolve with $x_{3k+2} + \neg x_{3k+3} \geq 1$ and $x_{3k+1} + \neg x_{3k+2} \geq 1$ learn $x_{3k+1} \geq 1$ backtrack to level 0
0	NIL	$x_{3k+1}, x_{3k}, x_{3k-1}$	NIL	jump to level 1

Our empirical evaluation indicates that while Eager-GJE is able to provide a remedy for some of the weaknesses of Lazy-GJE, the overhead due to GJE limits the scalability.

4.3 Shared-Watches

We now seek to take the middle road: we want to avoid both lazy and eager invocation of GJE. Our approach is to intermingle the PB and XOR propagations. Our proposed scheme, called Shared-Watches, is presented in Algorithm 3. Unlike the separate indexes for PB and XOR to trace propagation in Lazy-GJE, we use a shared index q for both constraints. At line 5 and 7, we detect PB and XOR propagation synchronously for each assignment l and terminate the unit propagation immediately if any of them detects a conflict.

We apply Shared-Watches to Example 2 and hold the same assumption that we select decision variables sequentially from x_1 to x_n and prefer negative polarity for each decision literal. Table 2 presents a shared-watches solver to solve Example 2. Every time at level 1, after a constant number (≤ 4) of PB propagations, the solver timely detects the conflict in XOR propagation. The fast detection of the conflict saves runtime from useless propagations, and then the solving time complexity is reduced to $O(n)$.

4.4 Mixed Watches

Our empirical analysis indicates that the key performance bottleneck for Shared-Watches and Eager-GJE is the computationally expensive (incremental) GJE that is invoked by propagateXOR. In order to reduce the overhead from XOR propagation and meantime *watch* XOR constraints timely, we propose Mixed-Watches. Mixed-Watches aims to *learn* partial PB constraints of interest implied by XOR constraints and add them to PB constraints. PB watches can detect partial XOR propagations and conflicts implied by equivalent PB constraints without access to XOR watches. In other words, Mixed-Watches reduce the invocation of XOR propagation but maintain the ability to *watch* XORs in a timely fashion. It is worth remarking that learning all PB constraints implied by a XOR constraint would either

■ **Algorithm 2** Eager-GJE.

```

1 Function propagationEagerGJE()
  Data: pseudo-Boolean constraints  $\phi$ , XOR constraints  $\psi$ ,
  trail  $\nu$ , propagation index  $q_\phi$ , XOR propagation index  $q_\psi$ 
2 while  $q_\phi < \text{size}(\nu)$  do
3    $l_\phi \leftarrow \nu[q_\phi]$  ;
4    $q_\phi \leftarrow q_\phi + 1$  ;
5   if propagatePB( $\phi, l_\phi$ ) == conflict then
6     return conflict
7   while  $q_\psi < \text{size}(\nu)$  do
8      $l_\psi \leftarrow \nu[q_\psi]$  ;
9      $q_\psi \leftarrow q_\psi + 1$  ;
10    if propagateXOR( $\psi, l_\psi$ ) == conflict then
11      return conflict
12 return NULL

```

■ **Algorithm 3** Shared-Watches.

```

1 Function propagationSharedWatches()
  Data: pseudo-Boolean constraints  $\phi$ , XOR constraints  $\psi$ , trail  $\nu$ , propagation
  index  $q$ 
2 while  $q < \text{size}(\nu)$  do
3    $l \leftarrow \nu[q]$ 
4    $q \leftarrow q + 1$ 
5   if propagatePB( $\phi, l$ ) == conflict then
6     return conflict
7   if propagateXOR( $\psi, l$ ) == conflict then
8     return conflict
9 return NULL

```

necessitate the addition of a large number of auxiliary variables or storage of exponentially many (in the size of XORs) PB constraints. Therefore, the quality of learned PB constraints from XOR is of significant importance.

We propose to learn both conflict and reason constraints used by conflict analysis (CA-reason) from XOR constraints since the conflict and propagation play an essential role in CDCL, and these constraints are likely to be triggered again in the future. Algorithm 4 presents the pseudocode for conflict analysis in PB-XOR solving with Mixed-Watches. Lines 2–3 add the conflict constraint (C_{conf}) to learned PB constraints if C_{conf} is detected from XOR propagation. Lines 4–11 perform conflict analysis. We retrieve the last assignment l from trail ν at line 5. If l in the conflict constraint, we fetch the reason constraint (C_{reason}) at Line 7. If the reason constraint is generated from a XOR constraint, we add C_{reason} to learned PB constraints at lines 8–9. The conflict constraint resolve with the reason constraint at line 10. The last assignment is removed from the trail ν at line 11, and then the next iteration starts. Finally, the function returns the constraint after analysis at line 12. In

■ **Table 2** Procedure to solve Example 2 by Shared-Watches. Column Level denotes the current decision level. Column Decision presents the decision literal at the current level. Column PB-propagation and XOR-propagation show the inferred assignments or the detected conflict by propagations. The last column specifies the conflict constraint, resolvents, and learned constraints in conflict analysis. Finally, jump to the next level if no conflict; otherwise, backtrack.

Level	Decision	PB propagation	XOR propagation	Conflict analysis
0	NIL	NIL	NIL	jump to level 1
1	$\neg x_1$	$\neg x_2, \neg x_3$	conflict at $x_1 \oplus x_2 \oplus x_3 = 1$	conflict constraint $x_1 + x_2 + x_3 \geq 1$ resolve with $x_2 + \neg x_3 \geq 1$ and $x_1 + \neg x_2 \geq 1$ learn $x_1 \geq 1$ backtrack to level 0
0	NIL	x_1	NIL	jump to level 1
1	$\neg x_2$	$\neg x_3, \neg x_4, \neg x_5, \neg x_6$	conflict at $x_4 \oplus x_5 \oplus x_6 = 1$	conflict constraint $x_4 + x_5 + x_6 \geq 1$ resolve with $x_5 + \neg x_6 \geq 1$ and $x_4 + \neg x_5 \geq 1$ learn $x_4 \geq 1$ backtrack to level 0
0	NIL	x_4, x_3, x_2	NIL	jump to level 1
repeat $k = 2, 3, \dots, \lfloor \frac{3}{n} \rfloor$				
1	$\neg x_{3k-1}$	$\neg x_{3k}, \neg x_{3k+1}, \neg x_{3k+2}, \neg x_{3k+3}$	conflict at $x_{3k+1} \oplus x_{3k+2} \oplus x_{3k+3} = 1$	conflict constraint $x_{3k+1} + x_{3k+2} + x_{3k+3} \geq 1$ resolve with $x_{3k+2} + \neg x_{3k+3} \geq 1$ and $x_{3k+1} + \neg x_{3k+2} \geq 1$ learn $x_{3k+1} \geq 1$ backtrack to level 0
0	NIL	$x_{3k+1}, x_{3k}, x_{3k-1}$	NIL	jump to level 1

Section 5, we use a portfolio method to empirically show that learning both conflict and CA-reason constraints is the best learning heuristic, and Mixed-Watches cooperates well with Lazy-GJE.

5 Experimental Evaluation

We equipped the state-of-the-art pseudo-Boolean solver RoundingSat[12] with proposed PB-XOR tactics and called the resulting solver LinPB. To showcase the impact of LinPB, we integrated LinPB into the state-of-the-art hashing-based counting technique ApproxMC, implementing the first pseudo-Boolean model counter, ApproxMCPB. We conducted an extensive study on 1076 benchmarks¹ arising from quantitative verification of binarized neural networks with respect to different properties such as robustness, trojan attack, and fairness. These benchmarks represent a wide range of security applications where quality and runtime performance of counters are key determining factors [2]. To evaluate the performance of ApproxMCPB, we performed a comparison with state-of-the-art CNF projected counting techniques ApproxMC4 [34], Ganak[31], GPMC[29] and projMC[19]. We used CNF encoding as described in [2], and equivalent pseudo-Boolean encoding² for ApproxMCPB. We developed the PB counter employing PB encoding of XOR constraints as another baseline.³

Experiments were conducted on a high-performance computer cluster, each node consisting of 2xE5-2690v3 CPUs with 2x12 real cores and 96GB of RAM. We set the time limit as 5000 seconds and the memory limit as 4GB for each counter per benchmark. Keeping in line with the prior work, we set the confidence factor $\delta = 0.2$ and tolerance factor $\epsilon = 0.8$ by default for approximate counters. We used the number of solved benchmarks and PAR-2 score to evaluate the performance. The PAR-2 score represents the average running time on benchmarks with a doubling-time penalty on timeout benchmarks.

¹ The benchmarks are available at <https://teobaluta.github.io/NPAQ/#benchmarks>.

² Refer to Appendix A for PB encoding.

³ The baseline solved nearly 200 fewer benchmarks than ApproxMCPB and thereby of no interest to us.

■ **Algorithm 4** Mixed-Watches.

```

1 Function conflictAnalysisMixedWatches( $C_{confl}, \nu$ )
  Data: conflict constraint  $C_{confl}$ , trail  $\nu$ 
2 if  $C_{confl}$  is from XOR propagation then
3    $\lfloor$  AddConstraintToPB( $C_{confl}$ )
4 while  $C_{confl}$  is not asserting do
5    $l \leftarrow \text{getLast}(\nu)$ 
6   if  $\neg l$  in  $C_{confl}$  then
7      $C_{reason} \leftarrow \text{getReason}(l)$ 
8     if  $C_{reason}$  is from XOR propagation then
9        $\lfloor$  AddConstraintToPB( $C_{reason}$ )
10     $C_{confl} \leftarrow \text{resolve}(C_{confl}, C_{reason})$ 
11   $\nu \leftarrow \text{removeLast}(\nu)$ 
12 return  $C_{confl}$ 

```

The objective of our experimental evaluation is to analyze the performance of ApproxMCPB both in terms of runtime and approximation quality. In particular, we sought to answer the following questions:

RQ 1 How does the performance of GJE tactics for ApproxMCPB?

RQ 2 How does the runtime performance of ApproxMCPB compare with ApproxMC4 and other state-of-the-art projected counting techniques?

RQ 3 How far are the counts computed by ApproxMCPB from the exact counts?

In summary, the usage of Lazy-GJE leads to ApproxMCPB solving 804 instances while usage of Shared-Watches, Eager-GJE, and Mixed-Watches allows ApproxMCPB solve 892, 892 and 912 instances respectively. While the state-of-the-art tool, ApproxMC4, can only solve 802 instances, ApproxMCPB can solve 912 instances, an increment of 110 instances. Furthermore, the PAR-2 score for ApproxMC4 is 3305 seconds while PAR-2 score for ApproxMCPB is 1822 seconds, thereby achieving almost 50% decrease in PAR-2 score. Moreover, the speedup of ApproxMCPB to ApproxMC4 is independent of the number of solutions. In terms of approximation quality, the average observed tolerance is 0.037, far better than the theoretical guarantee of 0.8.

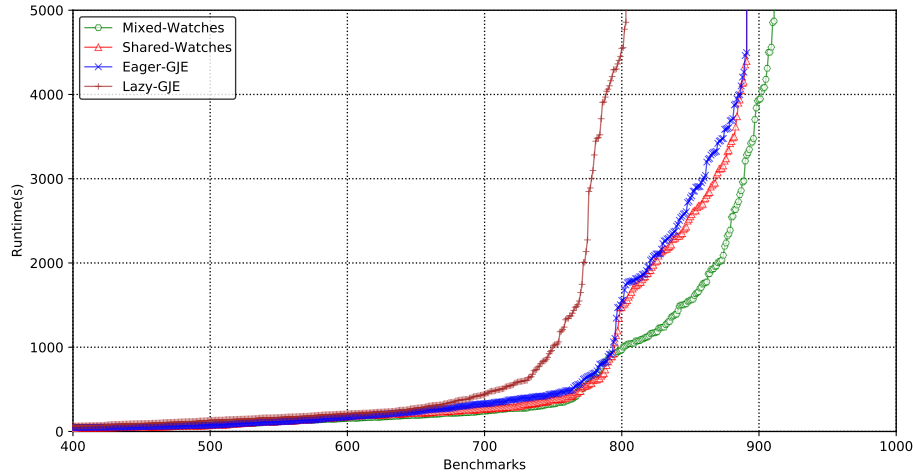
5.1 Performance of GJE tactics

This section evaluates the performance of ApproxMCPB augmented with different PB-XOR tactics: Lazy-GJE, Eager-GJE, Shared-Watches, and Mixed-Watches⁴. Table 3 summarizes the results. ApproxMCPB augmented with Lazy-GJE solved only 804 of 1076 benchmarks while ApproxMCPB augmented with Shared-Watches, Eager-GJE, and Mixed-Watches solved 892, 892, and 912 instances respectively, thereby achieving a gain of over 100 instances. Furthermore, the PAR-2 score for the usage of Lazy-GJE is 2755 seconds while the PAR-2 score for the usage of Shared-Watches, Eager-GJE, and Mixed-Watches is 2017, 2042, 1822 seconds respectively, thereby achieving a decrease of over 700 seconds. Observe that the usage of Mixed-Watches leads to ApproxMCPB solving 20 more instances than the other tactics.

⁴ See Appendix B for the optimal configuration of Mixed-Watches

■ **Table 3** The number of solved benchmarks for ApproxMCPB configured with different Gauss Jordan Elimination tactics. PAR-2 score is in parentheses. Mixed-Watches applies the heuristic of learning both conflict and CA-reason constraints based on Lazy-GJE. Time out after 5000s.

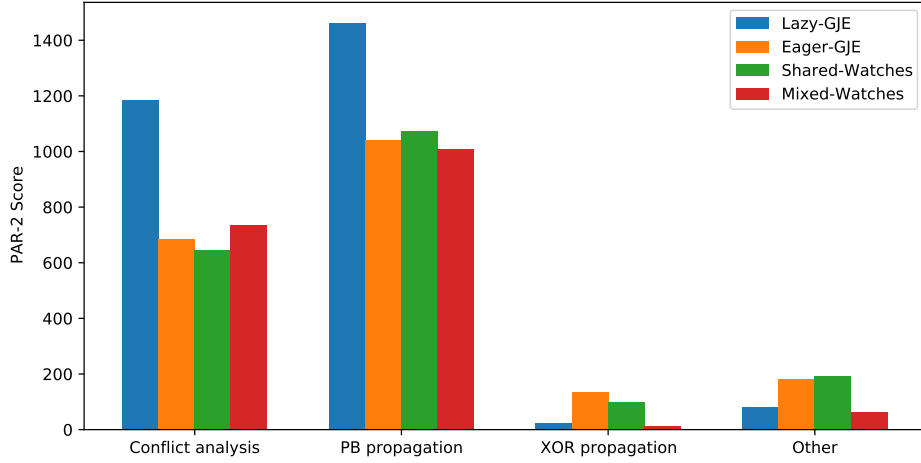
Total	Lazy-GJE	Eager-GJE	Shared-Watches	Mixed-Watches
1076 (PAR-2)	804 (2755)	892 (2042)	892 (2017)	912 (1822)



■ **Figure 1** Runtime for ApproxMCPB of different GJE tactics on 1076 BNN benchmarks. The x-axis represents the number of solved benchmarks, while the y-axis shows the counting time. A point (x, y) represents that x benchmarks can be solved within y seconds. The number of solved benchmarks sorts counters in descending order in the legend.

Figure 1 presents the cactus plot of the performance of different tactics. We present the number of solved benchmarks on the x-axis and the time taken on the y-axis. A point (x, y) represents that x benchmarks can be solved within y seconds for the particular tactic. We observed that all the curves almost converge to an overlapped curve before the 300-second runtime threshold, which means the usage of different tactics makes ApproxMCPB solve a similar number of benchmarks within a runtime threshold less than 300 seconds. Then, Lazy-GJE begins to diverge and leads to ApproxMCPB solving fewer benchmarks than other tactics with the same runtime threshold. Eager-GJE and Shared-Watches diverge together at around 1000-second threshold, and Shared-Watches slightly outperforms Eager-GJE after diversion. The observation reveals that the usage of Mixed-Watches always leads to ApproxMCPB solving no fewer benchmarks than other tactics no matter what runtime threshold is used. Similarly, the usage of Shared-Watches always produces a no worse result than Eager-GJE and Lazy-GJE. In summary, Eager-GJE, Shared-Watches, and Mixed-Watches successively extend the reach of ApproxMCPB.

Runtime breakdown. To analyze the time consumption of main procedures, we profile the runtime breakdown for conflict analysis, PB propagation, XOR propagation, and others. For each procedure, we sum the runtime over solved benchmarks and compute the proportion in total runtime. Then, we calculate the PAR-2 score and proportionally break it down into the four procedures. Figure 2 presents the breakdown of PAR-2 score. The x-axis shows the main procedures in PB-XOR solving, while the y-axis presents the PAR-2 score proportionally taken by each procedure. Colors represent different GJE tactics. We observed that Lazy-GJE



■ **Figure 2** Breakdown of PAR-2 score. We proportionally break down the PAR-2 score into four procedures according to the runtime taken by each procedure. The x-axis shows the four main procedures in PB-XOR solving, while the y-axis presents the PAR-2 score proportionally taken by each procedure.

spends much more time on conflict analysis and PB propagation than other tactics, while Eager-GJE and Shared-Watches spend more time on XOR propagation and other procedures. Particularly, Mixed-Watches spends relatively less time on all procedures among four tactics. The observation reveals that the usage of Eager-GJE and Shared-Watches indeed incurs more overhead from XOR-propagation and other procedures. The usage of Mixed-Watches leads to ApproxMCPB achieving a similar efficiency in conflict analysis and PB-propagation as Eager-GJE and Shared-Watches while maintaining a small overhead from XOR-propagation and other procedures, thereby emerging as the most efficient tactic.

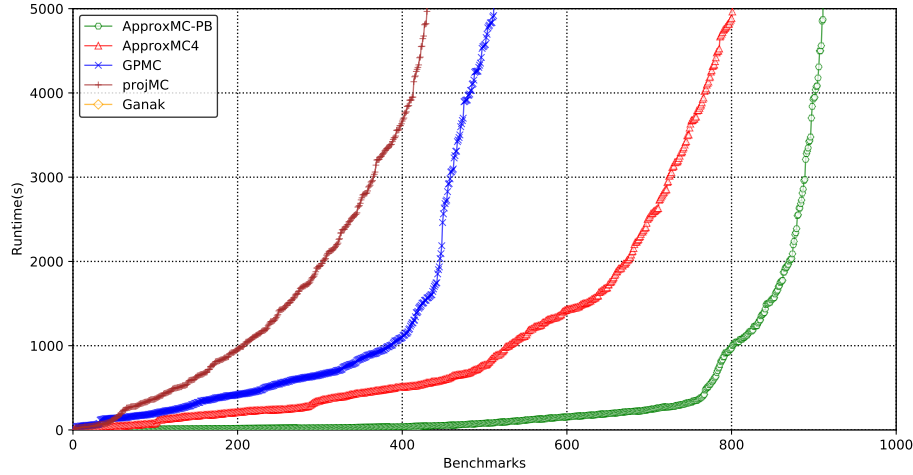
5.2 Performance vs. State-of-the-art Projected Counting Techniques

Since the design of LinPB was motivated by model counting applications, we present an empirical comparison of ApproxMCPB vis-a-vis other state-of-the-art counting techniques. For all the results in this section, we equip ApproxMCPB with Mixed-Watches tactic. Table 4 summarizes the results. We observed that state-of-the-art techniques can solve at most 802 of 1076 instances while ApproxMCPB can solve 912 instances, thereby achieving a gain of over 100 instances. The PAR-2 score for state-of-the-art techniques is at least 3305 seconds while the PAR-2 score for ApproxMCPB is 1822 seconds, thereby achieving an almost 50% decrease in PAR-2 score. Particularly, the exact counting techniques can solve only 511 instances, roughly half of ApproxMCPB. Therefore, ApproxMCPB significantly outperforms state-of-the-art projected counting techniques. Figure 3 presents the number of solved benchmarks in terms of the runtime threshold for all counters. The righter the curve is, the more benchmarks the counter can solve within a runtime threshold. We observed that ApproxMCPB can always solve more instances than other techniques given any runtime threshold.

Dependence on #Solutions. We now analyze how the speedup achieved by ApproxMCPB varies with the #Solutions. To this end, Figure 4 presents the speedup of the ApproxMCPB to ApproxMC4 on the y-axis with the #Solutions. We selected benchmarks solved by ApproxMCPB or ApproxMC4. Each point represents a benchmark. The x-axis presents

■ **Table 4** Number of solved benchmarks for ApproxMCPB vs. state-of-the-art projected model counting techniques. PAR-2 score is in the parentheses. ApproxMCPB uses the best configuration of Mixed-Watches. Time out after 5000s.

Total	Exact		Probabilistic	Exact	Approximate	
	GPMC	projMC	Ganak		ApproxMC4	ApproxMCPB
1076 (PAR-2)	511 (5713)	430 (6584)	1 (9991)		802 (3305)	912 (1822)



■ **Figure 3** Runtime for ApproxMCPB vs. state-of-the-art projected model counters. The x-axis represents the number of solved benchmarks, while the y-axis shows the counting time. A point (x, y) represents that x benchmarks can be solved within y seconds. Ganak can solve only one instance and therefore fails to be plotted. The number of solved benchmarks sorts counters in descending order in the legend. Time out 5000s.

the number of solutions of the benchmark in the log2 scale⁵, while the y-axis represents the speedup, i.e., the counting-time⁶ ratio of ApproxMC4 to the ApproxMCPB $\frac{T_{CNF}}{T_{PB}}$ on the benchmark. The horizontal gray line highlights the boundary of speedup $y = 1$.

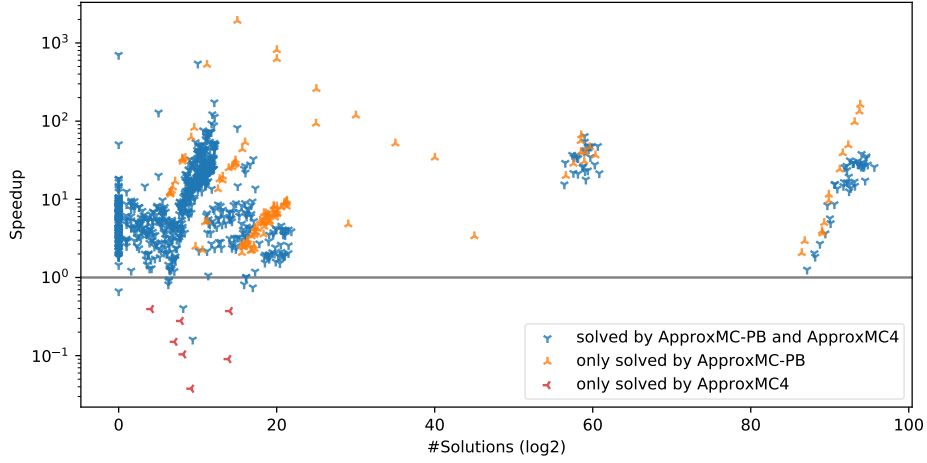
We observed that almost all points are above the horizontal line, indicating ApproxMCPB outperforms ApproxMC4 on most instances. Even though most instances can be both solved, ApproxMCPB can solve over 100 instances beyond the reach of ApproxMC4 while ApproxMC4 only solved 7 instances beyond the reach of ApproxMCPB. Furthermore, the speedup of ApproxMCPB to ApproxMC4 randomly falls into the interval $[10^0, 10^2]$ on almost all benchmarks. Therefore, ApproxMCPB is able to achieve consistent speedup over the entire spectrum of #Solutions.

5.3 Correctness

To evaluate the approximation quality, we compare the counts computed by approximate model counters with counts returned by exact model counters. Figure 5 shows the model counts computed by ApproxMCPB, and the bounds obtained by scaling the exact counts

⁵ Given that the estimation is $a * 2^b$, we denote the log value as $b + \log_2(a + 1)$ to avoid invalid $\log_2(0)$.

⁶ Drawing from the definition of PAR-2 score, we double the runtime if the benchmark is unsolved within the time limit.



■ **Figure 4** Speedup of ApproxMCPB to ApproxMC4 in terms of the number of solutions of benchmarks. Speedup represents counting-time ratio of ApproxMC4 to ApproxMCPB. #Solutions is in the log2 scale. The horizontal gray line denotes the boundary of speedup $y = 1$. The runtime is doubled if unsolved within the time limit (5000s).

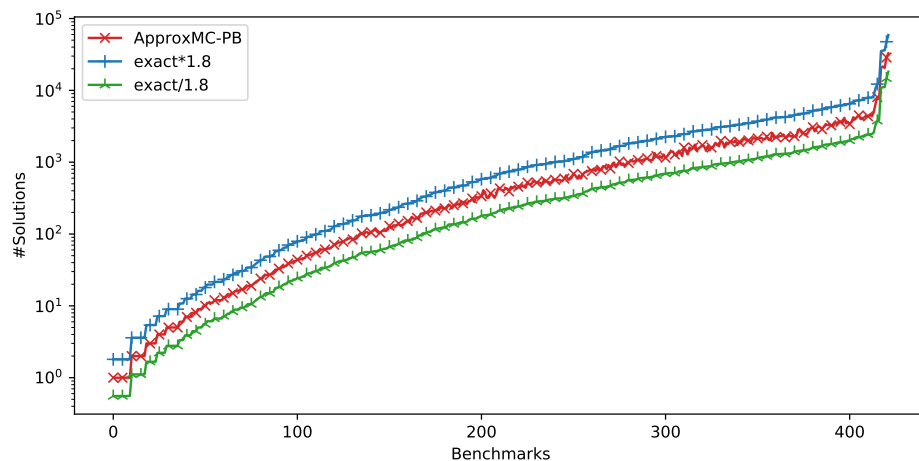
with the tolerance factor ($\epsilon = 0.8$). We selected benchmarks solved by at least one exact counter. All exact counts from the same benchmark are equal. The exact count sorts benchmarks in ascending order on the x-axis, while the y-axis represents the model count. We observed that for all the benchmarks, ApproxMCPB computed counts within the tolerance. Furthermore, for each instance, the observed tolerance (ϵ_{obs}) was calculated as $\max(\frac{|sol(F)|}{ApproxCount} - 1, \frac{ApproxCount}{|sol(F)|} - 1)$, where *ApproxCount* is the estimate by ApproxMCPB. We observed that the arithmetic mean of ϵ_{obs} across all benchmarks is 0.037 - far better than the theoretical guarantee of 0.8.

Furthermore, we observed that the estimates of ApproxMCPB always match that of ApproxMC4. Recall that the hashing-based approximate counting technique is to employ randomly generated XOR constraints to partition the solution space into roughly equal small cells and count the number of solutions in a randomly picked cell to estimate the total number of solutions. We used the same random seed for ApproxMCPB and ApproxMC4. Hence, both counters always generated the same set of XOR constraints and counted the same cell to estimate the #Solutions.

6 Conclusion and Discussion

In this paper, we focused on the design of LinPB, a solver with native support for PB-XOR formulas. The need for LinPB was motivated by the recent surge of interest in verification of (binarized) neural networks wherein the quantitative verification queries were shown to reduce to model counting. Binarized neural networks can be naturally represented as PB constraints while hashing-based techniques reduce counting to polynomially many SAT queries wherein the original formula is conjuncted with random XOR constraints.

We observed that a straightforward adaptation of the Lazy-GJE approach does not yield performance improvements. Our empirical investigations highlighted the importance of the interaction of PB and XOR propagations. To this end, we designed three propagation strategies: Eager-GJE, Shared-Watches, and Mixed-Watches. We demonstrate the effectiveness of LinPB by augmenting it with the state-of-the-art hashing-based algorithm, ApproxMC; we call the



■ **Figure 5** Plot showing counts obtained by ApproxMCPB vis-a-vis exact counts.

resulting counter, ApproxMCPB. Our empirical evaluation demonstrates ApproxMCPB is able to solve 110 more benchmarks than the baseline approach with a decrease of PAR-2 score by 1483.

The runtime performance of LinPB opens up several interesting directions of future work. We sketch out two directions of particular interest. First, it is worth observing that, unlike modern CNF solvers, the PB solvers are still in the nascent phase, and consequently lack intricate efficient preprocessing techniques. Therefore, in our design of LinPB, we did not adapt the BIRD architecture [35], which was designed to efficiently transform XOR constraints between clauses and native representation, aiming to utilize the powerful inprocessing technique of CNF. The development of efficient inprocessing for PB constraints would invite extending LinPB with a BIRD-eseque architecture. Secondly, the significant performance improvements of Mixed-Watches over Eager-GJE and Shared-Watches leads us to speculate that adoption of these strategies in the context of CNF-XOR solving would also lead to performance improvements.

References

- 1 Dimitris Achlioptas, Zayd S. Hammoudeh, and Panos Theodoropoulos. Fast sampling of perfectly uniform satisfying assignments. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 135–147, Cham, 2018. Springer International Publishing.
- 2 Teodora Baluta, Shiqi Shen, Shweta Shine, Kuldeep S. Meel, and Prateek Saxena. Quantitative verification of neural networks and its security applications. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, November 2019.
- 3 D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):305–317, 2005. doi: 10.1109/TCAD.2004.842808.
- 4 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In *Proceedings of International Conference on Constraint Programming (CP)*, pages 200–216, September 2013.
- 5 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Improving approximate counting for probabilistic inference: From linear to logarithmic sat solver calls. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, July 2016.

- 6 Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71*, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery. doi:10.1145/800157.805047.
- 7 W. Cook, C.R. Coullard, and Gy. Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, 1987. doi:10.1016/0166-218X(87)90039-4.
- 8 Jo Devriendt. Watched propagation of - integer linear constraints. In *Proceedings of International Conference on Constraint Programming (CP)*, pages 160–176, September 2020. doi:10.1007/978-3-030-58475-7_10.
- 9 Heidi Dixon, Matt Ginsberg, and Andrew Parkes. Generalizing boolean satisfiability i: Background and survey of existing work. *J. Artif. Intell. Res. (JAIR)*, 21:193–243, February 2004. doi:10.1613/jair.1353.
- 10 Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *Eighteenth National Conference on Artificial Intelligence*, page 635–640, USA, 2002. American Association for Artificial Intelligence.
- 11 N. Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *J. Satisf. Boolean Model. Comput.*, 2:1–26, 2006.
- 12 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 1291–1299. International Joint Conferences on Artificial Intelligence Organization, July 2018. doi:10.24963/ijcai.2018/180.
- 13 Stefano Ermon, Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Taming the curse of dimensionality: Discrete integration by hashing and optimization. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28, ICML'13*, page II–334–II–342. JMLR.org, 2013.
- 14 Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting: A new strategy for obtaining good bounds. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1, AAAI'06*, page 54–61. AAAI Press, 2006.
- 15 Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985. Third Conference on Foundations of Software Technology and Theoretical Computer Science. doi:10.1016/0304-3975(85)90144-6.
- 16 Cheng-Shen Han and Jie-Hong Roland Jiang. When boolean satisfiability meets gaussian elimination in a simplex way. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 410–426, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- 17 John Hooker. Generalized resolution for 0–1 linear inequalities. *Annals of Mathematics and Artificial Intelligence*, 6:271–286, March 1992. doi:10.1007/BF01531033.
- 18 Saurabh Joshi, Ruben Martins, and Vasco M. Manquinho. Generalized totalizer encoding for pseudo-boolean constraints. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 200–209. Springer, 2015. doi:10.1007/978-3-319-23219-5_15.
- 19 Jean-Marie Lagniez and Pierre Marquis. A recursive algorithm for projected model counting. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):1536–1543, July 2019. doi:10.1609/aaai.v33i01.33011536.
- 20 Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7:59–6, January 2010.
- 21 Leonid A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3), 1973.
- 22 J.P. Marques-Silva and K.A. Sakallah. Grasp: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999. doi:10.1109/12.769433.
- 23 Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-wbo: A modular maxsat solver,. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 438–445, Cham, 2014. Springer International Publishing.

- 24 Kuldeep S. Meel. *Constrained Counting and Sampling: Bridging the Gap between Theory and Practice*. PhD thesis, Rice University, 2017.
- 25 Kuldeep S. Meel, Moshe Y. Vardi, Supratik Chakraborty, Daniel J. Fremont, Sanjit A. Seshia, Dror Fried, Alexander Ivrii, and Sharad Malik. Constrained sampling and counting: Universal hashing meets sat solving. In *Proceedings of Workshop on Beyond NP(BNP)*, 2016.
- 26 Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 111–121, Cham, 2018. Springer International Publishing.
- 27 Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. Verifying Properties of Binarized Deep Neural Networks. *arXiv e-prints*, page arXiv:1709.06662, 2017. [arXiv:1709.06662](https://arxiv.org/abs/1709.06662).
- 28 Olivier Roussel and Vasco Manquinho. Pseudo-boolean and cardinality constraints. In *Handbook of satisfiability*, pages 695–733. IOS Press, 2009.
- 29 Kenji Hashimoto Ryosuke Suzuki and Masahiko Sakai. Improvement of projected model-counting solver with component decomposition using sat solving in components. *JSAT Technical Report*, SIG-FPAI-103-B506:31–36, March 2017. in Japanese.
- 30 Masahiko SAKAI and Hidetomo NABESHIMA. Construction of an robdd for a pb-constraint in band form and related techniques for pb-solvers. *IEICE Transactions on Information and Systems*, E98.D(6):1121–1127, 2015. [doi:10.1587/transinf.2014F0P0007](https://doi.org/10.1587/transinf.2014F0P0007).
- 31 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. Ganak: A scalable probabilistic exact model counter. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2019.
- 32 H.M. Sheini and K.A. Sakallah. Pueblo: a modern pseudo-boolean sat solver. In *Design, Automation and Test in Europe*, pages 684–685 Vol. 2, 2005. [doi:10.1109/DATE.2005.246](https://doi.org/10.1109/DATE.2005.246).
- 33 Hossein Sheini and Karem Sakallah. Pueblo: A hybrid pseudo-boolean sat solver. *JSAT*, 2:165–189, March 2006. [doi:10.3233/SAT190020](https://doi.org/10.3233/SAT190020).
- 34 Mate Soos, Stephan Gocht, and Kuldeep S. Meel. Tinted, detached, and lazy cnf-xor solving and its applications to counting and sampling. In *Proceedings of International Conference on Computer-Aided Verification (CAV)*, July 2020.
- 35 Mate Soos and Kuldeep S. Meel. Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, January 2019.
- 36 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending sat solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 244–257, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- 37 G. S. Tseitin. On the complexity of derivation in propositional calculus. *Automation of Reasoning*, pages 466–483, 1983. [doi:10.1007/978-3-642-81955-1_28](https://doi.org/10.1007/978-3-642-81955-1_28).
- 38 Alasdair Urquhart. Hard examples for resolution. *J. ACM*, 34(1):209–219, 1987. [doi:10.1145/7531.8928](https://doi.org/10.1145/7531.8928).

A Pseudo-Boolean Encoding of Binarized Neural Network

Conditional pseudo-Boolean constraint is a fundamental building block to binarized neural network (BNN). We introduce the pseudo-Boolean encoding of conditional pseudo-Boolean constraint in Definition 3. Then we sketch the idea to encode BNN on top of conditional pseudo-Boolean constraints.

► **Definition 3.** Given a conditional pseudo-Boolean constraint $\phi : y \rightarrow \sum_{i=1}^N w_i x_i$ op b , op $\in \{\geq, \leq\}$, $w_i, b \in \mathbb{Z}$, $x_i, y \in \{0, 1\}$, we define as the pseudo-Boolean encoding of ϕ :

$$\begin{cases} C = \sum_{i=1}^N |w_i| \\ \sum_{i=1}^N w_i x_i + (b + C) \neg y \geq b & \text{if op is } \geq \\ \sum_{i=1}^N w_i x_i + (b - C) \neg y \leq b & \text{if op is } \leq \end{cases} \quad (2)$$

The neuron, a basic component in binarized neural network can be logically represented by the constraint:

$$y \leftrightarrow \sum_{i=1}^N w_i x_i + b \geq 0 \quad (3)$$

in which $w_i \in \{+1, -1\}$. Formula (3) is equivalent to:

$$y \rightarrow \sum_{i=1}^N w_i x_i + b \geq 0 \wedge \neg y \rightarrow \sum_{i=1}^N w_i x_i + b < 0 \quad (4)$$

Finally, we encode formula (4) based on conditional pseudo-Boolean constraints introduced in Definition 3. The complete encoding is shown as follows.

Let's consider the k -th block of BNN: $BLK_k : v^k \rightarrow v^{k+1}$ ($v^k, v^{k+1} \in \{1, -1\}^N$) including

1. linear layer(w^k, b^k) : $v^k \rightarrow o^{lin}$
 $o_i^{lin} = \sum_{i=1}^N w_i^k v_i^k + b_i^k$
 $(v^k, w^k \in \{+1, -1\}^N, b, o^{lin} \in \mathbb{R}^N)$
2. batch normalization layer($\mu^k, \sigma^k, \alpha^k, \gamma^k$) : $o^{lin} \rightarrow o^{bn}$
 $o_i^{bn} = \frac{o_i^{lin} - \mu_i^k}{\sigma_i^k} \cdot \alpha_i^k + \gamma_i^k$
 $(\mu^k, \sigma^k, \alpha^k, \gamma^k, o^{lin}, o^{bn} \in \mathbb{R}^N)$
3. binarization layer: $o^{bn} \rightarrow v^{k+1}$
 $v_i^{k+1} = 1 \leftrightarrow o_i^{bn} \geq 0$
 $(v^{k+1} \in \{1, -1\}^N, o^{bn} \in \mathbb{R}^N)$

According to the encoding in “Quantitative Verification of Neural Networks and Its Security Applications” by Teo, we can get the following constraint for each neuron when $\alpha > 0$ (In following constraints $v_i^{k+1}, v_i^k \in \{0, 1\}$ because we have transferred them into boolean variables):

$$\begin{cases} v_i^{k+1} = 1 \leftrightarrow \sum_{i=1}^N w_i^k v_i^k \geq C_i'^k \\ C_i'^k = \left\lceil \frac{C_i^k + \sum_{i=1}^N w_i^k}{2} \right\rceil \\ C_i^k = \left\lceil -\frac{\sigma_i^k}{\alpha_i^k} \gamma_i^k + \mu_i^k - b_i^k \right\rceil \end{cases} \quad (5)$$

By Eq. 2, Eq 3, Eq. 4, we can get:

$$\begin{cases} \sum_{i=1}^N w_i^k v_i^k + \beta_i^k \neg v_i^{k+1} \geq C_i'^k \\ \beta_i^k = C_i'^k + N \\ -\sum_{i=1}^N w_i^k v_i^k + \beta_i'^k v_i^{k+1} \geq 1 - C_i'^k \\ \beta_i'^k = N + 1 - C_i'^k \end{cases} \quad (6)$$

Note that β_i^k and $\beta_i'^k$ are constants. The other two are linear encoding. Similarly we can get constraints for $\alpha < 0$:

$$\begin{cases} v_i^{k+1} = 1 \leftrightarrow \sum_{i=1}^N w_i^k v_i^k \leq C_i'^k \\ C_i'^k = \left\lfloor \frac{C_i^k + \sum_{i=1}^N w_i^k}{2} \right\rfloor \\ C_i^k = \left\lfloor -\frac{\sigma_i^k}{\alpha_i^k} \gamma_i^k + \mu_i^k - b_i^k \right\rfloor \\ -\sum_{i=1}^N w_i^k v_i^k + \beta_i^k v_i^{k+1} \geq -C_i'^k \\ \beta_i^k = -C_i'^k + N \\ \sum_{i=1}^N w_i^k v_i^k + \beta_i'^k v_i^{k+1} \geq 1 + C_i'^k \\ \beta_i'^k = N + 1 + C_i'^k \end{cases} \quad (7)$$

Corner case when $\alpha = 0$:

$$v_i^{k+1} = 1 \leftrightarrow \gamma_i^k \geq 0 \quad (8)$$

B Configuration of Mixed-Watches

In this section, we focus on examining the integration compatibility of Mixed-Watches with Lazy-GJE, Eager-GJE, and Shared-Watches with different heuristics for Mixed-Watches. To this end, we focus on the following heuristics:

1. CA-reason: Addition of the reason constraint generated from XOR-propagation and used by conflict analysis to PB constraints.
2. All-reason: Addition of all the reason constraints generated from XOR-propagation to PB constraints.
3. Confl: Addition of the conflict constraints detected from XOR-propagation to PB constraints.

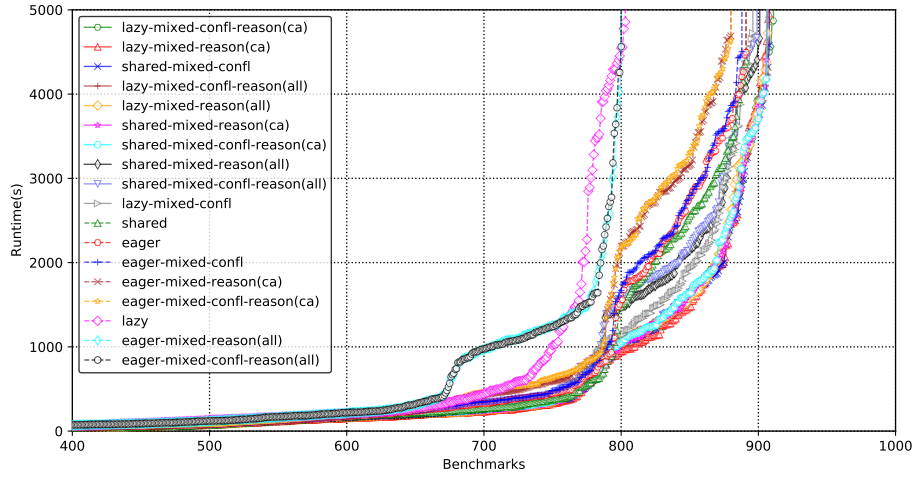
■ **Table 5** Number of solved benchmarks for Mixed-Watches integrated with other GJE tactics and applying different heuristics. PAR-2 score is in the parentheses. The heuristic means adding the corresponding constraints from XOR-propagation to PB constraints. Time out after 5000s.

GJE tactics	No mixed ⁷	Heuristics				
		CA-reason	All-reason	Confl	CA-reason \cup Confl	All-reason \cup Confl
Lazy-GJE	804 (2755)	909 (1834)	908 (1855)	897 (1961)	912 (1822)	908 (1854)
Eager-GJE	892 (2042)	881 (2172)	801 (2776)	889 (2071)	881 (2180)	801 (2777)
Shared-Watches	892 (2017)	907 (1850)	902 (1951)	909 (1839)	907 (1860)	900 (1976)

⁷ The original tactic without Mixed-Watches.

Table 5 summarizes the results. The first column shows the GJE tactic integrated with Mixed-Watches. The second column presents the number of solved benchmarks and PAR-2 score in the parentheses for the original tactic without Mixed-Watches, while the following columns show the result for GJE tactics integrated with different heuristics of Mixed-Watches. The third, fourth, and fifth columns refer to CA-reason, All-reason, and Confl heuristics while the last two columns refer to combination of the aforementioned heuristics.

The bold cell in Table 5 highlights that a Mixed-Watches integrated with Lazy-GJE and employing CA-reason and Confl heuristics, solved the most number of benchmarks and achieved the smallest PAR-2 score. Furthermore, we observe that Mixed-Watches improves the performance Lazy-GJE by around one hundred more solved benchmarks from 804 to 912 and improves the performance of Shared-Watches by a dozen solved benchmarks while making



■ **Figure 6** Runtime for Mixed-Watches integrated with other GJE tactics and applying different heuristics. The x-axis represents the number of solved benchmarks, while the y-axis shows the counting time. A point (x, y) represents that x benchmarks can be solved within the runtime threshold y . The number of solved benchmarks sorts counters in descending order in the legend. Time out after 5000s.

Eager-GJE solve fewer benchmarks than the original tactic. On the other hand, Table 5 summarizes that learning both conflict and reason constraints used by conflict analysis (CA-reason) from XOR-propagation is the best heuristic for Lazy-GJE based Mixed-Watches. All heuristics involving CA-reason constraints always solves more benchmarks than All-reason.

To provide a comprehensive picture, we present the cactus plot in Figure 6 for different combinations. The legend of the Figure has all the combinations sorted in descending order by the number of solved benchmarks.

Automated Random Testing of Numerical Constrained Types

Ghiles Ziat ✉

ISAE-SUPAERO, Université de Toulouse, France

Matthieu Dien ✉

Université de Caen, France

Vincent Botbol ✉

Nomadic labs, Paris, France

Abstract

We propose an automated testing framework based on constraint programming techniques. Our framework allows the developer to attach a *numerical* constraint to a type that restricts its set of possible values. We use this constraint as a partial specification of the program, our goal being to derive *property-based* tests on such annotated programs. To achieve this, we rely on the user-provided constraints on the types of a program: for each function f present in the program, that returns a constrained type, we generate a test. The tests consists of generating uniformly pseudo-random inputs and checking whether f 's output satisfies the constraint. We are able to automate this process by providing a set of generators for primitive types and generator combinators for composite types. To derive generators for constrained types, we present in this paper a technique that characterizes their inhabitants as the solution set of a numerical CSP. This is done by combining abstract interpretation and constraint solving techniques that allow us to efficiently and uniformly generate solutions of numerical CSP. We validated our approach by implementing it as a syntax extension for the OCaml language.

2012 ACM Subject Classification Software and its engineering → Dynamic analysis

Keywords and phrases Constraint Programming, Automated Random Testing, Abstract Domains, Constrained Types

Digital Object Identifier 10.4230/LIPIcs.CP.2021.59

Supplementary Material *Software (Source Code)*: <https://github.com/ghilesZ/Testify>
archived at `swh:1:dir:e80146ce697a659919067f8125461a6c5c9d553c`

Funding This work has been partially supported by the Defense Innovation Agency (AID) of the French Ministry of Defense under Grant No.: 2018.60.0072.00.470.75.01 and the RIN ALENOR project.

1 Introduction

In this article, we propose an automated testing framework that generates tests for a restricted class of dependent types, that is constrained types. Constrained types attach a membership predicate to a type and are used to restrict its set of possible values. For instance, to encode rationals as a pair of integer (n, d) , one could add the constraint $d \neq 0$ to filter invalid representations. Moreover, providing the constraint that n and d are coprime with $d > 0$ defines a canonical representation for this type. This is desirable when a given term has several structurally different but semantically equivalent representations, e.g. $\frac{2}{4}$ would violate the constraint while $\frac{1}{2}$ would be valid. However, type systems with constrained types are generally undecidable making it hard to obtain strong static guarantees at compile-time [2]. Dynamic verification, on the other hand, while not preserving these strong formal guarantees, makes the approach both feasible and practical. One instance of this is property-based testing [9] that can potentially detect bugs in programs given a specification. Still, this



© Ghiles Ziat, Matthieu Dien, and Vincent Botbol;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 59; pp. 59:1–59:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

requires to manually provide tests that may be tedious and error-prone. For these reasons, we focus on the development of an automated test system for constrained types. The goal of our framework is to exhibit wrong behaviours in such programs by finding instances of a constrained type that violate their predicates. We achieve this by automatically generating tests for functions that manipulate constrained types. This requires from the developer to input a constraint from which we extract a partial specification of the program. The resulting generated tests consist in generating random inputs for the tested functions and checking their output against the given specification. Therefore, the main challenge we face is the automatic derivation of uniform value generators for constrained types. We address this by combining two approaches: Constraint Programming [26] and Abstract Interpretation [11]. Abstract interpretation provides modular, efficient and precise abstractions, in particular, for numerical values. Coupling it to standard constraint programming resolution scheme allows us to provide fast and uniform input generators for our automatically generated tests.

Our implementation targets the OCaml [21] programming language, and the examples we show are written in it. However, our approach is generic and could be ported to other languages, hence, this paper voluntarily disregards some specificities of the OCaml language.

1.1 Example: Putting Constraints in Programming

Consider the following example:

```
1 type nat = int [@satisfying (fun x -> x >= 0)]
```

This type declaration `type nat = int` is an alias of the primitive for machine-integers. Adding the annotation `[@satisfying (fun x -> x >= 0)]` specifies the constraint that values x of this type must respect the constraint $x \geq 0$, that is the positive integers. We then generate a test for each function whose return type is `nat`, as in the following example:

```
1 let add (x : nat) (y : nat) : nat = x * y
```

Compiling this program with our preprocessor will generate the following test:

```
1 let add_test () =
2   let x_rnd = range 0 max_int in
3   let y_rnd = range 0 max_int in
4   (multiply x y) >= 0
```

To achieve this, we have solved the constraint $x \geq 0 \wedge x \leq 2^{64}$ and determined the set of its solution. This allowed us to define the generator for the `nat` type. Here, `range` is a primitive of our framework that draws uniformly within an integer range.

Running this test will (usually) yield an error message stating that the return value of `add` violates the predicate `(fun x -> x >= 0)` for some input. Indeed, when a random input close to 2^{64} is passed to the function, an overflow may occur which would cause the return value to violate the constraint.

This example is designed to illustrate our testing framework process. Throughout the paper, we will describe how to derive generators from more complex constraints.

1.2 Contributions

This article focuses on type-driven automated test generation. Our contribution is threefold:

- Firstly, we propose an automated testing framework capable to derive a partial specification of a program using a set of constraints given by the developer. We then run tests against this specification.
- Secondly, we present an abstract domain based solving technique able to characterize constrained types in a way that enables the automatic derivation of uniform random generators, i.e. each instance has the same probability to be drawn.

- Finally, we give abstract domains (i.e. boxes, polyhedra) and their associated operators that permit generic random uniform generation.

1.3 Outline

This paper is organized as follows: Section 2 presents the related works. Section 3 introduces the derivation of generators and specification for constrained types. Section 4 recalls definitions of our abstract domains usage in constraint solving, and explains the relations between Numerical Constraint Satisfaction Problems (NCSP) and constrained types. Section 5 discusses the problem of uniform random sampling within heterogeneous abstract values, in particular for polyhedra and, related, the cardinality estimation problem with such domains. Section 6 presents our implementation, its current limitations and gives some experimental results. Finally, Section 7 summarizes our work and discusses its future continuations.

2 Related Works

Random testing has been thoroughly studied, for testing correctness [16, 28], exhaustiveness [33], complexity [5] etc. Several frameworks exist and relate more or less to our work. For instance, *American Fuzzy Lop* [37] inspects the execution paths of the program and apply mutations to each input to potentially discover new ones. This method increases the coverage of the test but requires the binaries to be instrumented. In the *OCaml* ecosystem, the *ppx-inline-tests* preprocessor makes it possible to inline tests in the source code. Also, several testing libraries for *OCaml* programs exist, such as *monolith* [31], or *QCheck* [13], inspired from Haskell's *QuickCheck* library [9]. These property-based testing frameworks are widely used [24, 23]. In particular, we re-use from *QCheck* some basic value generators for our automatic test generation. However, these works require the developer to manually write the tests, that is the generators and the properties to be checked. Our approach makes it possible to define constrained types that will act as a partial specification of the program and automatically extract the needed generators. Our framework automatically generates tests and is, thus, less error-prone and is easily maintainable: a constrained type annotation is sufficient to generate tests for all functions that return values of this type. This way, the test suite is updated automatically each time a new function is added. Such as *ppx_inline_tests*, our framework is implemented as a preprocessing mechanism that has no side-effects on the execution of the original program.

Constrained types have been widely studied, for example, *Dependant ML* [36] enriches the type system of ML with a restricted form of dependent types. Also, *Cayenne* [2] has dependent types and is able to encode predicate logic at the type level allowing types to be used as specifications of programs. However, this makes *Cayenne*'s type checking undecidable. In [27], the authors presented a framework for a Hindley/Milner kind of type systems with constraints as a set of formulas over a cylindric algebra. These approaches to constrained types are either undecidable in the general case or do not support the same constraint language as we do, that is numerical constraints. Closer to our work, the study of automatic generator derivation for data types has already been studied in previous works. For example, in [6], the authors adapt a Boltzmann model for random generation of algebraic data types, in particular for inductive types. Our approach is similar but we handle constrained types. In [15], the author explores the definition of generators for a class of dependent types. However, the proposed techniques are not automatic and do not aim to be uniform.

There exists several uniform sampling tools for *SAT* [14, 8]. We aim for the same goal but for NCSP instead. In [17], the authors focus on generating random uniform solutions for CSP, however they target discrete problems, with very small domain size, which are

not adapted to numerical variables with large cardinalities (e.g. machine-integers). The use of CP techniques for test generation has already been explored in the CP literature. For instance, the *FocalTest* framework[7] uses a constraint-based approach to build inputs for property based testing. Another example is [18] in which the method proposed is used to generate white box tests. Our approach differs from these in several ways: our resolution scheme is based on abstract domain instead of *clp(FD)*. Also our constraints are extracted from the types while theirs are derived from the statements of the program. Finally we aim at producing uniform generators which is not the case in these works.

We also explore the problem of generators definition using numerical abstract domains. In [34], the author proposes a way to quantify the precision of abstract values through a measure of volumes, including an approximated measuring of polyhedra. Also, [19] proposes an algorithm to solve boolean and linear constraints, and to randomly select values among the set of solutions using rejection sampling. This is close from what we do in Section 5 except that we have additional hypothesis that allow us to compute an exact volume and minimize the use of rejection sampling.

3 Testing Semantics

Random testing within a typed language requires the definition of pseudo-random generators for types. These generators are used to provide inputs for the functions whose output is checked against a given specification. A pseudo-random generator g for a data type τ is a function $g : \mathcal{S} \rightarrow \tau$, with \mathcal{S} the type of random seeds, which is useful to make the tests reproducible. Previous work, such as [6], has shown that the derivation of efficient uniform random generators for algebraic data types can be made automatic, even in presence of recursive types. However, this is more difficult for constrained types: a constrained type can be seen as a pair $\langle \tau, p \rangle$, composed by an algebraic type τ and a predicate $p : \tau \rightarrow \text{bool}$. The set of its inhabitants is defined as $S = \{t \in \tau \mid p(t) = \text{true}\}$. As a result, a generator for a constrained type $\langle \tau, p \rangle$ needs to produce a value of type τ that satisfies p .

3.1 Type Language and Semantics

Our framework aims at generating tests that verify that some function does not violate the invariant attached to its return type. To do so, we provide to the developer the capability of constraining a type τ with an arbitrary predicate *i.e.* a function of type $\tau \rightarrow \text{bool}$. Therefore, our syntactic extension may be seen as a small but expressive annotation language. To be able to reason on the types of the program, we will suppose that all the values that we handle are explicitly typed. We consider a *ML*-like type language with constrained types. Its Bachus-Naur form (BNF) grammar is given in Figure 1.

A type declaration is composed of an identifier and a type expression. Type expressions can be: type identifiers, product types (tuples), sum types where each variant is differentiated by a unique constructor, and constrained type which we add to the language *via* the `[@satisfying]` annotation. Note that even if the syntax permits the definition of recursive types, these are not handled by our framework yet. The constraint language used for the definition of predicates over constrained types is relatively classic. Here, \mathbb{I} and \mathbb{F} are respectively the set of integers and floating point values, and \mathbb{V} is the set of variable identifiers.

Annotating a type τ with a predicate p defines a partial specification for the program: all functions returning a value of type τ are expected to produce values that satisfy p . This property being in the general case out of the reach of a type checker [2], we propose to test it.

<code>decl ::= 'type' ident '=' type</code>	declaration
<code>type ::= ident</code>	identifier
<code>type { '*' type }</code>	product
<code>ident 'of' type { ' ' ident 'of' type }</code>	sum
<code>type '@satisfying' constraint '['</code>	constrained
<code>constraint ::= arith \square arith</code>	$\square \in \{\geq, \leq, =, \neq\}$
<code>'not' constraint</code>	negation
<code>constraint ' ' constraint</code>	disjunction
<code>constraint '&&' constraint</code>	conjunction
<code>arith ::= i</code>	$i \in \mathbb{I}$
<code>f</code>	$f \in \mathbb{F}$
<code>v</code>	$v \in \mathbb{V}$
<code>arith \diamond arith</code>	$\diamond \in \{+, -, *, /, \%\}$
<code>- arith</code>	opposite

■ **Figure 1** Grammar of the constrained type language.

3.2 Constraints semantics

We introduce inference rules that enriches a standard type algebra with constrained types: because types are composable, we need to define inference rules to propagate constraints from atomic types to composite types. For example, when a type τ is product or a sum of types that were constrained by a property, this property is lifted to τ accordingly.

► **Example 1.** Consider the following constrained types which define the type of positive float and 2D circles:

```

1 type positive = float [@satisfying (fun x -> x >= 0)]
2 type circle = (float * float) * positive

```

Here, the type `circle` is not explicitly constrained, but as it depends on the type `positive`, an implicit constraint will be attached to it. More generally, whenever a type τ is defined using a constrained type τ' , the constraint over τ is also inherited by τ' . Here, `circle` is implicitly constrained by the function: `(fun ((cx,cy),radius) -> radius >= 0)`.

In Figure 2, we give a formal definition of the semantics for constraints composition. We define it by induction over the syntax while considering a predicate environment ρ that stores, for each type identifier, the predicate that was attached to it. For clarity, we consider that an unconstrained type is a constrained type who is attached a tautology. Also, we suppose the initial environment ρ_0 already equipped with tautological constraints for primitives types: $\rho_0 = [\text{unit} \mapsto \lambda().\text{true}, \text{bool} \mapsto \lambda b.\text{true}, \text{int} \mapsto \lambda i.\text{true}, \text{float} \mapsto \lambda f.\text{true}]$. For each rule, the conclusion gives the derivation formula of a constraint for a given type, using the constraints derived in the premises. The constraint for product types is the conjunction of constraints attached to the type sub-components. For sum types, we determine for each variant the corresponding constraint and build a predicate based on pattern matching¹ to select a constructor using case by case reasoning. For type declarations, we use the notation $\rho[id \mapsto p_\tau]$ to denote the setting of the constraint associated to the type identifier id to its new value p_τ .

¹ `function patterns` is equivalent to `fun x -> match x with patterns`

$$\begin{array}{l}
\text{(declaration)} \frac{\rho(\tau) \rightarrow p_\tau}{(\text{type } id = \tau, \rho) \rightarrow (\rho[id \mapsto p_\tau])} \\
\text{(constrained)} \frac{\rho(\tau) \rightarrow p_\tau}{(\tau[@satisfying p], \rho) \rightarrow \lambda x. p(x) \wedge p_\tau(x)} \\
\text{(identifier)} \frac{p = \rho(id)}{(id, \sigma) \rightarrow p} \\
\text{(product)} \frac{\rho(\tau_1) \rightarrow p_1 \quad \dots \quad \rho(\tau_n) \rightarrow p_n}{(\tau_1 * \dots * \tau_n, \rho) \rightarrow \lambda(x_1, \dots, x_n). p_1(x_1) \wedge \dots \wedge p_n(x_n)} \\
\text{(sum)} \frac{\rho(\tau_1) \rightarrow p_1 \quad \dots \quad \rho(\tau_n) \rightarrow p_n}{(c_1 \text{ of } \tau_1 \mid \dots \mid c_n \text{ of } \tau_n, \rho) \rightarrow \text{function } c_1(x_1) \mapsto p_1(x_1) \mid \dots \mid c_n(x_n) \mapsto p_n(x_n)}
\end{array}$$

■ **Figure 2** Constraint semantics.

3.3 Generator semantics

The derivation of random generators for composite types given random generators for atomic types, is made following the same inductive principle as in the previous section. To keep track of which generator is associated to which type, we consider an environment σ which associates to each type identifier its generator. We suppose an initial environment σ_0 populated with uniform random generators for primitive types. Figure 3 presents the derivation of uniform random generators for constrained types.

$$\begin{array}{l}
\text{(declaration)} \frac{\sigma(\tau) \rightarrow g_\tau}{(\text{type } id = \tau, \sigma) \rightarrow (\sigma[id \mapsto g_\tau])} \\
\text{(constrained)} \frac{}{(\tau[@satisfying p], \sigma) \rightarrow \mathbf{solve}(\tau, p)} \\
\text{(identifier)} \frac{g = \sigma(id)}{(id, \sigma) \rightarrow g} \\
\text{(product)} \frac{\sigma(\tau_1) \rightarrow g_1 \quad \dots \quad \sigma(\tau_n) \rightarrow g_n}{(\tau_1 * \dots * \tau_n, \sigma) \rightarrow \lambda i. (g_1(i), \dots, g_n(i))} \\
\text{(sum)} \frac{\sigma(\tau_1) \rightarrow \lambda i. c_1(g_1(i)) \quad \dots \quad \sigma(\tau_n) \rightarrow \lambda i. c_n(g_n(i))}{(c_1 \text{ of } \tau_1 \mid \dots \mid c_n \text{ of } \tau_n, \sigma) \rightarrow \mathbf{weighted}([(\mathbf{card}(\tau_1), g_n); \dots; (\mathbf{card}(\tau_n), g_n)])}
\end{array}$$

■ **Figure 3** Generator semantics.

For product types (similar to Cartesian product of sets), generators are obtained by composing the generators obtained for their components. An important point of our work is to derive random generators with uniform distribution: each inhabitants of the type has the same probability to be drawn. Because uniform distributions do not compose so easily, especially in the case of sum types (union of sets), we have to take care of the cardinal of each type's population. Hence for sum types, we decompose the uniform sampling of a value in two steps: first choosing a constructor, and then drawing a value for this constructor. For this to be uniform, the first step takes into account the cardinality of the components: the more inhabitant one has, the more likely it has to be chosen. To achieve that, we introduce two procedures: **card** and **weighted**.

The procedure **card** gives the number of inhabitants of a given type. As we restrict ourselves to non-recursive types, keeping track of the cardinality of an algebraic data type is straightforward: cardinality for sum (resp. product) types is given by the sum (resp. product) of the cardinalities of their components. Computing cardinalities of constrained types is equivalent in our case to counting the solutions of a CSPs, which is a known hard problem [30]. Hence, we use approximations to compute the cardinal of constrained types which we present in the next section. The procedure **weighted** chooses a generator among a list of generators. To choose uniformly, each generator has a probability of being chosen equal to its associated weight.

Finally generators for constrained types are given by the procedure **solve** that, given a type declaration and a predicate, builds the generator corresponding to the constrained type. We give a definition of this procedure in the next section.

3.4 Test Generation

From the generators and the constraints we derived from the annotated program, we may now produce tests. We retrieve value declarations in the program for which the return type is a constrained one. If the value declaration is a constant c , we simply generate a test that consists of applying the predicate to c . If the value declaration is a function f , we will first retrieve, for each of its arguments, the associated generators to build the inputs. Equipped with input generators, we then apply f to the uniformly drawn inputs and validate the outputs using the constraint predicate.

4 Solving Constrained Types

We now study the derivation of generators and cardinality estimation for constrained types. One way to automatically do this is to extensively compute the set of values of a type and keep only those that satisfy a constraint. We may then randomly choose among those whenever a generator is called. This approach however is not practical and does not scale. Another possibility is to use a rejection sampling technique using the generator for τ and checks its values against p . This should yield a uniform generator but present an important flaw: when the cardinality of the constrained type $\langle \tau, p \rangle$ is small compared to the cardinality of the original type τ , this tends to be ineffective. Moreover, cardinality may only be an estimation, not an exact result. Instead, our approach is to solve constrained types by providing a measurable characterization of their inhabitants. We are then able to define uniform random generators over it. To do so, we see a constrained type as a constraint satisfaction problem, and its inhabitants as the solutions of this problem. We use for this task a hybrid approach, that mixes both techniques from Abstract Interpretation [11] and Constraint Programming [26], based on *abstract domains*. These are a key notion in abstract interpretation as they implement an abstract semantic for which they provide data-structures and define algorithmic aspects. They are designed to abstract program values and are thus particularly well-suited for our needs. Moreover, many constructive and systematic methods to design and compose such domains exist in the literature: numerical domains (intervals, congruences, polyhedra, octagons, etc.), domain composition operators (products, powersets, etc.).

4.1 CSP extraction from a constrained type

A constraint-satisfaction problem can be defined as a triplet $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{V} = \{v_1, \dots, v_n\}$ is a set of variables, $\mathcal{D} = \{d_1, \dots, d_n\}$ a set of interval domains, each one being associated to a variable, and $\mathcal{C} = \{c_1, \dots, c_m\}$ is a set of constraints over the variables. A solution of a CSP $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ is an instance $i = \{v_1 \rightarrow x_1, \dots, v_n \rightarrow x_n\}$ that satisfies all of the constraints by substitution of the variables with their value in i , that is:

$$\forall i, x_i \in d_i \wedge \forall c \in \mathcal{C}, c(\{v_1 \rightarrow x_1, \dots, v_n \rightarrow x_n\})$$

In our case, we want to solve CSPs that are extracted from constrained types, to obtain an approximation of their sets of inhabitants. We can then tackle the problem of generating uniformly within this approximation. Consider the following declaration:

```
1 type itv = (int * int) [@satisfying (fun (inf,sup) -> inf <= sup)]
```

This type defines a bounded two-dimensional space, where dimensions correspond to the members of the tuple, named `inf` and `sup` in the predicate. These are constrained by the relation `inf ≤ sup`. From a constraint solving point of view, the set of inhabitant of this type is the set of solutions of the CSP: $\langle \mathcal{V} = \{\text{inf}; \text{sup}\}, \mathcal{D} = \{[-2^{64}; 2^{64}]; [-2^{64}; 2^{64}], \mathcal{C} = \{\text{inf} \leq \text{sup}\}\rangle$

We have automated the process of CSP extraction for type definition by *inlining* their declaration: we give an identifier to each of the value of the type and then work within a simple numerical abstract element where each identifier corresponds to a variable. Doing so builds a CSP that abstracts some information about the *shape* of the type and hence, in parallel, we build a function that will reconstruct from the solutions of the CSP, a value with the correct shape.

4.2 CSP solving

Solving a numerical CSP usually means finding one or all the solutions of the problem. Because this is generally impossible when the domains of the variables are continuous (or just very large), solvers generally compute a set of boxes, that is a Cartesian product of intervals, that *covers* the solution space. The resolution of such problems works from above: given an initial coarse approximation, several heuristics are used until a sufficiently good cover is found. In order to build this cover, a constraint solver generally alternates two main steps:

- *Filtering*: which reduces the variables domains by removing values that do not satisfy the constraints.
- *Splitting*: which duplicates the problem to create two (or more) complementary sub-problems that are smaller, w.r.t. a certain measure, than the original one.

Repeating these two steps in turn does not necessarily terminate. Hence, this procedure generally goes on until the search space contains: no solution, only solutions, or is smaller than a given parameter. Producing such a cover can be sufficient for us if we are able to compute its exact number of solutions, and select one of these uniformly. However, using boxes poorly fits our need, in particular, when we are considering relational constraints, which appear fairly commonly in constrained types. Therefore, we use the solving method of [29], which is designed to be parameterized by any abstract domain. The algorithm introduced in [29] builds a set of abstract elements \mathbb{S} that covers the solution space, i.e. for all instances i that satisfy all the constraints \mathcal{C} , we have $\exists e \in \mathbb{S}, i \in \gamma(e)$. Here γ is the usual concretization function for abstract values, that is the set of concrete instances abstracted by

an element. The algorithm starts from an initial abstract element e built from the domains of the variables. Then, e is filtered according to the set of constraints. If the filtered abstract element e' is not empty, three cases are possible:

- e' satisfies all the constraints, then it is added to the set of solutions.
- there is at least one constraint c that e' does not satisfy and its size is small enough with respect to a given threshold, it is also added to the set of solutions.
- otherwise, e' is divided into sub-elements using the split operator and the process is repeated with each of these sub-elements.

When all of the elements have been processed, the union of the element in \mathbb{S} is a sound over-approximation of the solution space. We propose a slightly modified version of this algorithm, which, we believe, is better suited for our needs.

4.3 Solving Algorithm

Contrarily to the algorithm of [29], we distinguish inner elements from outer elements. Inner elements are the ones that are guaranteed to only contain solutions while outer elements may contain non-solutions. Our algorithm is defined in Figure 1.

■ **Algorithm 1** Abstract solving for generator derivation.

```

1: function SOLVE( $\mathcal{D}, \mathcal{C}, \epsilon, max$ )
2:    $I \leftarrow \emptyset$ 
3:    $O \leftarrow \emptyset$ 
4:    $e = \text{init}(\mathcal{D})$ 
5:    $O \leftarrow \text{insert}(e, O)$ 
6:   while  $\mu(I, O) > \epsilon \vee |I| < max$  do
7:      $e \leftarrow \text{biggest}(O)$ 
8:      $e' \leftarrow \rho(e, \mathcal{C})$ 
9:     if  $e' \neq \perp$  then
10:      if  $\text{solution}(e', \mathcal{C})$  then
11:        push  $e'$  in  $I$ 
12:      else
13:        push split ( $e$ ) in  $O$ 
14: return  $I, O$ 

```

Our algorithm maintains two sets of elements: inner elements I , and outer elements O . Here, I under-approximates the solution set and O is such that $I \cup O$ over-approximates it. It first initializes an abstract element e and inserts it in the set of outer elements O . Then, the main loop proceeds repeating the steps: The biggest element of O is selected (which is more likely to contains more solutions), filtered using the propagator ρ , and pushed in I if it satisfies the constraints. Otherwise, it is split and the sub-elements are pushed back in O .

As we will see, the number of element in the cover is related to the size of the code generated for the random samplers. Also their rejection rate is closely related to the proportion of inner elements in the cover. Thus the tuning of the obtained generator may be controlled by the max and ϵ parameters: max is needed to avoid an exponential growth of the cover and ϵ fixes a rejection rate to reach. The next section gives insights about the code generation and defines precisely $\mu(I, O)$.

4.4 From covers to generators

Once we have computed a cover (I, O) for a constrained type $\langle \tau, p \rangle$, we have to compile it into a generator for $\langle \tau, p \rangle$. A cover is a set of inner elements and a set of outer elements. Thus, to compile a cover into a generator we start by compiling each element $e \in (I \cup O)$ into a generator. We then choose an element e of the cover, and generate an instance i using the generator associated with e . If e belongs to I , then i is kept, but if it belongs to O , then i must be checked against p to make sure that it is a valid member of the constrained type. In that case, we are forced to rely on rejection sampling.

Hence, the generator for the whole cover is actually a dispatcher to the generators of the elements: it randomly selects an element's generator with a probability proportional to the volume of the underlying set of solutions, then calls it. Note that this yields a uniform generator because the split operator produces elements that do not overlap. Otherwise, instances belonging to several elements would appear more often during the sampling.

To compute the cardinality of a constrained type $\langle \tau, p \rangle$ we must reason on the number of solutions of its associated cover. This exact number is given by:

$$\sum_{e \in I} |\gamma(e)| + \sum_{e \in O} |\{i \in \gamma(e) \mid p(i)\}|$$

However, as the number of solutions of an outer element e depends on the predicate p , it is hard to compute exactly in the general case. Instead, we use an over-approximation of this number that is $|\{i \in \gamma(e)\}|$. Our cardinality estimation for a cover (I, O) , denoted $\text{card}(I, O)$, is given by:

$$\text{card}(I, O) = \sum_{e \in I} |\gamma(e)| + \sum_{e \in O} |\gamma(e)|$$

Despite of this over-approximation, our sampling method is uniform, thanks to rejection sampling: rejecting erroneous solutions does not bias the uniform distribution. To ensure the sampling of the uniform distribution in case of rejection, the whole sampling process is restarted from the beginning *i.e.* an abstract element is drawn w.r.t. to its volume and so on.

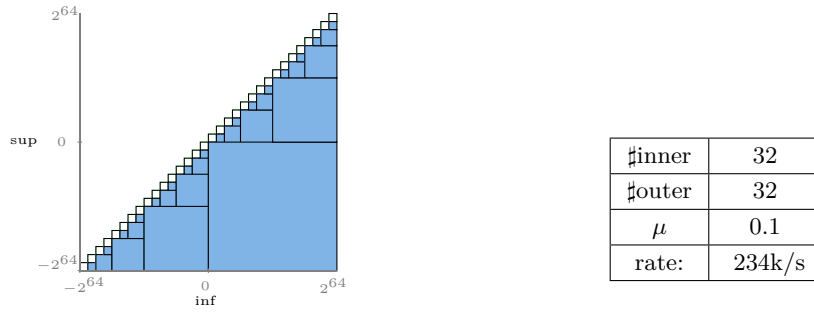
One way to mitigate the occurrences of rejections, is to minimize the volume of O . Hence the introduction of $\mu(I, O)$ defined as

$$\mu = 1 - \frac{\text{card}(I, \emptyset)}{\text{card}(I, O)}$$

It measures the relative error between the over-approximation and the exact volume of an abstract element. At the start of the algorithm process $\mu(I, O) = 1$. Then it decreases at each iteration until reaching the desired precision. Note that $\mu = 0$ means that the abstract element corresponds exactly to the set of solutions of the CSP and the random sampling will be made without rejection.

5 Abstract Domains for Random Testing

The technique presented in the previous section relies on abstract domains to solve and derive generators for constrained types. To use abstract domains in the context of constraint resolution, the authors of [29] define several operators and requirements they must satisfy. Our use-case requires the same operators, plus an additional one for the random sampling of an element. The following definition gives these.



(a) Graphical resolution of the CSP.

(b) Metrics over the resulting generator.

■ **Figure 4** Solving the `itv` constrained type with boxes.

► **Definition 2.** *Abstract domains for constraint solving are given by*

- a partial order $\langle D, \sqsubseteq \rangle$ and the usual abstract set operators and values $\langle \top, \perp, \sqcap, \sqcup \rangle$
- an abstraction α and a concretization function γ
- a size function **card**: $D \rightarrow \mathbb{N}^+$
- a split function **split**: $D \rightarrow P(D)$,
- a constraint filtering operator $\rho : D \times \mathcal{C} \rightarrow D \cup \{\perp\}$, which given an abstract value e and a constraint c computes the smallest abstract value (possibly empty) entailed by c and e .
- a generation function **uniform**: $D \rightarrow \mathbb{R}^n$

For sake of concision, we will not detail the compilation of abstract elements into expressions. Instead we define random generation functions on abstract elements.

5.1 Boxes

A very studied abstract domain in continuous constraint solving is the abstract domain of boxes. Its main operators rely on interval arithmetic [1] and were already introduced in previous work [35]. In our case, the cardinality of a box $b = [a_1, b_1] \times \dots \times [a_n, b_n]$ is its volume (defined as its Lebesgue measure) i.e. $\text{card}(b) = \prod_{i=1}^n (b_i - a_i)$. For the split operation, we use standard bisection of a variable with the so-called *largest-first* heuristic which chooses the variable with the biggest range as a variable selection strategy. Also, for the filtering operation we use the HC4 constraint propagation algorithm [3]. Finally, the uniform distribution over a n -dimensional box is sampled using n uniform (over $[0, 1]$) and independant random variables $(r_i)_{i \leq n}$ with the formula $(a_i + r_i * (b_i - a_i))_{i \leq n}$.

Defining these operators is sufficient to embed this domain within our testing framework. Using boxes, it is possible to derive efficient generators for constrained type that use non relational constraints, that are constraints that involve a single variable. However, producing fast generators in presence of relational constraints is harder *e.g.* the `itv` type of Example 4.1. Figure 4 shows graphically the approximation we obtain and the corresponding generated OCaml code is given in Appendix A.1. Filled elements correspond to inner elements, and empty elements correspond to outer elements. The Table 4b gives some metrics over the cover and the resulting generator: the row #inner (resp. #outer) gives the number of inner (resp. outer) elements, the third row gives the μ -score and the last line gives the generation rate of the obtained generator, which is measured experimentally and is given in number of calls per seconds.

We can see that when dealing with an affine constraint, the use of boxes misfits our needs: the precision needed to avoid rejections during the sampling leads to a high number of elements in the cover of the solution space and so a very large (in term of code size) and slow sampler. To solve this issue, we focus in the next subsection on a relational abstract domain.

5.2 Polyhedra

The polyhedra abstract domain [12] is a numerical relational abstract domain that approximates sets of points as convex closed polyhedra. Modern implementations [20] generally follow the “double description approach” and maintain two dual representations for each polyhedron: a set of linear constraints and a set of vertices. The constraint representation of the polyhedron is the intersection of the half-spaces defined by the linear constraints. The vertex representation is the convex hull of a set of points. These dual descriptions are very useful in practice as polyhedra operators [20] are generally easier to define on one representation rather than the other. We use both to define the filtering, splitting, measure and random sampling functions. Constraint filtering for polyhedra generally consists in building a sound linear approximation of a constraint (different approximations can be used e.g. quasi-linearization [25] or linearization for polynomial constraints [22]), and then adding it to the representation of a given polyhedron.

Volume computation and uniform random sampling within a polyhedron are notoriously hard tasks. The first problem is \sharp P-hard (see [4] for example). For the sampling, the fastest algorithm (to our knowledge) has an expected (time) complexity in $\mathcal{O}^*(n^3)^2$ (see Theorem 3.1.3 of [10]) whose result validity and running time are probabilistic.

We mitigate these problems by systematically considering a simpler case: simplices. A simplex is the most simple polyhedron with a non null volume, obtainable in an n -dimensional space as it is the convex hull of $n + 1$ vertices. Instead of stopping the split and filtering procedure when an inner polyhedron is found, we continue to split elements until all are simplices. To do so, we use a split operator that favors the creation of simplices that can be summarized as follows. Suppose a polyhedron P lives in a n -dimensional space and is not already a simplex, i.e. it is defined using at least $n + 2$ vertices:

- pick $n + 1$ arbitrary vertices, $\{v_0, \dots, v_n\}$
- compute Q the smallest polyhedron that encompasses $\{v_0, \dots, v_n\}$
- return $Q \cup (P \ominus Q)$

Here, the difference operator (\ominus) used in the last step corresponds to the set difference of two polyhedra. This operator, illustrated by Figure 5, uses the constraint representation: the polyhedron Q is a space defined by the conjunction of a set of constraints C_Q , where each constraint is a linear inequality over the variables of the polyhedron. It is defined as:

$$P \ominus Q \triangleq \left\{ \bigcup P \cap \neg c \mid c \in C_Q \right\}$$

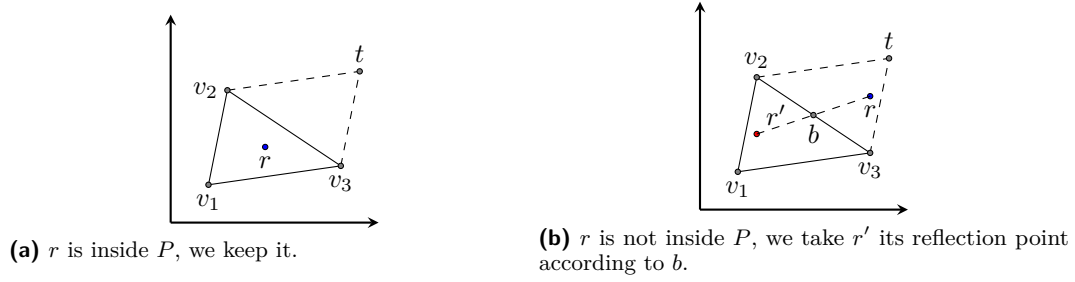
Our simplex split allows us to decompose a polyhedron while performing the resolution, which allows us to define both the volume estimation and the uniform random sampling on simplices. The way to compute the volume of a simplex is well known and not discussed here.

In order to sample a point of a simplex, we first consider its mirror parallelotope: given a simplex polyhedron P defined by the set of vertices $V = \{v_1, \dots, v_n\}$, we build its mirror parallelotope P' by adding one vertex t to V , obtained by translation of v_1 by

² The \star in $\mathcal{O}^*(n^3)$ hides logarithmic factor.



■ **Figure 5** Difference operator: $P \ominus Q$.

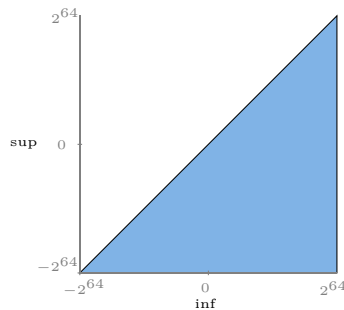


■ **Figure 6** Simplex uniform generation procedure.

the vector $\delta = \overrightarrow{v_1 v_2} + \overrightarrow{v_1 v_3} + \dots + \overrightarrow{v_1 v_n}$. Then, a point r of P' is uniformly drawn by sampling $n - 1$ independent and uniform random variables $(r_i)_{i \leq n-1}$ over $[0, 1]$, such that $r = r_1 \cdot \overrightarrow{v_1 v_2} + r_2 \cdot \overrightarrow{v_1 v_3} + \dots + r_{n-1} \cdot \overrightarrow{v_1 v_n}$.

- If r is inside P we keep it.
- Else, r is outside P , we keep r' its symmetrical according to b .

Where b is the barycenter of the opposite face to v_1 , i.e. the $n - 2$ dimensional face of corners $\{v_2, \dots, v_n\}$. Figure 6 illustrates this procedure for a 2D-simplex. Note that every point in P has exactly two ways of being chosen, that is directly or by symmetry, which makes this procedure uniform. Equipped with this relational abstract domain, we can compare the result obtained for the CSP of the `itv` type, illustrated by Figure 7. The corresponding code, given in Appendix A.2, is approximately three times faster than the one obtained with boxes, thanks to the null rejection rate ($\mu = 0$).



# inner	1
# outer	0
μ	0
rate:	627k/s

■ **Figure 7** Approximation of the `itv` type with polyhedra.

6 Current Implementation and Benchmarks

Our implementation is built as a syntax extension for *OCaml* based on a preprocessing mechanism. We use *OCaml*'s attributes to allow the user to annotate its types with constraints. Attributes are placeholders in the syntax tree which are ignored by the compiler but can be used by external tools such as ours. We have developed a prototype to demonstrate the interest of our technique. It is open-source and available at the url <https://github.com/ghilesZ/Testify>. It currently implements the work we have presented in Sections 3 and 4 plus some other features we describe briefly here. Our main focus is the constraint solving of constrained types to automatically derive generators. However for a more practical use, we provide the programmer two other ways of specifying a generator for a given type. The first one is using the **rejection** keyword, for example `type even = int [@@satisfying rejection(fun x -> x mod 2 = 0)]`. Constraints tagged as rejected will not be handled by the constraint solver but will simply be treated *a posteriori* after the generation to keep or discard generated values. The second one is by manual annotation of functions: the developer can tag one of its own function of type $\mathcal{S} \rightarrow \tau$ as a custom generator for the type τ . This will overload the generator automatically derived by our framework. Also we have presented a type language with tuples and sum types but our actual implementation also handles polymorphic and record types. We have not detailed those here as they do not represent a specific challenge from a constraint solving perspective.

Also, as the numerical values we manipulate in our CSPs correspond to sets of machine integers and floating point numbers, the computation we perform are likely to produce round-off errors if made using standard precision. To bypass this issue, all of our computations are made using arbitrary precision integers and rationals. Moreover, our uniformity metrics are valid in \mathbb{R} but not necessarily for floating point numbers. We believe that this is a reasonable approximation. Finally our current implementation suffers from some limitations, for example: we handle explicitly typed values only and do not enjoy the capabilities of *OCaml*'s type inference mechanism. Also, we do not have a generator derivation mechanism for recursive types and further work will be needed to lift these restrictions.

We have applied our testing framework on some open source *OCaml* libraries where we have identified and annotated some constrained types. These types use quite simple constraints that are mainly bound constraints, linear constraints of the form $x_i \leq x_j$, and some disjunctions of such constraints. This reflects the fact that when writing code, the developer keeps in mind a relatively simple representation of the set of possible values of its type. Table 1 presents some metrics over the tests we have generated using different configurations. The first two columns give some information about the constrained type from which we derive the CSP. The first column specifies the kind of constraints attached to the type: *bc* for bound constraints, *lin* for linear constraints, and *dis*, for disjunctions of linear constraints. The second column indicates the number of variables appearing in the corresponding CSP. The next columns give some quality metrics over the generators: the generation speed and uniformity. The column \mathcal{B}_8 (resp. \mathcal{B}_{64}) gives the measures for the boxes abstract domain with a cover size limited to 8, (resp. 64), the column \mathcal{P} gives these values for polyhedra (with a cover size limited to 64). For comparison purposes, we add a supplementary column \mathcal{RS} that gives the statistics we obtain using rejection sampling. The row μ shows the value of $\mu(I, O)$ at the end of Algorithm 1 and the last row gives the generation rate, i.e. the number of generated values (in thousands) per seconds. The results of Table 1 validates our intuition: constraint solving of constrained types helps producing efficient generators. All of our abstract domain based configurations outperform the rejection

■ **Table 1** Generation rate and value of μ per configuration.

CSP		\mathcal{B}_8		\mathcal{B}_{64}		\mathcal{P}		\mathcal{RS}	
kind	#var	rate	μ	rate	μ	rate	μ	rate	μ
<i>bc</i>	2	10075	0	10059	0	2604	0	816	0
<i>bc</i>	2	9860	0	9996	0	2586	0	842	0
<i>lin</i>	2	2551	0.6	3312	0.35	3262	0	1333	0
<i>bc</i>	1	26055	0	26146	0	7870	0	1922	0
<i>lin</i>	2	2602	0.6	3200	0.35	3358	0	1337	0
<i>lin</i>	2	2594	0.6	3166	0.35	3366	0	1080	0
<i>lin</i>	2	2622	0.6	3312	0.35	3234	0	428	0
<i>bc</i>	2	10246	0	10436	0	2768	0	1449	0
<i>lin</i>	2	2066	0.82	2812	0.44	6394	0	115	0
<i>lin</i>	3	615	1	716	1	1362	0	403	0
<i>lin</i>	2	2213	0.6	2750	0.35	6146	0	2278	0
<i>dis</i>	5	462	1	523	0.85	1189	0	452	0

sampling approach by one order of magnitude. In columns \mathcal{B}_8 and \mathcal{B}_{64} the size limit for the cover varies and as expected, it decreases μ . Moreover, on the benchmarks it almost always increases the generation rate. This indicates that the execution time benefits more from reducing the rejection rate than from increasing the size of the cover. However, it is slower on most examples using bound constraints than the boxes due to the relative complexity of the simplex generation procedure compared to the one of boxes. For the examples with linear relational constraints, the samplers produced with polyhedrons are faster.

7 Conclusion

We have proposed in this paper an automated type-driven testing framework for programs that manipulate constrained types. We have defined an automated technique to derive efficient random uniform generators for numerical constrained types. To do so, we have proposed several abstract domains, both relational and non-relational, with the addition of random value generator. The strength of our method lies in its genericity: it allows us to shift the problem of uniform random sampling of CSP solution to the definition of a uniform generation operator for abstract domains. Further works include the study of such operators for some popular domains of abstract interpretation such as congruences, products, octagons, etc. Another idea would be to couple our methods with Boltzmann generation techniques (like in [6]) in order to build generators for recursive types with constraints.

Our techniques were implemented in a prototype that can be used as a preprocessor for *OCaml* programs. Even though we targeted *OCaml* and although we have taken advantage of its generic syntax extension mechanism, the process we have presented could be adapted to most programming languages. The tests we are able to generate are not made to be as pertinent as some hand written tests and our goal is not to replace those. However our approach being fully automatic and fast, it can be used *on the fly*, while programming, to find bugs quickly. We believe that random uniform generators for abstract domain open the way for hybrid techniques at the border between sound static analysis approaches and complete testing techniques. For example, one could imagine a backward analysis able to derive necessary preconditions (as in [32]) that exhibit bugs in a program and then uniformly generating tests within the corresponding abstract element to find actual bugs.

References

- 1 Ignacio Araya, Gilles Trombettoni, and Bertrand Neveu. Exploiting Monotonicity in Interval Constraint Propagation. In Association for the Advancement of Artificial Intelligence, editor, *Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 9–14, Atlanta, United States, July 2010. <http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1699>. URL: <https://hal-enpc.archives-ouvertes.fr/hal-00654400>.
- 2 Lennart Augustsson. Cayenne – a language with dependent types. In S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques, editors, *Advanced Functional Programming*, pages 240–267, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
- 3 Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-Francois Puget. Revising hull and box consistency. In *Logic Programming: Proceedings of the 1999 International Conference on Logic Programming*, pages 230–244, January 1999.
- 4 G. Brightwell and P. Winkler. Counting linear extensions is #p-complete. In *STOC '91*, 1991.
- 5 Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering, ICSE '09*, page 463–473, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1109/ICSE.2009.5070545.
- 6 Benjamin Canou and Alexis Darrasse. Fast and sound random generation for automated testing and benchmarking in objective caml. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML, ML '09*, page 61–70, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1596627.1596637.
- 7 Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. Focaltest: A constraint programming approach for property-based testing. In José Cordeiro, Maria Virvou, and Boris Shishkov, editors, *Software and Data Technologies*, pages 140–155, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- 8 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable and nearly uniform generator of sat witnesses. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 608–623, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- 9 Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP '00*, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/351240.351266.
- 10 Benjamin Cousins. *Efficient high-dimensional sampling and integration*. PhD thesis, Georgia Institute of Technology, 2017.
- 11 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.
- 12 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978.
- 13 Simon Cruanes. *QuickCheck inspired property-based testing for OCaml*. URL: <https://github.com/c-cube/qcheck>.
- 14 Rafael Dutra, Kevin Laeuffer, Jonathan Bachrach, and Koushik Sen. Efficient sampling of sat solutions for testing. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), ICSE '18*, page 549–559, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3180155.3180248.
- 15 Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Random generators for dependent types. In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004*, pages 341–355, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- 16 Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, 2005. doi:10.1145/1064978.1065036.

- 17 Vibhav Gogate and Rina Dechter. A new algorithm for sampling csp solutions uniformly at random. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006*, pages 711–715, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- 18 A. Gotlieb, B. Botella, and M. Watel. Inka: Ten years after the first ideas. In *19th International Conference on Software & Systems Engineering and their Applications (ICSSEA'06)*, Paris, France, December 2006.
- 19 Erwan Jahier and Pascal Raymond. Generating random values using Binary Decision Diagrams and Convex Polyhedra. In Frédéric Benhamou, Narendra Jussien, and Barry O'Sullivan, editors, *Trends in Constraint Programming*, page 416 pp. ISTE, 2007. URL: <https://hal.archives-ouvertes.fr/hal-00389766>.
- 20 Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Proceedings of the 21th International Conference Computer Aided Verification (CAV 2009)*, 2009.
- 21 Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique*, 54, 2014.
- 22 Alexandre Maréchal, Alexis Fouilhé, Tim King, David Monniaux, and Michaël Périn. Polyhedral approximation of multivariate polynomials using handelman's theorem. In *17th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 166–184, 2016.
- 23 Jan Midtgaard. Quickchecking patricia trees. In Meng Wang and Scott Owens, editors, *Trends in Functional Programming*, pages 59–78, Cham, 2018. Springer International Publishing.
- 24 Jan Midtgaard, Mathias Justesen, Patrick Kasting, Flemming Nielson, and Hanne Nielson. Effect-driven quickchecking of compilers. *Proceedings of the ACM on Programming Languages*, 1:1–23, August 2017. doi:10.1145/3110259.
- 25 Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *7th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2006.
- 26 Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.
- 27 Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPoS*, 5:35–55, January 1999. doi:10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAP04>3.0.CO;2-4.
- 28 Michal Palka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. *Proceedings - International Conference on Software Engineering*, January 2011. doi:10.1145/1982595.1982615.
- 29 Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A constraint solver based on abstract domains. In *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2013)*, 2013.
- 30 G. Pesant. Counting solutions of cps: A structural approach. In *IJCAI*, 2005.
- 31 F. Pottier. Strong automated testing of ocaml libraries. In *JFLA 2021 - 32es Journées Francophones des Langages Applicatifs*, 2020.
- 32 Xavier Rival. Understanding the origin of alarms in astrée. In Chris Hankin and Igor Siveroni, editors, *Static Analysis*, pages 303–319, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- 33 Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck automatic exhaustive testing for small values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, volume 44, pages 37–48, January 2008.
- 34 Pascal Sotin. Quantifying the precision of numerical abstract domains, 2010. URL: <https://hal.inria.fr/inria-00457324>.
- 35 Charlotte Truchet, Marie Pelleau, and Frédéric Benhamou. Abstract domains for constraint programming, with the example of octagons. *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 72–79, 2010. doi:10.1109/SYNASC.2010.69.

- 36 H. Xi. Dependent ml an approach to practical programming with dependent types. *J. Funct. Program.*, 17:215–286, 2007.
- 37 M Zalewski. *American fuzzy lop*. URL: <https://lcamtuf.coredump.cx/afl/>.

A Generated code for generators

A.1 Generated code using the boxes abstract domain

The following code gives the generator for the `itv` type we derived using the boxes abstract domain. The `weighted` procedure chooses a generator w.r.t. their probability. To perform quickly on average the list is sorted in decreasing order of probabilities, first elements being bigger, and thus more likely to be chosen. The function is more thus likely to stop quickly in average without harming uniformity.

```

1 weighted
2 [(0.4000000000000001,
3   ((fun x ->
4     (fun i -> (((get_int "x") i), ((get_int "y") i)))
5     ((fun rs ->
6       [("y", ((mk_int_range 0 0x3fffffffffffffff) rs));
7        ("x", ((mk_int_range 0x4000000000000000 (-1)) rs))]) x)))));
8 (0.1000000000000001,
9   (reject (fun (x, y) -> x <= y)
10    (fun x ->
11      (fun i -> (((get_int "x") i), ((get_int "y") i)))
12      ((fun rs ->
13        [("y",
14          ((mk_int_range 0x2000000000000000 0x3fffffffffffffff) rs));
15         ("x", ((mk_int_range 0 0x1fffffffffffffff) rs))] x)))));
16 (0.1000000000000001,
17   (reject (fun (x, y) -> x <= y)
18    (fun x ->
19      (fun i -> (((get_int "x") i), ((get_int "y") i)))
20      ((fun rs ->
21        [("y", ((mk_int_range 0 0x1fffffffffffffff) rs));
22         ("x", ((mk_int_range 0 0x1fffffffffffffff) rs))] x)))));
23 (0.1000000000000001,
24   (reject (fun (x, y) -> x <= y)
25    (fun x ->
26      (fun i -> (((get_int "x") i), ((get_int "y") i)))
27      ((fun rs ->
28        [("y", ((mk_int_range 0x6000000000000000 (-1)) rs));
29         ("x",
30          ((mk_int_range 0x4000000000000000 0x5fffffffffffffff) rs))]
31        x)))));
32 (0.1000000000000001,
33   (reject (fun (x, y) -> x <= y)
34    (fun x ->
35      (fun i -> (((get_int "x") i), ((get_int "y") i)))
36      ((fun rs ->
37        [("y",
38          ((mk_int_range 0x4000000000000000 0x5fffffffffffffff) rs));
39         ("x",
40          ((mk_int_range 0x4000000000000000 0x5fffffffffffffff) rs))]
41        x)))));
42 (0.0500000000000001,
43   (reject (fun (x, y) -> x <= y)
44    (fun x ->
45      (fun i -> (((get_int "x") i), ((get_int "y") i)))
46      ((fun rs ->
47        [("y",
48          ((mk_int_range 0x2000000000000000 0x3fffffffffffffff) rs));
49         ("x",
50          ((mk_int_range 0x3000000000000000 0x3fffffffffffffff) rs))]
51        x)))));
52 (0.0500000000000001,
53   (reject (fun (x, y) -> x <= y)
54    (fun x ->
55      (fun i -> (((get_int "x") i), ((get_int "y") i)))
56      ((fun rs ->
57        [("y",
58          ((mk_int_range 0x2000000000000000 0x3fffffffffffffff) rs));

```

```

59         ("x",
60         ((mk_int_range 0x2000000000000000 0x2fffffffffffffff) rs)))
61         x)))));
62 (0.0500000000000001,
63 (reject (fun (x, y) -> x <= y)
64 (fun x ->
65 (fun i -> ((get_int "x") i), ((get_int "y") i)))
66 ((fun rs ->
67 [("y", ((mk_int_range 0x6000000000000000 (-1)) rs));
68 ("x", ((mk_int_range 0x7000000000000000 (-1)) rs))] x)))));
69 (0.0500000000000001,
70 (reject (fun (x, y) -> x <= y)
71 (fun x ->
72 (fun i -> ((get_int "x") i), ((get_int "y") i)))
73 ((fun rs ->
74 [("y", ((mk_int_range 0x6000000000000000 (-1)) rs));
75 ("x",
76 ((mk_int_range 0x6000000000000000 0x6fffffffffffffff) rs))]
77 x))))))
78

```

A.2 Generated code using the polyhedra abstract domain

The following code was generated by our framework as a generator for the **itv** type using the polyhedra abstract domain. The **simplex** procedure corresponds to drawing method given in section 5.2.

```

1 fun x ->
2 (fun i -> ((get_int "x") i), ((get_int "y") i)))
3 ((simplex
4  [((mk_int 0x4000000000000000), "y");
5   ((mk_int 0x4000000000000000), "x")]
6   [((mk_int 0x3ffffffffffffe00), "y");
7    ((mk_int 0x4000000000000000), "x")];
8   [((mk_int 0x3ffffffffffffe00), "y");
9    ((mk_int 0x3ffffffffffffe00), "x")]]
10  [((mk_int 0x3ffffffffffffe00), "y"); ((mk_int 0), "x")]) x)

```


The Effect of Asynchronous Execution and Message Latency on Max-Sum

Roie Zivan ✉ 


Ben Gurion University of the Negev, Beer Sheva, Israel

Omer Perry ✉ 

Ben Gurion University of the Negev, Beer Sheva, Israel

Ben Rachmut ✉ 

Ben Gurion University of the Negev, Beer Sheva, Israel

William Yeoh ✉ 

Washington University in Saint Louis, MO, USA

Abstract

Max-sum is a version of belief propagation that was adapted for solving distributed constraint optimization problems (DCOPs). It has been studied theoretically and empirically, extended to versions that improve solution quality and converge rapidly, and is applicable to multiple distributed applications. The algorithm was presented both as a synchronous and an asynchronous algorithm, however, neither the differences in the performance of these two execution versions nor the implications of message latency on the two versions have been investigated to the best of our knowledge.

We contribute to the body of knowledge on Max-sum by: (1) Establishing the theoretical differences between the two execution versions of the algorithm, focusing on the construction of beliefs; (2) Empirically evaluating the differences between the solutions generated by the two versions of the algorithm, with and without message latency; and (3) Establishing both theoretically and empirically the positive effect of damping on reducing the differences between the two versions. Our results indicate that in contrast to recent published results indicating the drastic effect that message latency has on distributed local search, damped Max-sum is robust to message latency.

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

Keywords and phrases Distributed constraints, Distributed problem solving

Digital Object Identifier 10.4230/LIPIcs.CP.2021.60

Funding This research is partially supported by US-Israel Binational Science Foundation (BSF) grant #2018081 and US National Science Foundation (NSF) grant #1838364.

1 Introduction

Recent advances in computation and communication have resulted in realistic distributed applications, in which humans and technology interact and aim to optimize mutual goals (e.g., IoT applications). A promising multi-agent approach to solve these types of problems is to model them as *distributed constraint optimization problems* (DCOPs), where decision makers are modeled as *agents* that assign *values* to their *variables*. The goal in a DCOP is to optimize a global objective in a decentralized manner. Unfortunately, the communication assumptions of the DCOP model are overly simplistic and often unrealistic: (1) All messages arrive *instantaneously* or have *very small and bounded delays*; and (2) Messages sent *arrive in the order that they were sent*. These assumptions do not reflect real-world characteristics, where messages may be disproportionally delayed due to different bandwidths in different communication channels.



© Roie Zivan, Omer Perry, Ben Rachmut, and William Yeoh;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles and Practice of Constraint Programming (CP 2021).

Editor: Laurent D. Michel; Article No. 60; pp. 60:1–60:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Recently, a study that investigated the effect of message latency on standard distributed local search algorithms, e.g., MGM and DSA, has shown that message delays have a *dramatic positive effect* on the performance of the asynchronous versions of these algorithms [18]. Apparently, message latency generates an exploration effect, which improves significantly the quality of the solutions they produce. Nevertheless, this study did not investigate the effect on distributed incomplete inference algorithms, e.g., the Max-sum algorithm, although, these algorithms have been shown recently to be most successful [3, 4]. Thus, we focus our attention to the effect of message latency on Max-sum and its variants in this paper.

Max-sum is a version of the belief propagation algorithm [16, 25], which is used for solving DCOPs. It has been recently proposed for solving multi-agent optimization problems in applications, such as sensor systems [23, 22], task allocation for rescue teams in disaster areas [19], and smart homes [21]. As with most belief propagation algorithms, Max-sum is known to converge to an optimal solution when solving problems represented by acyclic graphs. On problems represented by cyclic graphs, the beliefs may fail to converge, and the resulting assignments that are considered optimal under those beliefs may be of low quality [6, 30]. This occurs because cyclic information propagation leads to computation of inaccurate and inconsistent information [16].

To decrease the effect of cyclic information propagation in belief propagation, the *damping* method has been suggested. It balances the weight of the new calculation performed in each iteration and the weight of calculations performed in previous iterations, resulting in an increased probability for convergence [4]. Recently, splitting nodes in the factor graph on which belief propagation operates has been shown to be an effective method for accelerating the convergence of the algorithm when combined with damping [20, 4].

Max-sum has been presented both as an asynchronous and as a synchronous algorithm (e.g., [6, 30, 5]). In the synchronous version, agents perform in iterations. In each iteration, an agent sends messages to all its neighbors and waits for the messages sent to it from all its neighbors to arrive, before moving to the next iteration. In the asynchronous version, agents react to messages when they arrive. To best of our knowledge, the implications of this difference in the execution of the algorithm on its performance have not been studied to date. Moreover, while message latency does not affect the actions that agents perform (only delays them) in the synchronous version, intuitively, it is expected to have a major effect on the performance of the asynchronous version. The reason is that the beliefs included in messages are used by agents in the construction of beliefs that they propagate to others and in their assignment selection. In asynchronous execution, belief construction and assignment selection might be performed while considering imbalanced and inconsistent information.

In this paper, we make the following contributions:

1. We analyze the properties of the two execution versions of Max-sum, synchronous and asynchronous. More specifically, using backtrack cost trees [28], we investigate the possible differences between the propagated beliefs in synchronous and asynchronous executions of Max-sum.
2. We investigate the effect of damping on asynchronous Max-sum. While there are clear indications (both empirical and theoretical) that damping improves the performance of the synchronous version of Max-sum [4, 28], to best of our knowledge, the effect of damping on the asynchronous version of Max-sum has not been studied. We analyze this effect both theoretically and empirically. Both indicate that damping reduces the differences between synchronous and asynchronous execution.
3. We investigate the performance of the different versions of the algorithm in the presence of message latency. While the beliefs propagated and the computation that agents perform are not affected by message latency in the synchronous version (only delayed), this is not

true for the asynchronous version. Once again, our empirical results reveal that damping reduces the differences. Moreover, the version of Max-sum proposed by [4] that includes both damping and splitting maintains its fast convergence properties and the quality of solutions, even in asynchronous execution with message delays.

2 Background

In this section we provide background on *graphical models*, *distributed constraint optimization problems* (DCOPs), the DCOP versions of belief propagation – *Max-sum* and its variants – and *backtrack cost tree* (BCT) – the tool we use to analyze the algorithms’ behavior. While the Max-sum variants that we discuss are actually solving a min-sum problem [20], we will still refer to them as “Max-sum” since this name is commonly used [6, 7, 30].

2.1 Graphical Models

Graphical models such as Bayesian networks or constraint networks are a widely used representation framework for reasoning and solving optimization problems. The graph structure is used to capture dependencies between variables [11]. Our work extends the theory established in [24], which considered the most a priori Maximum a posteriori (MAP) assignment, which is solved using the Max-product version of belief propagation. The relation between MAP and constraint optimization is well established [11, 6, 15], and thus, results that consider Max-product for MAP apply to Max/Min-sum for solving constraint optimization problems, as well as the other way round [20]. Without loss of generality, we will focus on constraint optimization, since it is more common in AI literature. Moreover, we will consider the distributed version of the problem, since it is a natural representation for message passing algorithms. Nevertheless, our results apply to any version of problem represented by a graphical model and solved by belief propagation, as do the results of [24].

2.2 Distributed Constraint Optimization Problems

Without loss of generality, in the rest of this paper, we will assume that all problems are minimization problems, as it is common in the DCOP literature (e.g., [13]). Thus, we assume that all constraints define costs and not utilities.

A DCOP is defined by a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{R} \rangle$. \mathcal{A} is a finite set of agents $\{A_1, A_2, \dots, A_n\}$. \mathcal{X} is a finite set of variables $\{X_1, X_2, \dots, X_m\}$. Each variable is held by a single agent, and an agent may hold more than one variable. \mathcal{D} is a set of domains $\{D_1, D_2, \dots, D_m\}$. Each domain D_i contains the finite set of values that can be assigned to variable X_i . We denote an assignment of value $x \in D_i$ to X_i by an ordered pair $\langle X_i, x \rangle$. \mathcal{R} is a set of relations (constraints). Each constraint $R_j \in \mathcal{R}$ defines a non-negative *cost* for every possible value combination of a set of variables, and is of the form $R_j : D_{j_1} \times D_{j_2} \times \dots \times D_{j_k} \rightarrow \mathbb{R}^+ \cup \{0\}$. A *binary constraint* refers to exactly two variables and is of the form $R_{ij} : D_i \times D_j \rightarrow \mathbb{R}^+ \cup \{0\}$.¹ For each binary constraint R_{ij} , there is a corresponding cost table T_{ij} with dimensions $|D_i| \times |D_j|$ in which the cost in every entry e_{xy} is the cost incurred when X_i is assigned to x and X_j is assigned to y . A *binary DCOP* is a DCOP in which all constraints are binary. A *partial assignment* is a set of value assignments to variables, in which each variable appears at most once. $\text{vars}(PA)$ is the set of all variables that appear in partial

¹ We say that a variable is *involved* in a constraint if it is one of the variables the constraint refers to.

assignment PA , i.e., $\text{vars}(PA) = \{X_i \mid \exists x \in D_i \wedge \langle X_i, x \rangle \in PA\}$. A constraint $R_j \in \mathcal{R}$ of the form $R_j : D_{j_1} \times D_{j_2} \times \dots \times D_{j_k} \rightarrow \mathbb{R}^+ \cup \{0\}$ is *applicable* to PA if each of the variables $X_{j_1}, X_{j_2}, \dots, X_{j_k}$ is included in $\text{vars}(PA)$. The *cost* of a partial assignment PA is the sum of all applicable constraints to PA over the value assignments in PA . A *complete assignment* (or a *solution*) is a partial assignment that includes all the DCOP's variables (i.e., $\text{vars}(PA) = \mathcal{X}$). An *optimal solution* is a complete assignment with minimal cost.

For simplicity, we make the common assumption that each agent holds exactly one variable (i.e., $n = m$) and we concentrate on binary DCOPs. These assumptions are common in the DCOP literature (e.g., [17, 26]). In addition to the standard motivation for focusing on binary DCOPs, in the case of Max-sum it is essential, since the runtime complexity of each iteration of Max-sum is exponential in the arity of the constraints.

2.3 The Max-Sum Algorithm

Max-sum operates on a *factor graph*, which is a bipartite graph in which the nodes represent variables and constraints [10]. Each variable-node representing a variable of the original DCOP is connected to all function-nodes representing constraints that it is involved in. Similarly, a function-node is connected to all variable-nodes representing variables in the original DCOP that are involved in it. Variable-nodes and function-nodes are considered “agents” in Max-sum (i.e., they can send and receive messages, and can perform computation).

A message sent to or from variable-node X (for simplicity, we use the same notation for a variable and the variable-node representing it) is a vector of size $|D_X|$ including a cost for each value in D_X . These costs are also called *beliefs*. Before the first iteration, all nodes assume that all messages they previously received (in iteration 0) include vectors of zeros. A message sent from a variable-node X to a function-node F in iteration $i \geq 1$ is formalized as follows:

$$Q_{X \rightarrow F}^i = \sum_{F' \in F_X, F' \neq F} R_{F' \rightarrow X}^{i-1} - \alpha \quad (1)$$

where $Q_{X \rightarrow F}^i$ is the message variable-node X intends to send to function-node F in iteration i , F_X is the set of function-node neighbors of variable-node X , and $R_{F' \rightarrow X}^{i-1}$ is the message sent to variable-node X by function-node F' in iteration $i - 1$. α is a constant that is reduced from all beliefs included in the message (i.e., for each $x \in D_X$) in order to prevent the costs carried by messages throughout the run of the algorithm from growing arbitrarily large.

A message $R_{F \rightarrow X}^i$ sent from a function-node F to a variable-node X in iteration i includes for each value $x \in D_X$:

$$\min_{PA_{-X}} \text{cost}(\langle X, x \rangle, PA_{-X}) \quad (2)$$

where PA_{-X} is a possible combination of value assignments to variables involved in F not including X . The term $\text{cost}(\langle X, x \rangle, PA_{-X})$ represents the cost of a partial assignment $a = \{\langle X, x \rangle, PA_{-X}\}$, which is:

$$f(a) + \sum_{X' \in X_F, X' \neq X, \langle X', x' \rangle \in a} (Q_{X' \rightarrow F}^{i-1})_{x'} \quad (3)$$

where $f(a)$ is the original cost in the constraint represented by F for the partial assignment a , X_F is the set of variable-node neighbors of F , and $(Q_{X' \rightarrow F}^{i-1})_{x'}$ is the cost that was received in the message sent from variable-node X' in iteration $i - 1$, for the value x' that is assigned to X' in a . X selects its value assignment $\hat{x} \in D_X$ following iteration k as follows:

$$\hat{x} = \arg \min_{x \in D_X} \sum_{F \in F_X} (R_{F \rightarrow X}^k)_x \quad (4)$$

In the synchronous version (*Syn_Max-sum*), at each iteration t , an agent waits to receive all messages sent to it in iteration $t - 1$ before performing computation and generating the messages to be sent in that iteration [30]. In the asynchronous version (*Asy_Max-sum*), agents react to messages they receive. Whenever a node receives a message, it performs computation and sends out messages to its neighbors, taking into consideration the last message received from each of its neighbors [6]. In both versions, the logic for the actions of the agents are identical, only the trigger for performing those actions is different.

2.3.1 Damped Max-sum (DMS)

DMS has an additional feature, which is the damping of the propagated beliefs. In order to add damping to Max-sum, a parameter $\lambda \in [0, 1)$ is used. Before sending a message in iteration k , an agent performs calculations as in standard Max-sum. We use $\widehat{m_{i \rightarrow j}^k}$ to denote the result of the calculation made by agent A_i for the content of a message intended to be sent from A_i to agent A_j in iteration k and $m_{i \rightarrow j}^{k-1}$ to denote the message sent by A_i to A_j at iteration $k - 1$. The message sent by A_i to A_j at iteration k is calculated as follows:

$$m_{i \rightarrow j}^k = \lambda m_{i \rightarrow j}^{k-1} + (1 - \lambda) \widehat{m_{i \rightarrow j}^k} \quad (5)$$

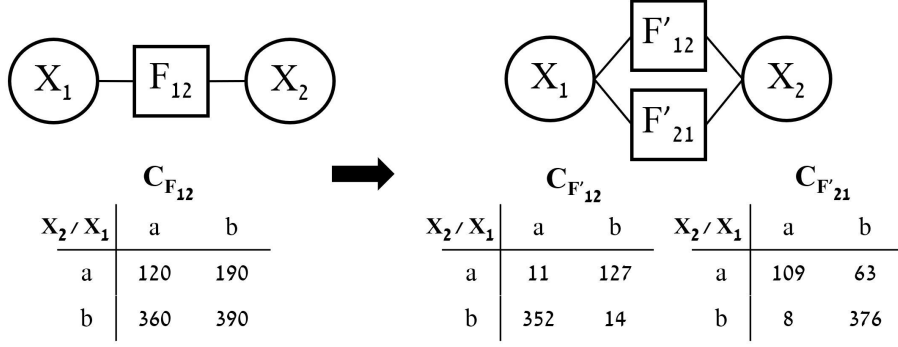
Thus, λ expresses the weight given to previously performed calculations with respect to the most recent calculation performed. Moreover, when $\lambda = 0$ the resulting algorithm is standard Max-sum.

We use *Syn_DMS* and *Asy_DMS* to denote the synchronous and asynchronous versions of DMS, respectively, in this paper.

2.3.2 Asynchronous Execution

All the definitions used for describing Max-sum (and DMS) above use the iteration number k . It was used to describe how a message is generated, using the information received by the factor graph node in the previous iteration ($k - 1$). In asynchronous execution, there are no iterations, and agents perform computation steps whenever they receive messages. Thus, in asynchronous execution, the information that a node N_i uses, when it generates a message in some time t , is, for each neighbor N_j , the information included in the last message received from N_j (prior to t), regardless of when it was sent by N_j . If no message has been received from N_j yet, N_i uses a vector of zeros in its computation. Notice, that in the presence of message delays, a node N_i may receive messages from its neighbor N_j , not in the order they were sent. This is true for both the synchronous and the asynchronous versions of the algorithm. Nevertheless, the agents use the messages in the order in which they were received.

In order to avoid this phenomenon, we implemented a time-stamp method that allowed the agents receiving messages to consider the information they include in the order that they were sent. However, the results were not significantly different from the results obtained when we did not use this method, thus, we do not report these results in our empirical study.



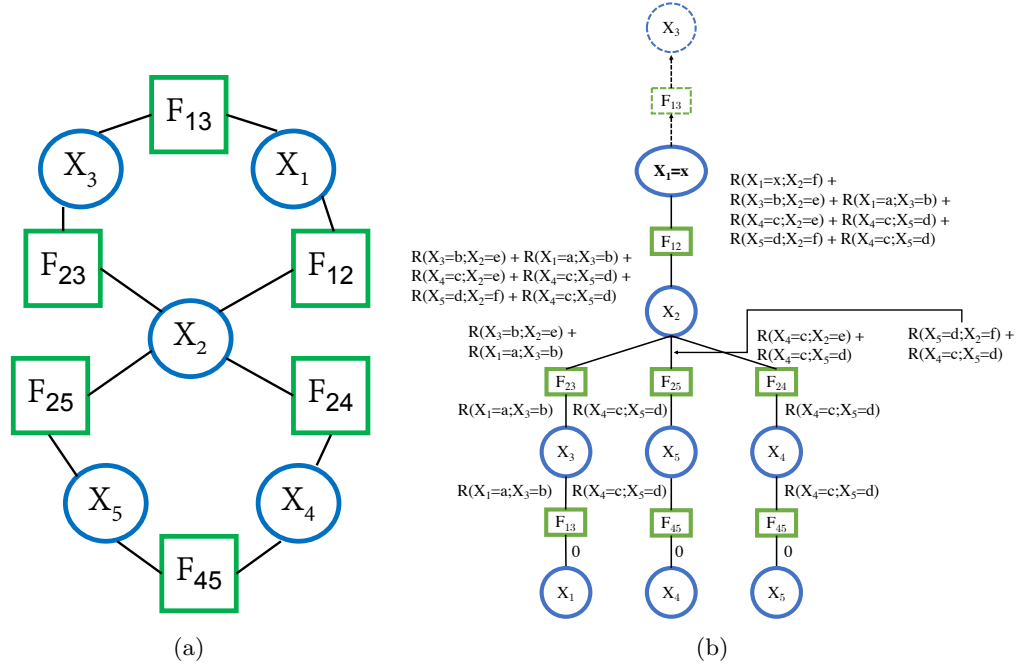
■ **Figure 1** An acyclic DCOP factor graph (on the left) and its equivalent SCFG (on the right).

2.3.3 Max-sum with Split Constraint Factor Graphs

When Max-sum is applied to an asymmetric problem, the representing factor graph has each (binary) constraint represented by two function-nodes, one for each part of the constraint held by one of the involved agents. Each function-node is connected to both variable-nodes representing the variables involved in the constraint [31]. Figure 1 presents two equivalent factor graphs that include two variable-nodes, each with two values in its domain, and a single binary constraint. On the left, the factor graph represents a (symmetric) DCOP including a single constraint between variables X_1 and X_2 , hence, it includes a single function node representing this constraint. On the right, the equivalent factor graph representing the equivalent asymmetric DCOP is depicted. It includes two function-nodes, representing the parts of the constraint held by the two agents involved in the asymmetric constraint. Thus, the cost table in each function-node includes the asymmetric costs that the agent holding this function-node incurs. In this example function-node F'_{12} is held by agent A_1 , while F'_{21} is held by A_2 . The factor graphs are equivalent since the sum of the two cost tables held by the function-nodes representing the constraints in the factor graph on the right, is equal to the cost table of the single function-node representing this constraint in the factor graph on the left (see [32] for details). Researchers have used such *Split Constraint Factor Graphs* (SCFGs) as an enhancement method for Max-sum [20, 4]. This is achieved by splitting each constraint that was represented by a single function-node in the original factor graph into two function-nodes. The SCFG is equivalent to the original factor graph if the sum of the cost tables of the two function-nodes representing each constraint in the SCFG is equal to the cost table of the single function-node representing the same constraint in the original factor graph. By tuning the similarity between the two function-nodes representing the same constraint one can determine the level of asymmetry in the SCFG. The use of symmetric SCFGs was shown to trigger very fast convergence to high quality solutions. However, generating mild asymmetry, postpones convergence and generates some exploration, which results in improved solution quality [4].

2.3.4 Non-Concurrent Logic Operations

In order to evaluate the performance of distributed algorithms performing in a distributed environment, there is a need to establish which of the operations performed by agents could not have been performed concurrently and, thus, the run-time performance of the algorithm is the longest non-concurrent sequence of operations that the algorithm performed. In [29], DisCSP algorithms were evaluated, which their basic logic operations were constraint



■ **Figure 2** (a) A lemniscate factor-graph. (b) An example of a BCT for a belief in the message sent from X_1 to the function-node F_{13} at time $t = 6$ in the lemniscate depicted on the left hand side.

checks (CCs), thus, the performance was measured in terms of non-concurrent constraint checks (NCCCs). In [14], search based complete algorithms were compared with inference algorithms, thus, algorithms that perform different atomic logic operations (i.e., constraint checks and compatibility checks) were compared, and the results were reported in terms of non-concurrent logic operations (NCLOs). This approach is the one we adopt in this study, since we evaluate the quality of the solutions of the algorithms, as a function of the asynchronous advancement of the algorithm, when agents perform computation concurrently.

Recently, these insights were generalized such that similar statements can be made when the algorithm is solving finite factor-graphs with multiple cycles [28]. Zivan *et al.* have proved that, as in the single cycle case, on every finite factor-graph, Max-sum at some point in time starts to repeatedly follow a path that minimizes its beliefs. When a large enough damping factor is used, this minimal path is indeed the minimal path in the factor-graph, and thus, if it is consistent, the algorithm converges to the optimal solution.

2.4 Backtrack Cost Trees

For analyzing the behavior of Max-sum on factor graphs with an arbitrary (finite) number of cycles, Zivan *et al.* proposed the use of a *backtrack cost tree* (BCT) [28]. It allows one to trace, for each belief, the entries in the cost tables held by function-nodes that were used to compose this belief. That is, what were the components of the assignment's cost. Their analysis included insights regarding the constructions of beliefs from costs incurred by constraints. Thus, for every pair of constrained variables, X_i and X_j , for each $x \in D_i$, $x' \in D_j$, the cost incurred by the constraint for assigning x to X_i and x' to X_j was denoted as $R(X_i = x, X_j = x')$. Formally, a BCT is defined as follows:

► **Definition 1.** A *Backtracking Cost Tree (BCT)* is defined for a belief that appears either in a message sent from variable X_i at time t , to a function node connecting it to a variable X_j or to a message sent from that function node to variable X_i . The belief is regarding the cost of assigning some $x \in D_i$ to X_i . Without loss of generality, we will elaborate on the first among these two and denote it as $BCT_{i \rightarrow j}^t$.

The belief, as constructed by the Max-sum algorithm, is a sum of various components, and the tree is composed from them. At the root is the belief, i.e., a cost for assigning some $x \in D_i$ to X_i , and it is connected to all nodes it received a message from at time $t - 1$, with the edges containing the beliefs it was passed that ended up in the calculation of the belief it sent. Each of those nodes is connected itself to the nodes that send it messages at time $t - 2$, with the edges containing the beliefs that passed to it that ended up in its message. The tree leaves are all at time 0 (see Figure 2 (b)).

For a single-cycle factor graph, the BCT for every belief is a chain. Factor graphs with multiple cycles include variable-nodes with more than two neighbors, and thus, the BCTs of their beliefs include nodes with multiple children.

A BCT starts from the end point (i.e., the root of the BCT as presented in Figure 2 (b)), which is the *belief* (cost) of assigning to X_i some value x from its domain D_i , as sent to a neighboring node. The values from which that belief was calculated can then be backtracked to the messages and costs due to all the individual constraints that were summed up to create that belief. An example of such a tree for a belief generated when Max-sum solves the factor-graph depicted in Figure 2(a) is depicted in Figure 2(b).

For each BCT, there is an implied assignment tree that consists of the value assignments that the variables at each time-point of the tree would need to be assigned in order to incur the costs included in the BCT. The value assignment selected by a variable at time t is the one with the minimal sum of beliefs sent to the corresponding variable-node at iteration $t - 1$. The tree for this minimal sum of beliefs will be denoted by BCT_i^t , as it does not depend on any specific belief that appears in a message to another variable.

2.5 Convergence Properties

Belief propagation converges in linear time to an optimal solution when the problem's corresponding factor graph is acyclic [16]. For a single-cycle factor graph, we know that if belief propagation converges, then it is to an optimal solution [8, 24]. Moreover, when the algorithm does not converge, it periodically changes its set of assignments. In order to explain this behavior, Forney *et al.* show the similarity of the performance of the algorithm on a cycle to its performance on a chain, whose nodes are similar to the nodes in the cycle, but whose length is equal to the number of iterations performed by the algorithm. One can consider a sequence of messages starting at the first node of the chain and heading towards its other end. Each message carries beliefs accumulated from costs added by function-nodes. Each function-node adds a cost to each belief, which is the constraint value of a pair of value assignments to its neighboring variable-nodes. Each such sequence of cost accumulation (route) must at some point become periodic, and the minimal belief would be generated by the minimal periodic route. If this periodic route is consistent (i.e., the set of assignments implied by the costs contain a single value assignment for each variable), then the algorithm converges. Otherwise, it does not [8].

Recently, these insights were generalized such that similar statements can be made when the algorithm is solving factor graphs with multiple cycles. Specifically (using BCTs), Zivan *et al.* proved that, as in the single cycle case, on every finite factor graph, Max-sum at some

point in time starts to repeatedly follow a path that minimizes its beliefs. When a large enough damping factor is used, this minimal path is indeed a minimal path in the factor graph, and thus, if it is consistent, then the algorithm converges to an optimal solution [28].

3 The Effect of Asynchronous Execution

In order to analyze the differences in the performance of Syn_Max-sum and Asy_Max-sum, one must investigate the differences in the structure of the BCTs of beliefs sent by the algorithms' nodes. In Syn_Max-sum, the height of a BCT for a belief included in a message sent at iteration t is t and, for each node in the tree, the heights of the sub-trees rooted by each of its children nodes are equal. On the other hand, in Asy_Max-sum, messages can have different delays and, thus, each sub-tree in a BCT can have a different height.

Our first theoretical property addresses the results proved in [28] regarding the convergence of the synchronous version of Max-sum (Syn_Max-sum). More specifically, we prove that the property that was proved in Lemma 1 in [28], and was used to prove the main theorem of this study (i.e., the main theorem in [28]), is not guaranteed when the algorithm is performed asynchronously in an environment that includes message latency.

► **Proposition 1.** *In the presence of message delays, unlike Syn_Max-sum, Asy_Max-sum is not guaranteed to converge to a minimal repeated route.*

Proof. The structure of the BCTs of the beliefs that are exchanged by agents, depend on the timing of the arrival of messages from which they are composed. Each BCT (and as a result, the corresponding belief that it demonstrates its construction), is an outcome of a specific combination of message delays, resulting in different orders of message arrivals and the number of such combinations is exponential in the maximal number of messages that the beliefs they carry can be included in the BCT. Moreover, the combination of message delays that resulted in a specific minimal route of beliefs is not guaranteed to repeat itself. Thus, even if the algorithm reaches a minimal route, it may not repeat it. ◀

The proposition above seems to put an end to the natural wish that the convergence property of Syn_Max-sum can be established for Asy_Max-sum as well. However, the differences between the executions of the two versions of the algorithm can be minimized. More specifically, the effect caused by sub-trees of the BCTs having different heights in Asy_Max-sum can be significantly reduced through the use of damping.

Denote by $layer_k$ the set of nodes of a BCT with depth k (distance from the root), and by BCT_k the layers of the BCT with depth k or less. We will say that a $layer_k$ is *effective* if and only if there exists a belief calculated using BCT_k that is different than the belief calculated when taking into consideration the complete BCT. For each BCT B , we say that its effective BCT B' is $BCT_{k'}$ such that $layer_{k'}$ is effective and for any $layer_k$ that is effective in B , $k' \geq k$.

► **Lemma 1.** *When asynchronous DMS (Asy_DMS) is performed with a large enough damping factor², in an environment including bounded message delays, there exists a finite number of non-concurrent steps³ of the algorithm ns_1 , such that in the steps following it, for every two beliefs included in the same message, if $layer_k$ in each of the corresponding BCTs is effective, then the number of nodes in $layer_k$ of both BCTs are equal.*

² For an analysis of the size of the damping factor required, with respect to the largest number of neighbors (degree) that a node in the factor graph has, see [28].

³ We consider a step to be an action that starts when a node in the graph received some messages (at least one), performed computation and ends when it sent some messages (at least one).

Proof. Since delays are bounded, there exists a number of non-concurrent steps $ns_0 < ns_1$ in which the roots of the BCTs of all beliefs received in messages for every step following ns_0 have the same number of children. This will be true for all non-concurrent steps $ns > ns_0$ and, thus, layers of BCTs of beliefs that are sent in the same message with depth k following $ns \geq ns_0 + \delta k$ (where δ is the maximal size of a message delay, in terms of non-concurrent steps) must have the same number of nodes. Damping with a large enough damping factor, causes the bottom layers of BCTs to have less influence on the calculation made by the nodes in the algorithm following each computation step (see [28] for details). Let ϵ denote the smallest cost that can affect the nodes' actions in the algorithm. If we wait for a sufficiently large enough number of steps, the maximal sum of costs in the BCTs, of steps performed before ns_0 will be smaller than ϵ . We use ns_1 to denote that sufficiently large enough number of steps. ◀

An immediate corollary from Lemma 1 is that in Asy_DMS (which is using a large enough damping factor), following ns_1 , the effective BCTs of all beliefs included in each message have the same number of nodes. This reduces the possible differences between beliefs that can be generated by each node. Moreover, for the case that the algorithm does converge, the effect of the asynchronous performance vanishes, as we prove below.

► **Proposition 2.** *When Asy_DMS using a large enough damping factor, is performed in an environment with bounded message delays, if after performing $ns_2 > ns_1$ (ns_1 as described in Lemma 1) non-concurrent steps, it reaches a minimal consistent route (i.e., all nodes perform k sequential asynchronous steps in which the value assignments corresponding to the minimal route are selected), then it will repeatedly follow this route (i.e., it has converged).*

Proof. As established above, following ns_1 , the effective BCTs for beliefs included in the same message have the same number of nodes (in each layer and altogether) regardless of message delays. When the algorithm reaches a minimal consistent route, the beliefs corresponding to this minimal route involve only one value in each domain, and the belief corresponding to it is minimal in each message. Additional nodes added to the BCTs of the beliefs corresponding to the assignments in the minimal route represent costs in the entries of the cost tables of function-nodes that are part of the minimal route. Hence, they will not change its minimal property or the choice of the minimal route assignments, i.e., for every $ns > ns_2$ the effective BCT_i^{ns} will be identical. Similarly, the addition of nodes to BCTs of beliefs corresponding to assignments that are not included in the minimal route represent costs that belong to routes with larger overall costs. ◀

Proposition 2 has a major importance to our discussion. Both the asynchronous and the synchronous versions of DMS will converge when they reach a consistent minimal path (i.e., the differences between them can exist only when the minimal path is inconsistent. In such a case, the synchronous execution version will repeat the minimal non consistent route while the asynchronous execution version may leave it and explore other routes).

4 Experimental Evaluation

In order to evaluate the implications of asynchronous execution (compared to synchronous execution) and message latency on the different versions of Max-sum, we used an asynchronous simulator, in which agents are implemented by Java threads. It includes a *mailing agent* that simulates the delays of messages as suggested by [29]. Using this type of simulator allows us to implement any type of message delay pattern. Other simulators, such as ns-3 [12, 1], offer a

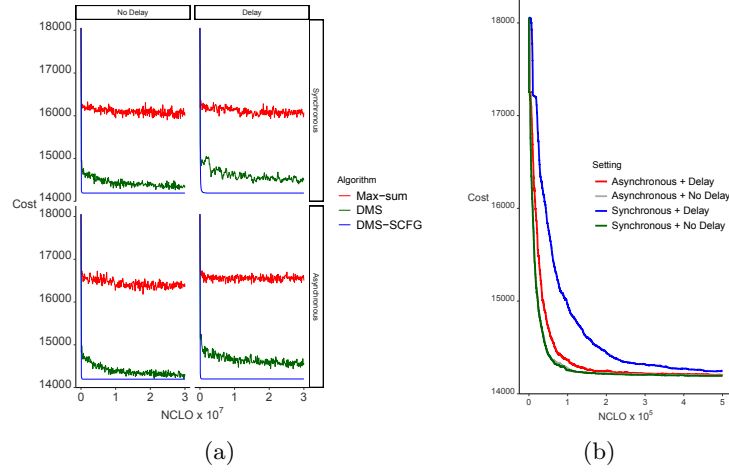
number of communication patterns from which one can select. However, we prefer the use of the simulator proposed in [29], which allows complete flexibility in the design of the message delay pattern and it allows to measure run-time in implementation independent units. Thus, the results are presented as a function of the number of *non-concurrent logic operations* (NCLOs). The atomic logic operations in these algorithms are the evaluation of the cost of a combination of two assignments (i.e., an access to the cost table of a function-node). Each agent performed the computation for the function-nodes that were assigned to it. We used a greedy heuristic to evenly assign function-nodes to agents and, thus, increase concurrency. In order to simulate message delays, for each message sent between nodes that their roles were performed by different agents, a delay in terms of NCLOs was selected, and the message was delivered to the receiving agent after that agent had the opportunity to perform this number of logic operations.

We evaluated the algorithms on problems including 50 agents, which are too large for complete DCOP algorithms to solve. These included random graph problems, graph coloring problems, scale-free network problems, and overlapped solar systems problems (details below).

In each experiment, we randomly generated 50 different problem instances. The results presented in the graphs are an average of those 50 runs. In order to demonstrate the convergence of the algorithms, we present the sum of costs of the constraints involved in the assignment that would have been selected by each algorithm every 100K NCLOs. We also performed *t-tests* to evaluate the significance of differences between all presented results.

As mentioned above, the experiments were performed on four types of distributed constraint optimization problems. Each type of problem exhibits a different level of structure in the constraint graph topology and in the constraint functions. All problems were formulated as minimization problems.

- **Random Graph Problems:** These problems are random constraint graph topologies with density $p_1 = \{0.1, 0.6\}$. They include variables with 10 values in each domain. The cost tables held by function-nodes include costs that were selected uniformly between 100 and 200. Both the constraint graph and the constraint functions are unstructured.
- **Graph Coloring Problems:** These problems are random constraint graph topologies in which each variable has three values (i.e., colors), and all constraints are “not-equal” cost functions, where an equal assignment of neighbors in the graph incurs a random cost between 100 and 200 and non equal value assignments incur zero cost. Such random graph coloring problems are commonly used in DCOP formulations of resource allocation problems. We set the density to $p_1 = 0.05$ and had three values (i.e., colors) in each domain [27, 6, 4].
- **Scale-free Network Problems:** Problems generated using the model by [2]. An initial set of 10 agents was randomly selected and connected. Additional agents were added sequentially and connected to 3 other agents with a probability proportional to the number of links that the existing agents already had. The cost of each joint assignment between constrained variables was independently drawn from the discrete uniform distribution from 100 to 199. Each variable had 10 values in its domain. Similar problems were previously used to evaluate DCOP algorithms by Kiekintveld *et al.* [9]. The constraint graph is somewhat structured but the constraint functions are unstructured.
- **Overlapped Solar Systems Problems:** The overlapped solar system is a realistic problem, inspired by the Constant Speed Propagation Delay Model implemented in the ns-3 simulator [12, 1]. The graph topology is inspired by scale-free networks. An initial set of 5 agents are randomly selected to be the centers of the solar systems, and they are connected. Each of these agents A_i^c is assigned two coordinates that are drawn from a



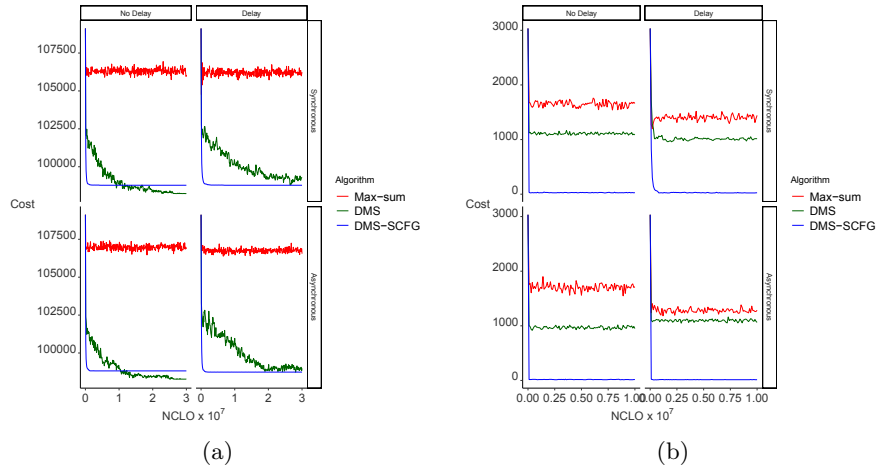
■ **Figure 3** (a) Solution quality as a function of $NCLOs$, of Max-sum versions solving sparse random problems ($p_1 = 0.1$). (b) A closer look at the solution quality of DMS-SCFG versions on these problems.

continuous uniform distribution: $x_i^c \sim U(0, 1)$ and $y_i^c \sim U(0, 1)$. All other agents (i.e., stars in the solar systems) are randomly assigned to one of the solar systems. The index c represents the solar system in which the agent is assigned too, and it is equal to the index of the center agent of the solar system (i.e., if A_i^c is the center of a solar system, then $i = c$). The coordinates for an assigned agent (A_j^c where $j \neq c$) are drawn from a Normal distribution as follows: $x_j^c \sim N(\mu = x_i^c, \sigma = 0.05)$ and $y_j^c \sim N(\mu = y_i^c, \sigma = 0.05)$ based on the location of the center of the solar system that it was attached to.

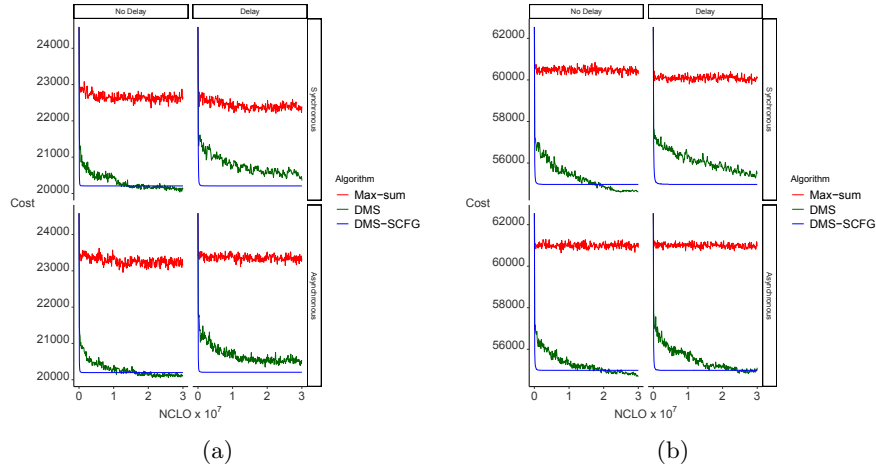
The probability that two arbitrary agents A_i and A_j will be neighbors is defined by $p_{ij} = (1 - \frac{distance_{ij}}{maxDistance})^\beta$ where $distance_{ij}$ is the Euclidean distance between agents A_i and A_j , $maxDistance$ is the Euclidean distance between agent A_i to the farthest agent, and β expresses the changes in the probability that both agents will be neighbors as a function of their distance (in our experiments we used $\beta = 3$). For each pair agents, a random probability $p_r \in [0, 1]$ was generated, and two agents are considered as neighbors if $p_r < p_{ij}$. Costs between connected agents were selected uniformly between 100 and 200.

While the structure of these problems is similar to scale-free networks, the addition of the geographic locations of nodes allows one to calculate the size of message delays with respect to physical distance as specified below.

For random uniform problems, graph coloring problems, and scale-free network problems, all algorithms were run in a setup with no message delays and a setup with random message delays selected uniformly from the range $(0, 10K)$ NCLOs. For overlapped solar systems problems, in addition to the no message delay setup, the delay for each sent message between agents A_i and A_j was drawn from a Poisson distribution $Poisson(\Gamma \cdot distance_{ij})$ NCLOs where Γ is the average delay. This is in contrast to the Constant Speed Propagation Delay Model implemented in ns-3 where the delays that were calculated as a function of the distance between the geographic location of the nodes in the communication graph, were fixed and not sampled [12, 1].



■ **Figure 4** Solution quality as a function of *NCLOs*, of Max-sum versions solving dense random problems ($p = 0.6$) (a) and graph coloring problems (b)).

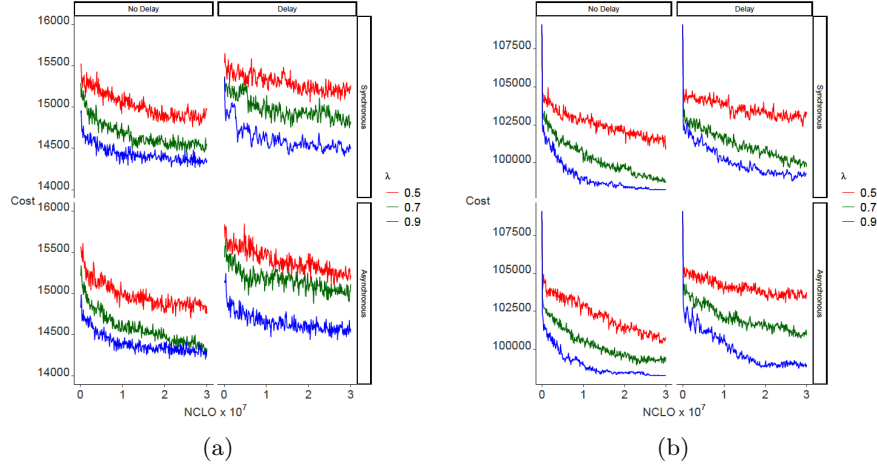


■ **Figure 5** Solution quality as a function of *NCLOs*, of Max-sum versions solving scale-free network problems (a) and overlapped solar systems problems (b)).

4.1 Results

Figure 3(a) presents the quality of solutions produced by the different versions of Max-sum when solving sparse random graph problems with $p_1 = 0.1$. Each figure presented in this section includes four graphs, presenting results of the algorithms when performing synchronously, asynchronously, with message delays and without. The versions include Max-sum, DMS with $\lambda = 0.9$, DMS-SCFG.⁴ Asy_Max-sum (with and without message delays) traversed solutions with higher costs on average than Syn_Max-sum. The results of the different runs of the algorithms were scattered and, thus, the differences from the synchronous versions were not found to be statistically significant. Asy_DMS, on the other hand, performed similarly to Syn_DMS, with and without message delays (as expected following Proposition 1).

⁴ DMS-SCFG is the damped Max-sum (DMS) algorithm with split constraint factor graphs (SCFGs). We used the 0.4-0.6 version of DMS-SCFG, which was found to perform best by [4].



■ **Figure 6** Solution quality as a function of *NCLOs*, of DMS with different λ values, solving random uniform problems with $p_1 = 0.1$ (a) and $p_1 = 0.6$ (b)).

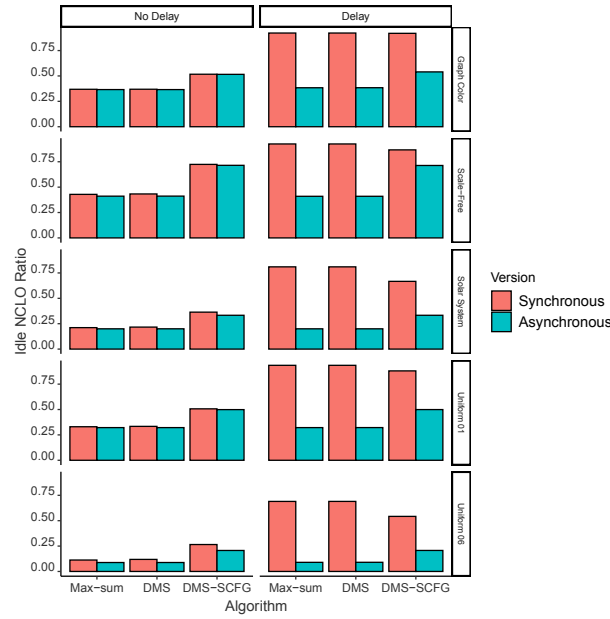
Another observation is that all versions of DMS-SCFG converged very fast compared to the other versions of the algorithm. Figure 3(b) provides a closer look that allows one to better compare their convergence rates. Both the synchronous and the asynchronous versions converge at the same rate in environments that do not include message delays. Clearly, message delays affect the synchronous version more than the asynchronous version of the algorithm. Nevertheless, in all execution modes, the algorithm converges very fast to solutions with the same quality.

Figure 4(a) presents the results for the same algorithms solving dense random graph problems with $p_1 = 0.6$. While the results seem similar to the results presented in Figure 3(a), there are fewer differences between the Max-sum versions. On the other hand, on these problems, the DMS versions in scenarios that do not include message delays find high quality solutions faster and converge.

Figure 4(b) presents the results of the algorithms solving graph coloring problems. It is apparent that the exploration performed by Max-sum and DMS is less effective on these problems, and thus, the advantage of DMS-SCFG is prominent. Moreover, in the presence of message delays, standard Max-sum improves its performance. We assume that delays break the very structured execution on this type of problems, and has a positive exploration affect. This affect is diminished when damping for the same properties that we established in the section titled “The Effect of Asynchronous Execution.”

The results of the algorithms when solving scale free network and the overlapping solar system problem are presented in in Figure 5. They were found to be similar to the results presented in Figure 4(a) for the dense random problems. The differences in the performance of Asy_Max-sum from Syn_Max-sum was found to be significant when solving scale-free networks, with and without message delays. No significant difference was found between the synchronous and asynchronous versions when solving overlapped solar system problems. It seems for these problems that the similar structure has a more major effect on the behavior of the algorithms than the pattern of the message delays.

In our second set of experiments we evaluated the influence of the selection of the damping factor on the effect that asynchronous execution and message latency have on DMS’s performance. Figure 6 presents the results of the algorithm with three different values of the damping parameter, i.e., $\lambda = 0.5$, $\lambda = 0.7$ and $\lambda = 0.9$, solving sparse (a) and dense



■ **Figure 7** Ratio between the number of *NCLOs* in which the agents were idle and the total number of *NCLOs* for all algorithms and all execution modes.

(b) random uniform problems. As expected from the properties established in Propositions 1 and 2, asynchronous execution affects the performance of all versions of DMS when it does not converge. However, it is apparent that the $\lambda = 0.9$ version is less affected by message delays in the asynchronous execution, as expected. Similar results were obtained for all types of problems and were omitted to avoid redundancy.

In order to compare the effect that message delays have on the agents performing synchronously and asynchronously, we measured the average number of *NCLOs* in which agents were idle in each mode of execution of the algorithm. The results are presented in Figure 7. It includes for each algorithm, in each mode of execution, the average ratio of the number of *NCLOs* in which the agent was idle (i.e., waiting for message to arrive) and the total number of *NCLOs* the algorithm was executed. It is apparent that when solving all problem types, the agents performing asynchronously spend less time idle than the agents performing synchronously. This difference between the performance of the synchronous and the asynchronous versions was most apparent in *DMS_SCFG*. Nevertheless, while the difference in the time the agents spent idle when performing this type of the *Max-sum* algorithm, the synchronous and the asynchronous versions were most similar in their convergence time and the solution quality.

4.2 Discussion

The advantage of *DMS* over standard *Max-sum*, when solving graphs with multiple cycles, was reported empirically in a number of studies (e.g., [4]) and explained theoretically by [28]. In *Max-sum*, costs that are aggregated in the beginning of the run are duplicated in every node of the graph that has more than two neighbors and, thus, they are taken into consideration an exponential number of times in the calculation of beliefs and in the assignment selection. Damping reduces the weight of these costs in the belief calculation until it becomes negligible. A similar phenomenon reduces the differences between the performance of *Syn_DMS* and

Asy_DMS. As we established in the corollary of Lemma 1, when using a large enough damping factor, the effect of BCTs with different heights is eliminated in DMS and, thus, after enough NCLOs are performed, the effective BCTs of the beliefs in each message have the same number of nodes. The results comparing DMS with different damping factor values, demonstrate the need to use a high damping factor in order to achieve robustness to message delays. This empirical evidence, strengthens the property established in Lemma 1 and its corollary, that if the damping factor used is not high enough, the effect of the lower layers of the BCTs, which may have different structure and a different number of nodes, on the generation of beliefs by the nodes, is not eliminated. Thus, message delays have a greater effect on the algorithm's performance when the damping factor used is not low. Finally, Asy_DMS-SCFG maintains the fast convergence properties and the quality of the solutions of the synchronous version. It is also robust to message latency.

5 Conclusions

In this paper, we filled the gap in the Max-sum literature on the difference of synchronous and asynchronous executions of the algorithm in distributed environments. Our theoretical analyses revealed that, unlike its synchronous counterpart, the asynchronous version of Max-sum in the presence of message latency can cause the propagation of inconsistent beliefs, resulting in the loss of guaranteed properties (Proposition 1). However, not all is lost as one can use damping to minimize this effect and, subsequently, ensure that when asynchronous DMS finds a minimal route, it will converge, as does the synchronous version (Proposition 2). Finally, experimental results show that when the algorithm is further optimized through split constraint factor graphs, it converges very fast to high-quality solutions even in the presence of message delays. Taken together, these results extend significantly our understanding of Max-sum in distributed environments with more realistic messaging assumptions, propose algorithmic tools that are theoretically grounded to alleviate the issues raised, and enable a more effective use of Max-sum by real-world practitioners.

References

- 1 Andy Bubune Amewuda, Ferdinand Apietu Katsriku, and Jamal-Deen Abdulai. Implementation and evaluation of wlan 802.11ac for residential networks in ns-3. *Journal of Computer Networks and Communications*, 2018, 2018.
- 2 Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
- 3 Ziyu Chen, Yanchen Deng, Tengfei Wu, and Zhongshi He. A class of iterative refined max-sum algorithms via non-consecutive value propagation strategies. *Auton. Agents Multi Agent Syst.*, 32(6):822–860, 2018.
- 4 Liel Cohen, Rotem Galiki, and Roie Zivan. Governing convergence of max-sum on dcops through damping and splitting. *Artificial Intelligence Journal (AIJ)*, 279, 2020.
- 5 Yanchen Deng and Bo An. Speeding up incomplete gdl-based algorithms for multi-agent optimization with dense local utilities. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence, (IJCAI)*, pages 31–38, 2020.
- 6 A. Farinelli, A. Rogers, A. Petcu, and N. R. Jennings. Decentralised coordination of low-power embedded devices using the max-sum algorithm. In *Proceeding of the 7th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 639–646, 2008.
- 7 Alessandro Farinelli, Alex Rogers, and Nick R. Jennings. Agent-based decentralised coordination for sensor networks using the max-sum algorithm. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 28(3):337–380, 2014.

- 8 G David Forney, Frank R Kschischang, Brian Marcus, and Selim Tuncel. Iterative decoding of tail-biting trellises and connections with symbolic dynamics. In Brian Marcus and Joachim Rosenthal, editors, *Codes, Systems, and Graphical Models*, pages 239–264. Springer, 2001.
- 9 C. Kiekintveld, Z. Yin, A. Kumar, and M. Tambe. Asynchronous algorithms for approximate distributed constraint optimization with quality bounds. In *AAMAS*, pages 133–140, 2010.
- 10 F. R. Kschischang, B. J. Frey, and H. A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47:2:181–208, 2001.
- 11 Radu Marinescu and Rina Dechter. AND/OR branch-and-bound search for combinatorial optimization in graphical models. *Artif. Intell.*, 173(16-17):1457–1491, 2009.
- 12 Lesly Mayuga-Marcillo, Luis Urquiza-Aguilar, and Martha Paredes-Paredes. Wireless channel 802.11 in ns-3, 2018.
- 13 P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. Adopt: asynchronous distributed constraints optimization with quality guarantees. *Artificial Intelligence Journal (AIJ)*, 161:1-2:149–180, 2005.
- 14 Arnon Netzer, Alon Grubshtein, and Amnon Meisels. Concurrent forward bounding for distributed constraint optimization problems. *Artificial Intelligence Journal (AIJ)*, 193:186–216, 2012.
- 15 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.
- 16 J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco, California, 1988.
- 17 A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, (IJCAI)*, pages 266–271, 2005. URL: <http://www.ijcai.org/papers/0445.pdf>.
- 18 Ben Rachmut, Roie Zivan, and William Yeoh. Latency-aware local search for distributed constraint optimization. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1019–1027, 2021.
- 19 S. D. Ramchurn, A. Farinelli, K. S. Macarthur, and N. R. Jennings. Decentralized coordination in robocup rescue. *Computer J.*, 53(9):1447–1461, 2010.
- 20 Nicholas Ruozzi and Sekhar Tatikonda. Message-passing algorithms: Reparameterizations and splittings. *IEEE Trans. Information Theory*, 59(9):5860–5881, 2013.
- 21 Pierre Rust, Gauthier Picard, and Fano Ramparany. Using message-passing DCOP algorithms to solve energy-efficient smart environment configuration problems. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence, (IJCAI)*, pages 468–474, 2016.
- 22 R. Stranders, A. Farinelli, A. Rogers, and N. R. Jennings. Decentralised coordination of mobile sensors using the max-sum algorithm. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, (IJCAI)*, pages 299–304, 2009.
- 23 W. T. Luke Teacy, Alessandro Farinelli, N. J. Grabham, Paritosh Padhy, Alex Rogers, and Nicholas R. Jennings. Max-sum decentralized coordination for sensor systems. In *Proceeding of the 7th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1697–1698, 2008.
- 24 Yair Weiss. Correctness of local probability propagation in graphical models with loops. *Neural Computation*, 12(1):1–41, 2000.
- 25 Chen Yanover, Talya Meltzer, and Yair Weiss. Linear programming relaxations and belief propagation - an empirical study. *Journal of Machine Learning Research*, 7:1887–1907, 2006.
- 26 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.
- 27 W. Zhang, Z. Xing, G. Wang, and L. Wittenburg. Distributed stochastic search and distributed breakout: properties, comparison and applications to constraints optimization problems in sensor networks. *Artificial Intelligence*, 161:1-2:55–88, January 2005.

- 28 Roie Zivan, Omer Lev, and Rotem Galiki. Beyond trees: Analysis and convergence of belief propagation in graphs with multiple cycles. In *Proceedings of the 34th International Conference of the Association for the Advancement of Artificial Intelligence (AAAI)*, pages 7333–7340, 2020.
- 29 Roie Zivan and Amnon Meisels. Message delay and discsp search algorithms. *Annals of Mathematics and Artificial Intelligence (AMAI)*, 46:415–439, 2006.
- 30 Roie Zivan, Tomer Parash, Liel Cohen, Hilla Peled, and Steven Okamoto. Balancing exploration and exploitation in incomplete min/max-sum inference for distributed constraint optimization. *Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 31(5):1165–1207, 2017.
- 31 Roie Zivan, Tomer Parash, Liel Cohen-Lavi, and Yarden Naveh. Applying max-sum to asymmetric distributed constraint optimization problems. *Journal of Autonomous Agents and Multi Agent Systems (JAAMAS)*, 34(1):13, 2020.
- 32 Roie Zivan, Tomer Parash, and Yarden Naveh. Applying max-sum to asymmetric distributed constraint optimization. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 432–439, 2015.