

John Hooker (Ed.)

LNCS 11008

Principles and Practice of Constraint Programming

24th International Conference, CP 2018
Lille, France, August 27–31, 2018
Proceedings



Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, Lancaster, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Friedemann Mattern

ETH Zurich, Zurich, Switzerland

John C. Mitchell

Stanford University, Stanford, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

C. Pandu Rangan

Indian Institute of Technology Madras, Chennai, India

Bernhard Steffen

TU Dortmund University, Dortmund, Germany

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum


Max Planck Institute for Informatics, Saarbrücken, Germany

More information about this series at <http://www.springer.com/series/7408>

John Hooker (Ed.)

Principles and Practice of Constraint Programming

24th International Conference, CP 2018
Lille, France, August 27–31, 2018
Proceedings

Editor
John Hooker 
Carnegie Mellon University
Pittsburgh, PA
USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-98333-2 ISBN 978-3-319-98334-9 (eBook)
<https://doi.org/10.1007/978-3-319-98334-9>

Library of Congress Control Number: 2018950526

LNCS Sublibrary: SL2 – Programming and Software Engineering

© Springer Nature Switzerland AG 2018, corrected publication 2018, 2019

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

This volume contains the proceedings of the 24th International Conference on the Principles and Practice of Constraint Programming (CP 2018) held August 27–31, 2018, in Lille, France. Detailed information about the CP 2018 conference can be found at <http://cp2018.a4cp.org>.

The CP conference is the annual international conference on all aspects of computing with constraints including theory, algorithms, environments, languages, models, systems, and applications such as decision-making, resource allocation, scheduling, configuration, and planning. The organizers of CP 2018 made a particular effort to build bridges to related fields that may provide new applications for CP. This theme was reflected in invited plenary talks, tutorials, a panel session, and seven themed tracks in addition to the main technical track: Applications; CP and Data Science; CP and Music; CP and Operations Research; CP, Optimization, and Power System Management; Multiagent and Parallel CP; and Testing and Verification. Each track had its own track chair(s) and Program Committee to ensure that the papers would be peer reviewed by experts in the relevant field.

The 114 submitted papers were allocated to tracks specified by the author(s). Each paper received at least three reviews. A total of 395 reviews were provided by Program Committee members, and 44 by external reviewers. The review process in each themed track was managed by the respective track chair, while papers submitted to the Main Technical Track were assigned to a senior Program Committee member, who conducted the discussion for that paper. The senior Program Committee consisted of seven prominent researchers from the CP community as well as the 11 track chairs and the conference program chair. Authors had an opportunity to respond to the initial reviews. Following this, the senior Program Committee conducted an intense asynchronous online discussion of the papers via EasyChair over an 11-day period, involving the regular Program Committee members as needed.

The reviewing process was double blind, meaning that authors and reviewers were anonymous to each other throughout the review process. In addition, Program Committee members indicated potential conflicts of interest by selecting from a list of submitting authors those with whom they had professional relationships. This prevented members from seeing the reviews or participating in the discussion of any papers with which they had a conflict of interest. In addition, papers submitted by track chairs to their own track were transferred to other track chairs, who managed the review process and obtained reviewers as necessary from the relevant Program Committees. The final deliberations of the senior Program Committee were conducted so as to respect all conflict-of-interest restrictions.

The senior Program Committee selected 50 papers for presentation at the conference, resulting in an acceptance rate of 44%. The committee also awarded the Best Paper Prize to Emmanuel Hebrard and George Katsirelos for their paper, “Clause Learning and New Bounds for Graph Coloring.” This paper was presented in a plenary

session, along with a presentation by the recipient of the ACP Doctoral Research Award. In addition, the authors of six outstanding papers were offered an opportunity to publish a longer version in *Constraints* rather than in the conference proceedings, and three accepted this offer. Because the longer versions would not appear in time for the conference, the original conference versions of the papers were posted on the conference website along with this proceedings volume. Abstracts of these papers appear at the end of the volume.

The first day of the conference was allocated to four workshops and the Doctoral Program. The workshops were the International Workshop on Graphs and Constraints, the Second Workshop on Progress Towards the Holy Grail, Constraints and AI Planning, and the 17th International Workshop on Constraint Modelling and Reformulation. The Doctoral Program afforded 26 participating students an opportunity to present their work, meet one-on-one with a senior researcher, and attend invited talks targeted to the experiences of a PhD student.

The main conference program featured three invited talks that described opportunities to apply CP technology in related fields. Srinivas Bollapragada, Chief Scientist at General Electric's Global Research Center, presented industrial scheduling problems that have heretofore been addressed by operations research methods. James Cussens, Senior Lecturer in Computer Science at the University of York, showed how CP can contribute to machine learning. Malte Helmert, Head of the AI Research Group at the University of Basel, discussed the role of constraints in automated planning. In addition, the program included several hour-long tutorials that showed how to formulate problems for modelling and solution software systems in related fields. Finally, a plenary panel session discussed opportunities for collaboration between the CP and automated planning communities.

A conference is a more complicated affair than is often thought, presenting literally hundreds of issues that must be resolved for a successful event. Our thanks go to Conference Chairs Gilles Audemard and Christophe Lecoutre for securing financial support and making the many necessary arrangements. In addition, we thank Publicity Chair George Katsirelos, Workshop Chair Sébastien Tabary, and Doctoral Program Chairs Anastasia Paparrizou and Nadjib Lazaar for their service.

The quality of a conference program relies on the hard work of many reviewers. CP 2018 is indebted to 117 members of eight Program Committees, some of whom served on multiple committees. We also thank the track chairs for recruiting their Program Committees and managing the review process in their tracks; they include Meinolf Sellmann (Applications), Michele Lombardi and Tias Guns (CP and Data Science), Charlotte Truchet (CP and Music), David Bergman and And e Cir e (CP and Operations Research), Bhagyesh Patil (CP, Optimization, and Power System Management), Ferdinando Fioretto and William Yeoh (Multiagent and Parallel CP), and Arnaud Gotlieb and Nadjib Lazaar (Testing and Verification). Special thanks go to the senior Program Committee for moderating discussions and making the tough final decisions.

Finally, we are grateful to our financial sponsors, which include *Artificial Intelligence*, Association for Constraint Programming, Association Fran aise pour la Programmation par Contraintes, Centre de Recherche en Informatique de Lens, Centre

National de la Recherche Scientifique, Cosling, European Association for Artificial Intelligence, Horizontal Software, Huawei, N-SIDE, ROADEF, Siemens, and Université d'Artois.

June 2018

John Hooker

Workshops and Tutorials

Workshops

Constraints and AI Planning

Christopher Beck	University of Toronto, Canada
Michael Cashmore	King's College, London, UK
Malte Helmert	University of Basel, Switzerland
Gilles Pesant	École Polytechnique de Montréal, Canada

International Workshop on Graphs and Constraints

Stefan Mengel	CRIL, Université d'Artois, France
Florent Capelli	Université de Lille

Second Workshop on Progress Towards the Holy Grail

Eugene Freuder	University College Cork, Ireland
----------------	----------------------------------

17th International Workshop on Constraint Modelling and Reformulation

Kevin Leo	Monash University, Australia
Alan Frisch	University of York, UK

Tutorials

Xpress Mosel Tutorial: Modelling and Solving Optimization Problems with Various Solvers

Sébastien Lannez	FICO, France
------------------	--------------

Automated Modeling with Conjure and Savile Row

Özgür Akgün	University of St. Andrews, UK
Peter Nightingale	University of St. Andrews, UK

MiniZinc: An Expressive Extensible Modelling Language

Peter Stuckey	University of Melbourne, Australia
Guido Tack	Monash University, Australia

Model-Based Optimization: Principles and Trends

Robert Fourer	Northwestern University, USA
---------------	------------------------------

Machine Learning for SAT Solvers

Jia Hui Liang	University of Waterloo, Canada
---------------	--------------------------------

Workshop Chair

Sébastien Tabary CRIL, Université d'Artois, France

Doctoral Program Chairs

Nadjib Lazaar LIRMM Montpellier, France
Anastasia Paparrizou CRIL, Université d'Artois, France

Publicity Chair

George Katsirelos INRA Toulouse, France

Senior Program Committee

Maria Garcia de la Banda Monash University, Australia
Laurent Michel University of Connecticut, USA
Gilles Pesant Polytechnique Montréal, Canada
Louis-Martin Rousseau Polytechnique Montréal, Canada
Peter Stuckey University of Melbourne, Australia
Pascal Van Hentenryck Georgia Institute Of Technology, USA
Roland Yap National University of Singapore
Track chairs (ex officio)

Main Technical Track Program Committee

Carlos Ansótegui Universitat de Lleida, Spain
Fahiem Bacchus University of Toronto, Canada
Roman Barták Charles University, Czech Republic
Chris Beck University of Toronto, Canada
Nicolas Beldiceanu IMT Atlantique (LS2N), France
Clément Carbonnel University of Oxford, UK
Mats Carlsson RISE, Sweden
David Cohen Royal Holloway, University of London, UK
Simon De Givry INRA, France
Sophie Demassey CMA, MINES ParisTech, France
Agostino Dovier Università degli Studi di Udine, Italy
Pierre Flener Uppsala University, Sweden
Carmen Gervet Université de Montpellier, France
Arnaud Gotlieb SIMULA Research Laboratory, Norway
Emmanuel Hebrard LAAS, CNRS, France
Matthias Heizmann Universität Freiburg, Germany
Hiroshi Hosobe Hosei University, Japan
Said Jabbour Université d'Artois, France
Peter Jeavons University of Oxford, UK
Philip Kilby NICTA, Australia

Zeynep Kiziltan	Università di Bologna, Italy
Philippe Laborie	IBM France
Jimmy Lee	The Chinese University of Hong Kong, SAR China
Boonping Lim	Australian National University
Andrea Lodi	École Polytechnique de Montréal, Canada
Samir Loudni	Université de Caen Normandie, France
Ines Lynce	Ténico Lisboa, Portugal
Arnaud Malapert	Université Nice Sophia Antipolis, France
Ciaran McCreesh	University of Glasgow, UK
Kuldeep S. Meel	National University of Singapore
Claude Michel	Université Nice Sophia Antipolis, France
Ian Miguel	University of St. Andrews, UK
Peter Nightingale	University of St. Andrews, UK
Barry O’Sullivan	University College Cork, Ireland
Justin Pearson	Uppsala Universitet, Sweden
Laurent Perron	Google France
Thierry Petit	Worcester Polytechnic Institute, USA
Patrick Prosser	University of Glasgow, UK
Claude-Guy Quimper	Université Laval, Canada
Jean-Charles Régim	Université Nice-Sophia Antipolis, France
Andrea Rendl	Satalia, UK
Emma Rollon	Universitat Politècnica de Catalunya, Spain
Francesca Rossi	Università di Padova, Italy
Pierre Schaus	Université catholique de Louvain, Belgium
Thomas Schiex	INRA, France
Christian Schulte	KTH Royal Institute of Technology, Sweden
Paul Shaw	IBM France
Mohamed Siala	Insight Centre for Data Analytics, Ireland
Helmut Simonis	Insight Centre for Data Analytics, Ireland
Christine Solnon	INSA, France
Peter J. Stuckey	University of Melbourne, Australia
Guido Tack	Monash University, Australia
Michael Trick	Carnegie Mellon University, USA
Christel Vrain	Université de Orléans, France
Mohamed Wahbi	Insight Centre for Data Analytics, Ireland
William Yeoh	Washington University in St. Louis, USA
Alessandro Zanarini	Université de Montréal, Canada
Roie Zivan	Ben Gurion University of the Negev, Israel

Track Program Committees

Applications

Carlos Ansótegui	Universitat de Lleida, Spain
David Bergman	University of Connecticut, USA
André Ciré	University of Toronto, Canada

Willem-Jan Van Hove	Carnegie Mellon University, USA
Serdar Kadioğlu	Fidelity Investments, USA
George Katsirelos	INRA, France
Yuri Malitsky	J. P. Morgan, USA
Louis-Martin Rousseau	École Polytechnique de Montréal, Canada
Kevin Tierney	Universität Paderborn, Germany
Petr Vilim	IBM Czech Republic
Markus Wagner	University of Adelaide, Australia

CP and Data Science

Patrick De Causmaecker	Katholieke Universiteit Leuven, Belgium
Georgiana Ifrim	University College Dublin, Ireland
Kristian Kersting	TU Darmstadt, Germany
Lars Kotthoff	University of Wyoming, USA
Nadjib Lazaar	Université de Montpellier, France
Michela Milano	Università di Bologna, Italy
Pascal Van Hentenryck	University of Michigan, USA
Christel Vrain	Université de Orléans, France
Yingqian Zhang	Eindhoven University of Technology, The Netherlands

CP and Music

Gerard Assayag	IRCAM, France
Elaine Chew	Queen Mary University of London, UK
Tim Dwyer	Monash University, Australia
Mathieu Giraud	CNRS, CRISAL and Université de Lille, France
Dorien Herremans	Singapore University of Technology and Design
Camilo Rueda	Pontificia Universidad Javeriana–Cali, Colombia

CP and Operations Research

Serdar Kadioğlu	Fidelity Investments, USA
Jimmy Lee	Chinese University of Hong Kong, SAR China
Laurent Perron	Google France
Dominico Salvagnin	Università di Padova, Italy
Petr Vilim	IBM, Czech Republic

CP, Optimization, and Power System Management

Bruno François	École Centrale de Lille, France
Nandha Kandaswamy	Singapore University of Technology and Design
Seshadri Kumar	IIT Hyderabad, India
Rémy Rigo-Mariani	Cambridge Centre for Advanced Research and Education in Singapore
P. S. V. Nataraj	IIT Bombay, India

Pascal Van Hentenryck University of Michigan, USA
 Ahmed Zidna Université de Lorraine, France

Multiagent and Parallel CP

Roberto Amadini University of Melbourne, Australia
 Filippo Bistaffa IIIA-CSIC, Spain
 Agostino Dovier Università di Udine, Italy
 Andrea Formisano Università di Perugia, Italy
 Tal Grinshpoun Ariel University, Israel
 T. K. Satish Kumar University of Southern California, USA
 Tiep Le New Mexico State University, USA
 Amnon Meisels Ben Gurion University of the Negev, Israel
 Gauthier Picard MINES Saint-Etienne, France
 Enrico Pontelli New Mexico State University, USA
 Mohamed Wahbi University College Cork, Ireland
 Makoto Yokoo Kyushu University, Japan
 Roie Zivan Ben Gurion University of the Negev, Israel

Testing and Verification

Sébastien Bardin CEA LIST, France
 Catherine Dubois ENSIIE-Samovar, France
 Vijay Ganesh University of Waterloo, Canada
 Matthias Heizmann Universität Freiburg, Germany
 Roberto Castaneda Lozano SICS, Sweden
 Mehdi Maamar CRIL Lens, France
 Marie Pelleau Université Cote d'Azur, France
 Pascal Van Hentenryck University of Michigan, USA
 Lebbah Yahia Université d'Oran 1, Algeria

Additional Reviewers

Özgür Akgün	Vitor Hama	Bertrand Neveu
Ekaterina Arafailova	Hassan Hijazi	Alexandre Papadopoulos
Arthur Bit-Monnot	Alexey Ignatiev	Alberto Policriti
Guillaume Burel	Mikolas Janota	Badran Raddaoui
Sara Ceschia	Christopher Jefferson	Philippe Refalo
Supratik Chakraborty	Amina Kemmar	Lakhdar Sais
Eldan Cohen	Javier Larrosa	Domenico Salvagnin
Yves Crama	Alexandre Lemos	Vaskar Sarkar
Nguyen Dang	Kevin Leo	Joe Scott
Alban Derrien	Olivier Lhomme	Claudio Sole
Daniel Dietsch	Tong Liu	James Trimble
Daniel J. Fremont	Samba Ndojh Ndiaye	Sicco Verwer
Alexandre Goldsztejn	Saeed Nejati	Hong Xu

Local Organizing Committee

Yazid Boumarafi

Zied Bouraoui

Frédéric Boussemart

Guillaume Cavory

Gael Glorian

Fred Hemery

Yacine Izza

Jerry Lonlac

Mehdi Maamar

Valentin Montmirail

Sylvain Merchez

Anastasia Paparrizou

Cédric Piette

Nicolas Szczepanski

Sébastien Tabary

Hélène Verhaeghe

Hugues Watez

Abstracts of Invited Talks

Potential Applications of CP in Industrial Scheduling

Srinivas Bollapragada

General Electric Global Research Center, USA
bollapragada@research.ge.com

Abstract. Scheduling and planning algorithms have the potential to realize significant gains in key industrial sectors such as rail, aviation, power, oil & gas, and healthcare. Improving system level efficiencies even by one percent can save billions of dollars per year in each of these sectors. For example, increasing the average speed of trains by one mile per hour saves the rail industry \$2.5 billion per year. This talk will describe some of our optimization algorithms based industrial applications that saved hundreds of millions of dollars for our customers.

Towards the Holy Grail in Machine Learning

James Cussens

University of York, UK
james.cussens@york.ac.uk

Abstract. The holy grail in machine learning—like that in CP—is that the user merely states the (machine learning) problem and the “system” solves it for them. In the Bayesian approach the user would state what they know as a prior distribution and then a posterior distribution is “learned” by conditioning on the observed data. Point estimates, expectations, predicted values and so on can then be extracted from this posterior.

The reality of machine learning is rather different (witness “gradient descent by grad student” in deep learning!) but progress towards this holy grail is happening right now with the development of probabilistic programming languages like stan. I will argue that the CP community has a contribution to make here. In particular, where the discrete structure of probabilistic model has to be learned (rather than just the continuous parameters of a given model) CP has much to offer. Constraints are also the natural choice when we wish to provide the user with a flexible and expressive language in which to declare any domain knowledge. I will use a number of examples of how CP is already being used in machine learning, including (but not restricted to) my own work on using integer programming to learn the structure of Bayesian networks.

Constraints at the Heart of Classical Planning

Malte Helmert

University of Basel, Switzerland
malte.helmert@unibas.ch

Abstract. The last two decades have seen significant advances in domain-independent planning. Besides improved scalability through better planning algorithms, several breakthroughs have been made in the theoretical understanding of classical planning heuristics. This talk discusses the critical role that constraints play in the modern theory of classical planning heuristics and presents the new opportunities and challenges brought about by a constraint-based view of classical planning.

Contents

Main Technical Track

Automatic Discovery and Exploitation of Promising Subproblems for Tabulation.	3
<i>Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, Peter Nightingale, and András Z. Salamon</i>	
Propagating Regular Membership with Dashed Strings	13
<i>Roberto Amadini, Graeme Gange, and Peter J. Stuckey</i>	
A Constraint-Based Encoding for Domain-Independent Temporal Planning. . . .	30
<i>Arthur Bit-Monnot</i>	
Decremental Consistency Checking of Temporal Constraints: Algorithms for the Point Algebra and the ORD-Horn Class	47
<i>Massimo Bono and Alfonso Emilio Gerevini</i>	
Domain Reduction for Valued Constraints by Generalising Methods from CSP.	64
<i>Martin C. Cooper, Wafa Jguirim, and David A. Cohen</i>	
Solver-Independent Large Neighbourhood Search	81
<i>Jip J. Dekker, Maria Garcia de la Banda, Andreas Schutt, Peter J. Stuckey, and Guido Tack</i>	
Solution-Based Phase Saving for CP: A Value-Selection Heuristic to Simulate Local Search Behavior in Complete Solvers	99
<i>Emir Demirović, Geoffrey Chu, and Peter J. Stuckey</i>	
An SMT Approach to Fractional Hypertree Width.	109
<i>Johannes K. Fichte, Markus Hecher, Neha Lodha, and Stefan Szeider</i>	
On the Non-degeneracy of Unsatisfiability Proof Graphs Produced by SAT Solvers	128
<i>Rohan Fossé and Laurent Simon</i>	
Sequential Precede Chain for Value Symmetry Elimination	144
<i>Graeme Gange and Peter J. Stuckey</i>	
An Incremental SAT-Based Approach to Reason Efficiently on Qualitative Constraint Networks	160
<i>Gael Glorian, Jean-Marie Lagniez, Valentin Montmirail, and Michael Sioutis</i>	

Clause Learning and New Bounds for Graph Coloring.	179
<i>Emmanuel Hebrard and George Katsirelos</i>	
Portfolio-Based Algorithm Selection for Circuit QBFs	195
<i>Holger H. Hoos, Tomáš Peitl, Friedrich Slivovsky, and Stefan Szeider</i>	
Making Compact-Table Compact	210
<i>Linnea Ingmar and Christian Schulte</i>	
Approximation Strategies for Incomplete MaxSAT	219
<i>Saurabh Joshi, Prateek Kumar, Ruben Martins, and Sukrut Rao</i>	
A Novel Graph-Based Heuristic Approach for Solving Sport Scheduling Problem.	229
<i>Meriem Khelifa, Dalila Boughaci, and Esma Aïmeur</i>	
Augmenting Stream Constraint Programming with Eventuality Conditions . . .	242
<i>Jasper C. H. Lee, Jimmy H. M. Lee, and Allen Z. Zhong</i>	
A Complete Tolerant Algebraic Side-Channel Attack for AES with CP	259
<i>Fanghui Liu, Waldemar Cruz, and Laurent Michel</i>	
Evaluating QBF Solvers: Quantifier Alternations Matter.	276
<i>Florian Lonsing and Uwe Egly</i>	
Quantified Valued Constraint Satisfaction Problem	295
<i>Florent Madelaine and Stéphane Secouard</i>	
MLIC: A MaxSAT-Based Framework for Learning Interpretable Classification Rules.	312
<i>Dmitry Malioutov and Kuldeep S. Meel</i>	
Objective as a Feature for Robust Search Strategies.	328
<i>Anthony Palmieri and Guillaume Perez</i>	
PW-CT: Extending Compact-Table to Enforce Pairwise Consistency on Table Constraints	345
<i>Anthony Schneider and Berthe Y. Choueiry</i>	
Automatic Generation and Selection of Streamlined Constraint Models via Monte Carlo Search on a Model Lattice	362
<i>Patrick Spracklen, Özgür Akgün, and Ian Miguel</i>	
Efficient Methods for Constraint Acquisition	373
<i>Dimosthenis C. Tsouros, Kostas Stergiou, and Panagiotis G. Sarigiannidis</i>	
A Circuit Constraint for Multiple Tours Problems	389
<i>Philippe Vismara and Nicolas Briot</i>	

Towards Semi-Automatic Learning-Based Model Transformation 403
Kiana Zeighami, Kevin Leo, Guido Tack, and Maria Garcia de la Banda

Finding Solutions by Finding Inconsistencies 420
Ghiles Ziat, Marie Pelleau, Charlotte Truchet, and Antoine Miné

The Effect of Structural Measures and Merges on SAT Solver Performance . . . 436
*Edward Zulkoski, Ruben Martins, Christoph M. Wintersteiger,
 Jia Hui Liang, Krzysztof Czarnecki, and Vijay Ganesh*

Learning-Sensitive Backdoors with Restarts 453
*Edward Zulkoski, Ruben Martins, Christoph M. Wintersteiger,
 Robert Robere, Jia Hui Liang, Krzysztof Czarnecki, and Vijay Ganesh*

Applications Track

Process Plant Layout Optimization: Equipment Allocation 473
*Gleb Belov, Tobias Czauderna, Maria Garcia de la Banda,
 Matthias Klapperstueck, Ilankaikone Senthoran, Mitch Smith,
 Michael Wybrow, and Mark Wallace*

A Constraint Programming Approach for Solving Patient
 Transportation Problems. 490
*Quentin Cappart, Charles Thomas, Pierre Schaus,
 and Louis-Martin Rousseau*

Unifying Reserve Design Strategies with Graph Theory
 and Constraint Programming 507
Dimitri Justeau-Allaire, Philippe Birnbaum, and Xavier Lorca

Self-configuring Cost-Sensitive Hierarchical Clustering with Recourse 524
Carlos Ansotegui, Meinolf Sellmann, and Kevin Tierney

CP and Data Science Track

User’s Constraints in Itemset Mining 537
Christian Bessiere, Nadjib Lazaar, and Mehdi Maamar

On Maximal Frequent Itemsets Mining with Constraints 554
*Said Jabbour, Fatima Ezzahra Mana, Imen Ouled Dlala,
 Badran Raddaoui, and Lakhdar Sais*

A Parallel SAT-Based Framework for Closed Frequent Itemsets Mining 570
Imen Ouled Dlala, Said Jabbour, Badran Raddaoui, and Lakhdar Sais

Towards Effective Deep Learning for Constraint Satisfaction Problems 588
Hong Xu, Sven Koenig, and T. K. Satish Kumar

CP and Music Track

Extending the Capacity of $1/f$ Noise Generation 601
Guillaume Perez, Brendan Rappazzo, and Carla Gomes

CP and Operations Research Track

Securely and Automatically Deploying Micro-services in an Hybrid Cloud Infrastructure. 613
Waldemar Cruz, Fanghui Liu, and Laurent Michel

Improving Energetic Propagations for Cumulative Scheduling. 629
Alexander Tesch

CP, Optimization, and Power System Management Track

A Fast and Scalable Algorithm for Scheduling Large Numbers of Devices Under Real-Time Pricing 649
Shan He, Mark Wallace, Graeme Gange, Ariel Liebman, and Campbell Wilson

Multiagent and Parallel CP Track

Balancing Asymmetry in Max-sum Using Split Constraint Factor Graphs. . . . 669
Liel Cohen and Roie Zivan

A Large Neighboring Search Schema for Multi-agent Optimization 688
Khoi D. Hoang, Ferdinando Fioretto, William Yeoh, Enrico Pontelli, and Roie Zivan

Distributed Constrained Search by Selfish Agents for Efficient Equilibria. . . . 707
Vadim Levit and Amnon Meisels

Testing and Verification Track

Metamorphic Testing of Constraint Solvers 727
Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale

Algebraic Fault Attack on SHA Hash Functions Using Programmatic SAT Solvers. 737
Saeed Nejati, Jan Horáček, Catherine Gebotys, and Vijay Ganesh

Correction To: PW-CT: Extending Compact-Table to Enforce Pairwise Consistency on Table Constraints E1
Anthony Schneider and Berthe Y. Choueiry

Correction to: MLIC: A MaxSAT-Based Framework for Learning
 Interpretable Classification Rules. C1
Dmitry Malioutov and Kuldeep S. Meel

Abstracts

Encoding Cardinality Constraints Using Multiway Merge
 Selection Networks 757
Michał Karpiński and Marek Piotrów

Not All FPRASs Are Equal: Demystifying FPRASs for DNF-Counting
 (Extended Abstract). 759
Kuldeep S. Meel, Aditya A. Shrotri, and Moshe Y. Vardi

Constraint Games for Stable and Optimal Allocation of Demands in SDN . . . 760
Anthony Palmieri, Arnaud Lallouet, and Luc Pons

Author Index 763

Main Technical Track



Automatic Discovery and Exploitation of Promising Subproblems for Tabulation

Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel,
Peter Nightingale^(✉), and András Z. Salamon

School of Computer Science, University of St Andrews, St Andrews, UK
{ozgur.akgun,ian.gent,caj21,ijm,pwn1,Andras.Salamon}@st-andrews.ac.uk

Abstract. The performance of a constraint model can often be improved by converting a subproblem into a single table constraint. In this paper we study heuristics for identifying promising subproblems. We propose a small set of heuristics to identify common cases such as expressions that will propagate weakly. The process of discovering promising subproblems and tabulating them is entirely automated in the tool SAVILE ROW. A cache is implemented to avoid tabulating equivalent subproblems many times. We give a simple algorithm to generate table constraints directly from a constraint expression in SAVILE ROW. We demonstrate good performance on the benchmark problems used in earlier work on tabulation, and also for several new problem classes.

1 Introduction

In order to improve the performance of a constraint model, a common step is to reformulate the expression of a subset of the problem constraints, either to strengthen the inferences made during search by the constraint solver by increasing constraint propagation, or to maintain the level of propagation while reducing the cost of propagating the constraints. One such method is *tabulation*: to aggregate a set of constraint expressions into a single table constraint [1–3], which explicitly lists the allowed tuples of values for the decision variables involved. This allows us to exploit efficient table constraint propagators that enforce generalised arc consistency [4], typically a stronger level of inference than is achieved for a logically equivalent collection of separate constraints. Successful examples of this approach where the reformulation has been performed by hand include Black Hole patience [5] and Steel Mill Slab Design [6].

Recently, Dekker et al. [7] presented a method for the partial automation of tabulation. In their approach a predicate (a Boolean function) expressed in the MiniZinc language [8] may be annotated to be converted automatically into a table constraint. In the same vein, the IBM ILOG CPLEX Optimization Studio software supports **strong** annotations to indicate that the solver should find a precomputed table constraint corresponding to a specified set of variables; the resulting table constraint is then added to the model as an implied constraint [9]. The Propia library performed a similar step for an annotated goal

in ECLiPSe [10]. In all of these approaches, the crucial first step of identifying promising parts of a given model for tabulation is left to the human modeller.

In this work we present an entirely automatic tabulation method situated in the automated constraint modelling tool SAVILE ROW [11–13]. A set of heuristics is employed to identify in an ESSENCE PRIME [14] model candidate sets of constraints for tabulation, which are then tabulated automatically. In order to demonstrate the effectiveness of our approach, we first examine the same four case studies used by Dekker et al. to demonstrate the utility of tabulation from manual model annotations. We show that our automated approach can identify the same opportunities to improve the model by tabulation. We also study four additional problem classes that show that our tabulation heuristics remain effective on a wider range of problems.

Preliminaries. A *constraint satisfaction problem* (CSP) is defined as a set of variables X , a function that maps each variable to its domain, $D : X \rightarrow 2^{\mathbb{Z}}$ where each domain is a finite set, and a set of constraints C . A *constraint* $c \in C$ is a relation over a subset of the variables X . The *scope* of a constraint c , named $\text{scope}(c)$, is the sequence of variables that c constrains. The scope has an order and may contain a decision variable more than once. During a systematic search for a solution to a CSP, values are progressively removed from the domains D . A *literal* is a variable-value pair (written $x \mapsto v$). A literal $x \mapsto v$ is *valid* iff $v \in D(x)$. For a constraint c we use r for the size of $\text{scope}(c)$. A constraint c is Generalised Arc Consistent (GAC) if and only if there exists a support containing every valid literal of every variable in $\text{scope}(c)$. GAC is established by identifying all literals $x \mapsto v$ for which no support exists and removing v from the domain of x . A *support* of constraint c is a set of literals containing exactly one literal for each variable in $\text{scope}(c)$, such that c is satisfied by the assignment represented by these literals. In a table constraint the set of supports are explicitly listed.

2 Identifying Promising Subproblems for Tabulation

We have designed four heuristics to identify cases where expert modellers might experiment with tabulation to improve the performance of a CP solver. The heuristics and tabulation operate on the abstract syntax tree (AST) of a model, once all problem class parameters have been substituted in, all quantifiers and comprehensions have been unrolled, and matrices of variables have been replaced by individual variables. Tabulation is applied before common subexpression elimination and general flattening. Details of the tailoring process of SAVILE ROW are given elsewhere [13]. Our heuristics are:

Duplicate Variables identifies a constraint containing at most 10 distinct variables, with at least one variable occurring more than once in the scope.

Large AST identifies a constraint where the number of nodes in the AST is greater than 5 times the number of distinct decision variables in scope.

Weak Propagation identifies a constraint c_1 that is likely to propagate weakly (i.e. less than GAC), such that there is another constraint c_2 that propagates strongly, with at least one variable in the scope of both c_1 and c_2 .

Identical Scopes identifies sets of two or more constraints whose scopes contain the same set of decision variables.

Each of the four heuristics is based on a simple rationale regarding either propagation strength or propagation speed of the constraint(s). The Duplicate Variables heuristic identifies constraints that are likely to propagate weakly even when the target solver has a strong propagator for the constraint type. In most cases a GAC propagator will enforce GAC only when there are no duplicate variables. For example, enforcing GAC on the Global Cardinality Constraint (GCC) is known to be NP-hard with duplicate variables [15], therefore Régin’s polynomial-time GAC propagator [16] achieves GAC only when there are no duplicate variables. The replacement table constraint will not have duplicate variables in scope and will therefore achieve GAC.

The Large AST heuristic identifies constraints that are not compactly represented in the AST. A typical example would be an element constraint $M[x] = y$ with a large constant matrix M . The rationale behind it is that a table propagator may be more efficient while achieving the same or stronger propagation.

The Weak Propagation heuristic is intended to catch cases where the weak propagation of one constraint is hindering strong propagation of another. For example, suppose we have the constraints $\text{allDifferent}(x_1, x_2, x_3)$ and $x_1 = 10x_4 + x_5$, a GAC propagator is used for allDifferent , and a bound consistency propagator is used for sum equality. Tabulating the sum equality constraint and therefore potentially pruning more values from x_1 may strengthen propagation of the allDifferent onto x_2 and x_3 . To implement the Weak Propagation heuristic we need to define which constraint expressions are expected to propagate strongly. The definition is recursive on the AST representing the expression. Each type of AST node is defined to be either weak, or strong iff all its children are strong. At the leaves of the AST, constants and references to variables are defined to be strong. For example, the allDifferent constraint often has a GAC propagator so it is defined to be strong iff all its children are strong. The constraint $\text{allDifferent}(x_1, x_2, x_3)$ is strong. Sums are defined to be weak because they are often implemented with bound consistency propagators. The constraint $\text{allDifferent}(x_1 - x_2, x_3 - x_4, x_5 - x_6)$ is therefore defined to be weak. Its eventual representation in the CP solver is unlikely to enforce GAC.

Finally we consider the Identical Scopes heuristic. It is well known that multiple constraints on the same scope may not propagate strongly together, even if each constraint individually does propagate strongly. The Identical Scopes heuristic is intended to collect such sets of constraints into a single table constraint that may propagate more strongly and also may be faster.

Each heuristic fires on at least one of the case studies in Sects. 3 and 4. We discuss the expressions that trigger the heuristics, and the benefits of tabulation.

Caching. We use caches to avoid generating identical tables many times for similar constraints. To store or retrieve a table for an expression e , we first place e into a normal form: the expression is simplified and placed into negation normal form [13, Sec. 3.3]. Then all associative and commutative k -ary expressions (such as sums) and commutative binary operators (e.g. $=$) within e are

sorted. Alphabetical order is used because it will group together references to the same matrix (all else being equal) and place references to different matrices in a consistent order regardless of the indices. The expression is traversed in left-first order to collect a sequence of decision variables (without duplication), and the variables in the sequence are then renamed to a canonical sequence of names to create e' . Thus the actual variable names in e do not affect e' , only their relative positions. e' and the variable domains together are used as a key to store and retrieve tables in the caches. We have a persistent cache stored on disk containing tables, and two memory caches: the first contains tables, and the second stores cases where tabulation failed because the tabulator reached one of its limits. When an expression is identified by a heuristic to tabulate, we look it up in the memory caches then the disk cache. In our experiments we disabled the disk cache because it would cause timings to change depending on the order of processes.

Generating Tables. Given a boolean expression e to tabulate, we first sort e and collect a list of its variables (without duplicates) in the order used by the cache. This ensures that the columns of the table are in the right order for it to be stored in the cache. A table is generated by depth-first search with a static variable ordering and d -way branching. At each node the expression is simplified [13]; if it evaluates to *false* then the search backtracks. At each leaf that evaluates to *true*, we store the assignment as a tuple in the table. In some cases a heuristic will identify a constraint that is simply too large to be tabulated. To deal with these cases we limit the depth-first search to generate at most 10,000 tuples, and to fail and backtrack at most 100,000 times.

3 Experimental Evaluation: Baseline

Tabulation (whether performed manually or with tool support) is a well-established technique. Therefore, instead of examining whether tabulation is effective, we consider whether we can automatically identify subproblems that can be usefully tabulated. Our first four case studies are the four problems presented by Dekker et al. [7]. In each case we show that our heuristics can automatically identify the same subproblems that Dekker et al. identified by hand, to then yield comparable performance improvements.

Black Hole. Black Hole is a patience card game where cards are played one by one into the ‘black hole’ from seventeen face-up fans of three cards. All cards can be seen at all times. A card may be played into the ‘black hole’ if it is adjacent in rank to the previous card. Black Hole was modelled for a variety of solvers by Gent et al. [5] and a table constraint was used in the CP model. We use the simplest and most declarative model of Dekker et al. [7] where two variables \mathbf{a} and \mathbf{b} represent adjacent cards iff $|\mathbf{a}-\mathbf{b}| \% 13 \text{ in } \{1, 12\}$. The adjacency constraint triggers the Weak Propagation heuristic because it overlaps with an `allDifferent` constraint. No other constraint triggers any heuristic, so our set of constraints to tabulate exactly match those identified by hand, first by Gent et al. and later by Dekker et al.

Block Party Metacube Problem. The Block Party Metacube Problem is a puzzle in which eight small cubes are arranged into a larger *metacube*, such that the visible faces on each of the six sides of the metacube form a “party”. Each small cube has a symbol at each corner of each of its faces (24 symbols per cube in total), and each symbol has three attributes, with each attribute in turn taking one of four values. To form a valid party (the *party constraint*), the four small cubes forming a visible face of the large cube must be arranged so that the four symbols in the middle of the visible face are either all different, or all the same, for each of the three attributes. We use the model and instances of Dekker et al. [7].

Dekker et al. tabulated a channelling constraint linking cubes and icons. The Duplicate Variables heuristic identifies the same channelling constraint and it is successfully tabulated. The party constraint as a whole triggers the Identical Scopes heuristic, and the Duplicate Variables heuristic is triggered by each of the four conjuncts of the party constraint, however these constraints are not tabulated because the tabulator reaches a limit. Overall our system tabulates exactly the same set of constraints as Dekker et al.

Handball Tournament Scheduling. Handball Tournament Scheduling requires scheduling matches of a tournament, while respecting the rules governing the tournament, and minimising a cost function related to the availability of venues. We use the simplified 7+7 team model and 20 instances (all of the same size) used by Dekker et al. [7] with a standard decomposition of the *regular* constraint because SAVILE ROW and Minion do not currently implement *regular*.

Dekker et al. experimented with tabulating two types of subproblem, the second of which provided a significant performance improvement. The second type of subproblem is a part of the objective function that calculates the cost of one row of the schedule. The Large AST heuristic triggers for this type of constraint, however one of the limits described in Sect. 2 prevents these constraints being tabulated. It seems that fixed limits may be too coarse, and a more sophisticated cost-benefit calculation may be required. The Large AST heuristic also triggers for a small number of element constraints containing constant matrices. Tabulation creates a unary table which is absorbed into the variable’s domain.

JP Encoding Problem. The JP Encoding problem was introduced in the MiniZinc Challenge 2014. In brief, the problem is to find the most likely encoding of each byte of a stream of Japanese text where multiple encodings may be mixed. The encodings considered are ASCII, EUC-JP, SJIS, UTF-8 or unknown (with a large penalty). Once again our model closely follows that of Dekker et al. [7]. We use all 10 instances in the MiniZinc benchmark repository. The instances are from 100 to 1900 bytes in length. Each byte has four variables: the encoding, a ‘byte status’ variable that combines the encoding with the byte’s position within a multibyte character, a ‘char start’ variable indicating whether the byte begins a new multibyte character, and the score which contributes to the objective.

Dekker et al. tabulate three subproblems. The first connects two adjacent status variables, and the Identical Scopes heuristic triggers on this. The second links status, encoding, and char start, and we found that the Identical

Scopes heuristic separately links status to encoding, and status to char start. The encoding and char start variables are both functionally defined by status so no propagation is lost with two binary table constraints compared to one ternary table. Thirdly Dekker et al. tabulate the constraint linking the score to the encoding. The Duplicate Variables heuristic triggers on this. In summary, the heuristics identify almost the same set of constraints to tabulate as Dekker et al. did manually, and all identified constraints are successfully tabulated.

4 Experimental Evaluation: New Case Studies

In this section we present four case studies that were not featured in Dekker et al. [7]. In each case we briefly describe the model and discuss the expressions that trigger our heuristics. We evaluate tabulation with three CP solvers:

Minion-Static Minion 1.8 [17], ascending value and static variable orderings.

Minion-Conflict Same as the above with Conflict variable ordering [18].

Chuffed Current version of the learning CP solver Chuffed [19] with free search.

Each reported time is the median of five runs on a 64-core AMD Opteron 6376 (32 processes in parallel, 6 hour time limit). Times include the time taken by SAVILE ROW to tailor the instance and (if activated) to tabulate. Software, models and parameter files for the experiment are available online [20], with some additional analysis of experimental results. The results are plotted in Fig. 1.

Sports Scheduling Completion. The Sports Scheduling problem is to construct a schedule of $n(n-1)/2$ games among n teams where each team plays every other team once with some other constraints. In Sports Scheduling Completion we start with a partial schedule. 10 instances were generated with $n = 12$ and 10 slots assigned uniformly at random. Trivially unsatisfiable instances were excluded. Each game between a pair of teams is represented as a pair of variables a and b and also a single variable c , with the channelling constraint $\mathbf{n}*(a-1)+b=c$. The Weak Propagation heuristic identifies the channelling constraint, and tabulating it proves to be highly beneficial for the two Minion configurations. With Chuffed the picture is mixed. Some instances are slowed by tabulation, particularly the easiest four, while some of the more difficult instances benefit from it. Van Hentenryck et al. manually tabulated the same constraint in their OPL model of Sports Scheduling [21].

Langford's Problem. Langford's problem (CSPLib problem 24 [22]) with parameters n and k is to find a sequence of length nk which contains k copies of each number in the set $\{1, \dots, n\}$. The sequence must satisfy the constraint that if the first occurrence of x is at position p , then the other occurrences appear at $p + (x+1)i$, for $i \in \{1, \dots, k-1\}$. We model Langfords as an $n \times k$ 2D matrix P , where row i represents the positions of the k occurrences of i . The constraints are $P[i, j] = P[i, j-1] + i + 1$ and all the positions $P[i, j]$ are different. We also break the symmetry that the entire sequence can be reversed by requiring $(P[1, 1] - 1) \leq (nk - P[1, k])$. We use all 80 instances

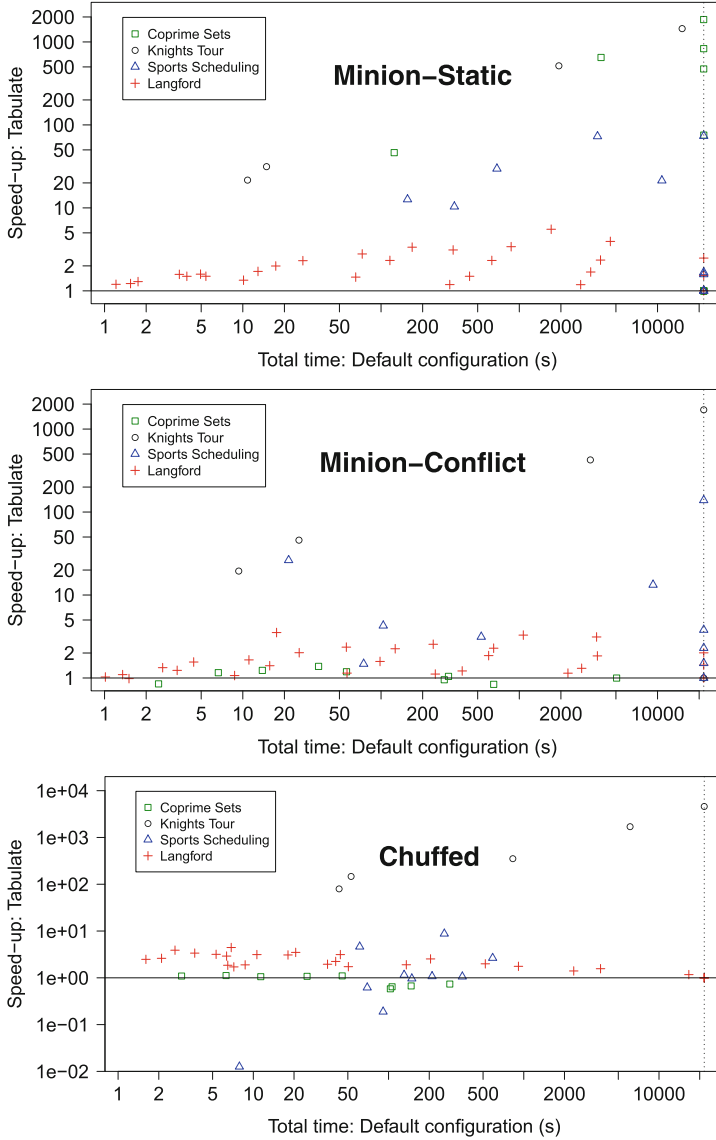


Fig. 1. Tabulate vs Default, total time with Minion solver and static variable ordering (top), Minion solver and Conflict variable ordering (middle), and Chuffed solver with free search (bottom). The x -axis indicates time taken by the default configuration (including both SAVILE ROW and the solver). The y -axis indicates the speed-up obtained by tabulation. Instances that time out are reported as if they completed in 6 hours. The dotted line indicates the time limit of 6 hours; points appearing on the line timed out with the default configuration.

where $n \in \{2 \dots 17\}$ and $k \in \{2 \dots 6\}$. The Weak Propagation heuristic triggers on the $P[i, j] = P[i, j - 1] + i + 1$ constraints (because they overlap with the global allDifferent). Tabulation of these constraints improves propagation and results in improvements for all three solvers.

Coprime Sets. Erdos and Sárközy [23] studied a range of problems involving coprime sets. A pair of numbers a and b are coprime if there is no integer $n > 1$ which is a factor of both a and b . The Coprime Sets problem of size k is to find the smallest m such that there is a set of k numbers in $\{m/2 \dots m\}$ that are pairwise coprime. In our model the set is represented as a sequence of integer variables. Each pair of variables a and b has a set of coprime constraints: $\forall d \in \{2 \dots m\} (a \not\equiv 0 \pmod{d}) \vee (b \not\equiv 0 \pmod{d})$. Adjacent variables are ordered to break symmetry. We use the instances $k \in \{8 \dots 16\}$. The Identical Scopes heuristic triggers on the coprime constraints (and any symmetry breaking constraint) for each pair of variables. All the original constraints are tabulated.

Static variable ordering follows the sequence from smallest to largest number, so would appear to be a natural choice. However, Minion-Static performs poorly compared to the other two solvers. In this case, tabulation makes the model more robust to the poor variable ordering, speeding it up by over 1000 times in some cases. Tabulation provides no benefit for the other two solvers that already solve the instances relatively well.

Knight’s Tour Problem. The Knight’s Tour Problem on an $n \times n$ chessboard is to visit every square of the board exactly once while making only knight’s moves. We use a model where the location of the knight is encoded as a single integer $(nx + y)$, we start at location $(0,0)$ and search for a sequence of n^2 distinct locations. We use instances $n \in \{6 \dots 10\}$. The knight’s move constraint contains two location variables and uses integer division and modulo to obtain the x and y coordinates. The coordinates are used multiple times in the expression. Identical common subexpression elimination (CSE) substantially improves the model by adding auxiliary variables for the x and y coordinates among others. The default configuration includes identical CSE. The knight’s move constraint triggers the Duplicate Variables, Large AST and Weak Propagation heuristics. Tabulation produces a quite different model with no auxiliary variables, much stronger propagation and far better performance with all three solvers.

5 Conclusions

In this paper we have demonstrated that a small set of heuristics can successfully and automatically identify promising subproblems in a constraint model for tabulation, and that these opportunities can be effectively exploited through an automated tabulation method incorporated into the automated constraint modelling system SAVILE ROW. Our heuristics identify the same tabulation opportunities as recent work by Dekker et al. using manual annotations of a MiniZinc model [7]. In addition we have presented four new case studies demonstrating the efficacy of our heuristics and automated tabulation.

Acknowledgements. We thank EPSRC for grants EP/P015638/1 and EP/P-026842/1. Dr Jefferson holds a Royal Society University Research Fellowship.

References

1. Mohr, R., Masini, G.: Good old discrete relaxation. In: Proceedings of ECAI 1988, pp. 651–656. Pitman Publishing (1988)
2. Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalised arc consistency for extensional constraints. In: Proceedings of AAAI 2007, pp. 191–197. AAAI Press (2007). <http://www.aaai.org/Papers/AAAI/2007/AAAI07-029.pdf>
3. Lecoutre, C.: STR2: optimized simple tabular reduction for table constraints. *Constraints* **16**(4), 341–371 (2011). <https://doi.org/10.1007/s10601-011-9107-6>
4. Bessiere, C.: Constraint propagation. In: Handbook of Constraint Programming, pp. 29–83. Elsevier (2006)
5. Gent, I.P., Jefferson, C., Kelsey, T., Lynce, I., Miguel, I., Nightingale, P., Smith, B.M., Tarim, S.A.: Search in the patience game ‘Black Hole’. *AI Communications* **20**(3), 211–226 (2007). <https://content.iospress.com/articles/ai-communications/aic405>
6. Gargani, A., Refalo, P.: An efficient model and strategy for the steel mill slab design problem. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 77–89. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_8
7. Dekker, J.J., Björdal, G., Carlsson, M., Flener, P., Monette, J.N.: Auto-tabling for subproblem presolving in MiniZinc. *Constraints* **22**(4), 512–529 (2017). <https://doi.org/10.1007/s10601-017-9270-5>
8. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard cp modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_38
9. IBM Knowledge Center: The strong constraint (2017). https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.ide.help/OPL_Studio/oplang_quickref/topics/tlr_oplsch_strong.html
10. Le Provost, T., Wallace, M.: Domain independent propagation. In: Proceedings of FGCS: International Conference on Fifth Generation Computer Systems, pp. 1004–1011. IOS Press (1992). <http://www.webmail.eclipseclp.org/reports/corefgcs.pdf>
11. Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I.: Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. In: O’Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 590–605. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_43
12. Nightingale, P., Spracklen, P., Miguel, I.: Automatically improving SAT encoding of constraint problems through common subexpression elimination in savile row. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 330–340. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_23
13. Nightingale, P., Akgün, O., Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in Savile Row. *Artif. Intell.* **251**, 35–61 (2017). <https://doi.org/10.1016/j.artint.2017.07.001>
14. Nightingale, P., Rendl, A.: Essence’ description (2016). [arXiv:1601.02865](https://arxiv.org/abs/1601.02865) [cs.AI]

15. Bessière, C., Hebrard, E., Hnich, B., Walsh, T.: The complexity of reasoning with global constraints. *Constraints* **12**(2), 239–259 (2007). <https://doi.org/10.1007/s10601-006-9007-3>
16. Régim, J.C.: Generalized arc consistency for global cardinality constraint. In: Proceedings of AAAI 1996, pp. 209–215. AAAI Press (1996). <http://www.aaai.org/Papers/AAAI/1996/AAAI96-031.pdf>
17. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: Proceedings of ECAI 2006, pp. 98–102. IOS Press (2006). <http://ebooks.iospress.nl/volumearticle/2658>
18. Lecoutre, C., Sais, L., Tabary, S., Vidal, V.: Last conflict based reasoning. In: Proceedings of ECAI 2006, pp. 133–137. IOS Press (2006). <http://ebooks.iospress.nl/volumearticle/2665>
19. Chu, G., Stuckey, P.J., Schutt, A., Ehlers, T., Gange, G., Francis, K.: Chuffed (2018). <https://github.com/chuffed/chuffed/>
20. Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P., Salamon, A.Z.: Tabulation experimental software and additional results (2018). <https://doi.org/10.5281/zenodo.1290656>, <https://github.com/stacs-cp/cp2018-tabulation>
21. Van Hentenryck, P., Michel, L., Perron, L., Régim, J.-C.: Constraint programming in OPL. In: Nadathur, G. (ed.) PDP 1999. LNCS, vol. 1702, pp. 98–116. Springer, Heidelberg (1999). https://doi.org/10.1007/10704567_6
22. CSPLib: A problem library for constraints (1999). <http://www.csplib.org>
23. Erdos, P., Sárközy, A.: On sets of coprime integers in intervals. *Hardy-Ramanujan J.* **16**, 1–20 (1993). <https://hal.archives-ouvertes.fr/hal-01108688>



Propagating Regular Membership with Dashed Strings

Roberto Amadini¹(✉), Graeme Gange², and Peter J. Stuckey¹

¹ University of Melbourne, Victoria, Australia
roberto.amadini@unimelb.edu.au

² Monash University, Melbourne, Victoria, Australia

Abstract. Using dashed strings is an approach recently introduced in Constraint Programming (CP) to represent the domain of string variables, when solving combinatorial problems with string constraints. One of the most important string constraints is that of regular membership: $\text{REGULAR}(x, R)$ imposes string x to be a member of the regular language defined by automaton R . The REGULAR constraint is useful for specifying complex constraints on fixed length finite sequences, and regularly appears in CP models. Dealing with REGULAR is also desirable in software testing and verification, because regular expressions are often used in modern programming languages for pattern matching. In this paper, we define a REGULAR propagator for dashed string solvers. We show that this propagator, implemented in the G-STRINGS solver, is substantially better than the current state-of-the-art. We also demonstrate that many REGULAR constraints appearing in string solving benchmarks can actually be tackled by dashed strings solvers without explicitly using REGULAR .

1 Introduction

String constraint solving is an emerging topic that bases its motivation in fields like web security and software analysis and verification. Suitable solvers have been introduced over the last years for solving combinatorial problems involving string variables and constraints [1, 9, 12, 14, 18–21].

Recent works [6] introduced the *dashed-string* representation for string variables in constraint programming (CP), and described propagation algorithms for equality and related constraints [5], lexicographic ordering and find/replace [4]. A key advantage of this representation is the ability to efficiently represent strings of uncertain – but possibly very large – length by dividing similarly behaved regions of a partially specified string into a sequence of concatenated *blocks*.

A common element of string constraint problems which has not yet been considered by dashed string solvers is the regular language membership constraint $\text{REGULAR}(x, R)$, whose semantics is $x \in L(R)$ where x is a string variable and $L(R)$ is the regular language denoted by the finite state automaton R .

Constraint programming treatments of REGULAR typically act on a *fixed-length* sequence of integer variables, and require that running the automaton on

this sequence finishes in an accepting state. Sequences of non-fixed (but *bounded*) length are typically padded with a special character to a maximum length in order to use the fixed-length REGULAR propagator. We do not want to adopt this strategy for dashed-strings, as it would totally defeat the advantage of dashed strings that operate effectively on strings whose length bound is large. Instead, we must develop a new propagation algorithm, which operates at the level of *blocks* of characters, rather than individual characters.

In this paper we present such an algorithm and integrate it into G-STRINGS, a constraint programming solver using the dashed-string representation. We evaluate its effectiveness on a range of real-world benchmarks containing regular language constraints, and find that it significantly outperforms existing CP and SMT approaches. We also identify a frequently-occurring subclass of regular languages which can be reformulated as basic string constraints, and evaluate the effect of this substitution.

2 Preliminaries

In this Section we give background notions about dashed strings representation, G-STRINGS solver, automata and regular expressions.

2.1 Dashed Strings

We assume a finite alphabet of symbols Σ . A *string* $w \in \Sigma^*$ is either the empty string ε or of the form cw' where $c \in \Sigma$ is a symbol and $w' \in \Sigma^*$ is a string. Typewriter font is used to denote constant characters $c \in \Sigma$. The *length* $|w|$ of string w is the number of symbols appearing in w . We use array notation to lookup the symbols in a string: $w[i]$ is the i^{th} symbol of string w , with $1 \leq i \leq |w|$.

Let us fix a maximum string length $\lambda \in \mathbb{N}$ and a universe $\mathbb{S} = \bigcup_{i=0}^{\lambda} \Sigma^i$. A *dashed string* of length k is defined by a concatenation of $k > 0$ *blocks* $S_1^{l_1, u_1} S_2^{l_2, u_2} \dots S_k^{l_k, u_k}$, where $S_i \subseteq \Sigma$ and $0 \leq l_i \leq u_i \leq \lambda$ for $i = 1, \dots, k$ and $\sum_{i=1}^k l_i \leq \lambda$. Note that the latter condition does not pose any upper bound to the dashed string length: we might have both $\sum_{i=1}^k l_i \leq \lambda$ and $k > \lambda$.

For each block $S^{l, u}$, we call S the *base* and (l, u) the *cardinality*. The i -th block of a dashed string X is denoted by $X[i]$, and $|X|$ is the length of X . We do not distinguish blocks from dashed strings of unary length and we consider only *normalised* dashed strings, where the adjacent blocks have distinct bases and the *null block* $\emptyset^{0,0}$ occurs only to denote the empty string. In this way we provide an unique representation for each concrete string $w \in \Sigma^*$.

Let $\gamma(S^{l, u}) = \{x \in S^* \mid l \leq |x| \leq u\}$ be the *language denoted* by block $S^{l, u}$. We extend γ to dashed strings: $\gamma(S_1^{l_1, u_1} \dots S_k^{l_k, u_k}) = (\gamma(S_1^{l_1, u_1}) \dots \gamma(S_k^{l_k, u_k})) \cap \mathbb{S}$ (intersection with \mathbb{S} excludes the strings with length greater than λ). A dashed string X is *known* if it denotes a single string: $|\gamma(X)| = 1$. Normalisation entails that each string $w \in \mathbb{S}$ has a unique known dashed string X such that $w = \gamma(X)$.

A block of the form $S^{0, u}$ is called *nullable*, i.e. $\varepsilon \in \gamma(S^{0, u})$. There is no upper bound on the length of a dashed string since an arbitrary number of nullable blocks may occur.



Fig. 1. Graphical representation of $X = \{\mathbf{B}, \mathbf{b}\}^{1,1} \{\mathbf{o}\}^{2,4} \{\mathbf{m}\}^{1,1} \{\mathbf{!}\}^{0,3}$.

The *size* $\|S^{l,u}\|$ of a block is the number of concrete strings it denotes, i.e., $\|S^{l,u}\| = |\gamma(S^{l,u})|$. The size of dashed string $X = S_1^{l_1, u_1} S_2^{l_2, u_2} \dots S_k^{l_k, u_k}$ is an overestimate of $|\gamma(X)|$, given by $\|X\| = \prod_{i=1}^k \|S_i^{l_i, u_i}\|$.

Given dashed strings X and Y we define the relation $X \sqsubseteq Y \Leftrightarrow \gamma(X) \subseteq \gamma(Y)$. Intuitively, \sqsubseteq denotes the relation “*is more precise than*” between dashed strings. Unfortunately, the set of dashed strings does not form a lattice according to \sqsubseteq [6]. For example, there is not a “best” dashed string denoting $\{\mathbf{ab}, \mathbf{ba}\} \subseteq \Sigma^*$. This implies that some workarounds have to be used to preserve the soundness of propagation. For more details, we refer the reader to [5, 6].

Intuitively, we can imagine each block $S_i^{l_i, u_i}$ of $X = S_1^{l_1, u_1} S_2^{l_2, u_2} \dots S_k^{l_k, u_k}$ as a continuous segment of length l_i followed by a dashed segment of length $u_i - l_i$. The continuous segment indicates that exactly l_i characters of S_i *must* occur in each concrete string of $\gamma(X)$, and defines the *mandatory part* $S_i^{l_i, l_i}$. The dashed segment indicates that n characters of S_i , with $0 \leq n \leq u_i - l_i$, *may* occur and defines the *optional part* $S_i^{0, u_i - l_i}$. Consider, for example, the graphical representation of dashed string $X = \{\mathbf{B}, \mathbf{b}\}^{1,1} \{\mathbf{o}\}^{2,4} \{\mathbf{m}\}^{1,1} \{\mathbf{!}\}^{0,3}$ in Fig. 1. Each string of $\gamma(X)$ starts with \mathbf{B} or \mathbf{b} , followed by 2 to 4 \mathbf{o} s, one \mathbf{m} , then 0 to 3 $\mathbf{!}$ s.

2.2 G-Strings Solver

Dashed string solving is implemented in G-STRINGS, an extension of GECODE solver [11]. It implements the domain $D(x)$ of every string variable x with a dashed string, and defines a *propagator* for each string constraint.

Most of the propagators refine the domains of the string variables based on the notion of dashed string *equation*. Equating dashed string X and Y means determining two dashed strings X' and Y' such that: (i) $X' \sqsubseteq X$, $Y' \sqsubseteq Y$; and (ii) $\gamma(X') \cap \gamma(Y') = \gamma(X) \cap \gamma(Y)$. Informally, we can see this problem as a semantic unification where we want to find a refinement of X and Y including all the strings of $\gamma(X) \cap \gamma(Y)$ and removing the most values not belonging to $\gamma(X) \cap \gamma(Y)$ (there may not exist a greatest lower bound for X, Y according to \sqsubseteq). G-STRINGS uses the sweep-based algorithm of [5] to propagate dashed strings equality. For example, string equality $x = y$ is simply propagated by equating the domains $D(x)$ and $D(y)$; the propagator for $z = x \cdot y$ is implemented by equating $D(z)$ and the concatenation of blocks $D(x) \cdot D(y)$.

G-STRINGS implements string (dis-)equality, (half-)reified equality, (iterated) concatenation, string domain, length, reverse, substring selection, global cardinality, channeling with integers, lexicographic ordering, find and replace. Since propagation is in general not complete, G-STRINGS also defines strategies for branching on variables (e.g., the one with smallest domain size or having the domain with the minimum number of blocks) and domain values (by heuristically selecting first a block, and then a character of its base).

2.3 Automata and Regular Expressions

A finite-state automaton (or simply automaton) is a tuple $R = \langle Q, \Sigma, \delta, q_0, F \rangle$ where Σ is the *alphabet*; Q is a finite set of *states* including the *initial state* q_0 and a set F of *accepting states*; and $\delta \subseteq Q \times \Sigma \times Q$ is a set of *transitions*. A transition $(q, c, q') \in \delta$ from state q to q' is also written as $q \rightarrow q'$. A *computation* of length l for string w is a sequence of l transitions $s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_l$ where $(s_{i-1}, w[i], s_i) \in \delta$ for $i = 1, \dots, l$. The string w is *accepted* by automaton R when $|w| = l$, $s_0 = q_0$ and $s_l \in F$. The *language* $L(R)$ of automaton R is the regular language consisting of all the strings of Σ^* accepted by R . If it does not exist an automaton $R' = \langle Q', \Sigma', \delta', q'_0, F' \rangle$ such that $L(R') = L(R)$ and $|Q'| < |Q|$, then R is *minimal*.

An automaton is *deterministic* (DFA) if δ is a (partial) function $Q \times \Sigma \rightarrow Q$. In this case, we can use the notation $\delta(q, c) = q'$ if $(q, c, q') \in \delta$; otherwise, $\delta(q, c) = \perp$ if undefined. A DFA is *trim* if for each $q \in Q$ there exists a computation $q_0 \rightarrow \dots \rightarrow q$ and a computation $q \rightarrow \dots \rightarrow q' \in F$.

If δ is a total function, then the DFA is *complete*. If a DFA $\langle Q, \Sigma, \delta, q_0, F \rangle$ is not complete, we can extend Q to $Q' = Q \cup \{q_\perp\}$, and δ to $\delta' = \delta \cup \{(q, c, q_\perp) \mid q \in Q', c \in \Sigma, \delta(q, c) = \perp\}$ in order to have a complete DFA $\langle Q', \Sigma, \delta', q_0, F \rangle$. Given an automata R , the *complement automaton* \bar{R} is an automata such that $L(\bar{R}) = \Sigma^* - L(R)$. Given a complete DFA $R = \langle Q, \Sigma, \delta, q_0, F \rangle$, we can easily compute $\bar{R} = \langle Q, \Sigma, \delta, q_0, Q - F \rangle$ by complementing the final states.

Given a complete automaton $R = \langle Q, \Sigma, \delta, q_0, F \rangle$, a state $q \in F$ is said *universally accepting* if all computations from q bring to a state $q' \in F$ (i.e., any computation reaching q will be accepted). Dually, a state $q \in Q - F$ is said *universally rejecting* if all computations from q bring to a state $q' \in Q - F$ (i.e., any computation reaching q will be rejected).

Let $acc(R)$ and $rej(R)$ be the set of universally accepting and rejecting states of R respectively. If R is minimal, then $|acc(R)|, |rej(R)| \leq 1$. We can efficiently compute the minimum and the maximum length string accepted by R , $minl(R) = \min\{|w| \mid w \in L(R)\}$ and $maxl(R) = \max\{|w| \mid w \in L(R)\}$ ¹. Note that $maxl(R)$ may be $+\infty$ (if loops occur) but for our purposes can be at most λ (the maximum allowed string length).

An alternative yet equivalent way to denote a regular language is by means of *regular expressions*. We define inductively the set \mathcal{RE} of regular expressions (over alphabet Σ), as well as the language $L(r)$ denoted by each $r \in \mathcal{RE}$, as: (i) $\emptyset \in \mathcal{RE}$, denoting $L(\emptyset) = \emptyset$; (ii) if $c \in \Sigma \cup \{\epsilon\}$, then $c \in \mathcal{RE}$ denoting $L(c) = \{c\}$; (iii) if $r, r' \in \mathcal{RE}$, then $r \cdot r' \in \mathcal{RE}$ denoting $L(r \cdot r') = L(r)L(r')$, and $r|r' \in \mathcal{RE}$ denoting $L(r|r') = L(r) \cup L(r')$; (iv) if $r \in \mathcal{RE}$ then $r^* \in \mathcal{RE}$ denoting $L(r^*) = L(r)^*$; (v) nothing else belongs to \mathcal{RE} .

Given a regular expression $r \in \mathcal{RE}$, we indicate with DFA the function such that $R = \text{DFA}(r)$ is the minimal automaton such that $L(r) = L(R)$. Note that in our case we actually consider the finite language $L(r) \cap \mathbb{S}$, having strings with length smaller or equal to λ .

¹ Note that $maxl(R) \neq \max\{|w| \mid w \in L(R), |w| \leq \lambda\}$, which is less easy to compute.

If $maxl(R) > \lambda$, we set $maxl(R) = \lambda$: this is a correct but not optimal upper bound.

3 Propagating regular on Dashed Strings

The REGULAR constraint arises fairly frequently in constraint programming problems. The usual CP constraint $\text{REGULAR}(x, R)$ constraint takes a fixed description of an automata R , a fixed length sequence x of integer variables, and constrains $x \in L(R)$. This form of REGULAR was introduced in [7] and several propagation algorithms have been developed [10, 16, 17]. These algorithms *unfold* the regular automaton into a *layered graph* – creating a copy of each automaton state for each variable – which is incrementally updated during search; propagation occurs when there is no longer a viable edge for some value k at some level.

For unfolding-based string constraint solvers like [18], these REGULAR propagators may be used directly. Given automaton $\langle Q, \Sigma, q_0, \delta, F \rangle$, if ϵ is the null symbol used to pad strings of length smaller than λ , it is enough to introduce a fresh state q_ϵ , such that $\delta(q_\epsilon, c) = q_\epsilon$, for each $c \in \Sigma, q \in F \cup \{q_\epsilon\}$. But dashed strings solvers represent a much richer sequence variable x , where crucially we do not know the length of various components. While we could use the unfolding approach for dashed strings, this would defeat their main purpose which is to reason about potentially long strings efficiently by means of a *lazy* approach.

In this paper we also consider the *reified* form of the regular constraint, which is rare in CP but frequent in SMT benchmarks derived, e.g. from security analysis and model checking. This is not surprising since we want also to express more complex constraints like $x \notin L(R)$ or **if** $x \in L(R)$ **then** $A(x)$ **else** $B(x)$.

We then implemented the reified constraint $b \Leftrightarrow \text{REGULAR}(x, R)$, where b is a Boolean variable. While in the general case we restrict R to be a complete DFA, if $b = \text{true}$, i.e. we just have the positive constraint $\text{REGULAR}(x, R)$, the propagation algorithms work for any non-deterministic and non-complete automaton.

3.1 Propagation

Let us propagate $b \Leftrightarrow \text{REGULAR}(x, R)$, where $R = \langle Q, \Sigma, q_0, \delta, F \rangle$ is a complete DFA and x is a string variable. We take inspiration from the propagation of the regular global constraint for integer variables.

Before posting the REGULAR constraint itself, we post some bound constraints on the length of x by taking advantage of *minl* and *maxl* functions (see Fig. 2). These constraints may detect early failures or provide additional information. For example, consider $D(x) = \{\mathbf{a}\}^{0,1}\{\mathbf{b}\}^{0,1}$ and $L(R) = \{\mathbf{a}, \mathbf{b}\}$. If $b = \text{true}$, then we get $|x| = 1$; otherwise, nothing can be inferred: $0 \leq |x| \leq 2$.

The reified regular constraint propagator, summarised in Fig. 3, takes the current domain $B = D(b)$ of Boolean variable b , the current domain $X = D(x)$ of string variable x , the automata R , and returns a triple $\langle B', X', s \rangle$, where B' (resp., X') is an updated domain for b (resp., x) and s is a Boolean value which determines if the constraint is *subsumed* (i.e. we cannot propagate further). Although we assume R is complete, the pseudo code is also correct for arbitrary automata if we omit the greyed out parts.


```

function POST-REIFIED-REGULAR( $b, x, R$ )
  if  $D(b) = \{true\}$  then                                     ▷ positive regular constraint
    POST( $minl(R) \leq |x| \leq maxl(R)$ )
    POST( $true \Leftrightarrow REGULAR(x, R)$ )
  else if  $D(b) = \{false\}$  then                               ▷ complemented regular constraint
    POST( $minl(\bar{R}) \leq |x| \leq maxl(\bar{R})$ )
    POST( $true \Leftrightarrow REGULAR(x, \bar{R})$ )
  else                                                         ▷ general form
    POST( $b \Leftrightarrow REGULAR(x, R)$ )

```

Fig. 2. Pre-checks before actually posting $b \Leftrightarrow x \in L(R)$.

```

function PROP-REIFIED-REGULAR( $B, X = S_1^{l_1, u_1}, \dots, S_n^{l_n, u_n}, R = \langle \Sigma, Q, q_0, \delta, F \rangle$ )
   $F_0 \leftarrow \{\{q_0\}\}$ 
  for  $i \in 1, 2, \dots, n$  do                                   ▷ forward pass.
     $F_i \leftarrow REACH-FWD(B, Q, \delta, LAST(F_{i-1}), S_i^{l_i, u_i})$ 
    if  $LAST(F_i) \subseteq rej(R)$  then                               ▷  $x$  surely rejected
      return  $B \cap \{false\}, X, true$ 
    if  $LAST(F_i) \subseteq acc(R)$  then                               ▷  $x$  surely accepted
      return  $B \cap \{true\}, X, true$ 
    if  $LAST(F_n) \subseteq F$  then                                   ▷  $x$  surely accepted
      return  $B \cap \{true\}, X, true$ 
    if  $LAST(F_n) \subseteq Q - F$  then                               ▷  $x$  surely rejected
      return  $B \cap \{false\}, X, true$ 
    if  $B = \{true\}$  then                                       ▷ positive regular constraint
       $E \leftarrow LAST(F_n) \cap F$ 
    else if  $B = \{false\}$  then                                   ▷ complemented regular constraint
       $E \leftarrow LAST(F_n) \cap (Q - F)$ 
    else                                                         ▷ nothing to propagate
      return  $B, X, false$ 
    if  $E = \emptyset$  then                                       ▷  $E$  is the set of feasible ending states.
      return  $\emptyset, \emptyset, true$ 
    for  $i \in n, n-1, \dots, 1$  do                               ▷ backward pass.
       $E, X'_i \leftarrow REACH-BWD(Q, \delta, F_i, E, S_i^{l_i, u_i})$ 
  return  $B, NORM([X'_1, \dots, X'_n]), false$ 

```

Fig. 3. Propagation algorithm for $b \Leftrightarrow x \in L(R)$.

The propagation algorithm essentially works in two steps: (i) a *forward* pass, where we compute a set of reachable states, potentially detecting inconsistency; (ii) a *backward* pass, where only feasible end-states are considered and the domains of the variables are possibly pruned.

The forward pass keeps track of the sets of states F_i reachable by any concrete string in $\gamma(X)$ after consuming block $X[i]$. REACH-FWD returns in particular a sequence $F_i = [Q_{i,0}, \dots, Q_{i,l_i}, Q_{i,l_i+1}]$ of sets of states where, for $j = 0, \dots, l_i$,

$Q_{i,j}$ is the set of states reachable after consuming *exactly* j characters of $X[i]$ (corresponding to the mandatory part of the block), while Q_{i,l_i+1} is the set of states reachable after consuming an arbitrary number $k \in [l_i, u_i]$ of characters of $X[i]$ (corresponding to the optional part of the block). The last set of states $\text{LAST}(F_i)$ (we assume that LAST is a function returning the last element of a sequence) is used to possibly detect when the constraint is subsumed.

After the loop, if all the states of $\text{LAST}(F_n)$ are accepting, then the constraint must hold (i.e., it is subsumed) and we can propagate $b = \text{true}$ (similarly, if they are all rejecting we can propagate $b = \text{false}$). We then calculate the set of accepting final states E , and if b is not fixed we return since no propagation is possible. If E is empty we detect unsatisfiability, and return. Otherwise, we iterate backward over the blocks of X and we use REACH-BWD to compute the sets of states that are both reachable and may lead to an accepting state.

At the end of the function, we return the possibly refined domains for variables b and x . Note that, for the latter, we use the NORM function to make the sequence of blocks $[X'_1, \dots, X'_n]$ a normalised dashed string. For example, $\text{NORM}([\{a, b\}^{0,2}, \emptyset^{0,0}, \{a, b\}^{1,1}]) = \{a, b\}^{1,3}$. In the following we shall explain the forward and backward phases in more details.

```

function REACH-FWD( $B, \delta, Q, Q_F, S^{l,u}$ )
     $\delta_{fwd} \leftarrow \{q \mapsto \{\delta(q, c) \mid c \in S\} \mid q \in Q\}$  ▷ Feasible forward transitions.
     $Q_0 \leftarrow Q_F$ 
    for  $i \in 1, 2, \dots, l$  do ▷ Mandatory region
         $Q_i \leftarrow \bigcup_{q \in Q_{i-1}} \delta_{fwd}(q)$ 
        if  $Q_i = Q_{i-1}$  then ▷ Fixpoint
             $Q_l \leftarrow \dots \leftarrow Q_{i+1} \leftarrow Q_i$ 
            return  $[Q_0, \dots, Q_l, Q_l]$ 
        if  $Q_i \subseteq \text{rej}(R) \vee Q_i \subseteq \text{acc}(R)$  then ▷ Constraint subsumed
            return  $[Q_i]$ 
     $Q_{bfs} \leftarrow \text{QUEUE}(Q_l)$ 
     $\text{dist} \leftarrow \left\{ q \mapsto \begin{cases} l & \text{if } q \in Q_l \\ +\infty & \text{if } q \in Q - Q_l \end{cases} \right\}$ 
    while  $Q_{bfs} \neq []$  do ▷ BFS over optional region.
         $q \leftarrow \text{POP}(Q_{bfs})$ 
         $d \leftarrow \text{dist}[q] + 1$ 
        if  $d \leq u$  then
            for  $q' \in \delta_{fwd}(q)$  where  $\text{dist}[q'] > d$  do
                 $\text{PUSH}(Q_{bfs}, q')$ 
                 $\text{dist}[q'] = d$ 
    return  $[Q_0, \dots, Q_l, \{q \in Q \mid \text{dist}[q] \leq u\}]$ 
    
```

Fig. 4. Forward pass of the algorithm. Returns reachable end-states, plus intermediate states needed for the backward pass.

Forward Pass. The forward pass is implemented by REACH-FWD (see Fig. 4). It computes a sequence of sets of states $[Q_0, Q_1, \dots, Q_l, Q_{l+1}]$ that are reachable after consuming characters in the block $S^{l,u}$. In particular, for $0 \leq i \leq l$, sets Q_i are those after consuming exactly i characters, while Q_{l+1} collects any states possible after consuming a number of characters between l and u inclusive.

The mandatory part is straightforward. If we find a set of states always rejecting (or accepting in the positive case) we can return since the constraint is subsumed. If instead a set of states Q_i is identical to the previous set Q_{i-1} , then we have reached a *fixpoint*: we are finished since $Q_j = Q_{j-1}$ for $j = i, \dots, l$.

The optional part proceeds by breadth first search (BFS) finding new states reachable in at most $u-l$ characters. We store in *dist* dictionary the least distance to reach any state starting from the states in Q_0 . The queue of states Q_{bfs} to expand consists initially of the states of Q_l , i.e. all the states reachable after consuming all the l characters of the mandatory part of $S^{l,u}$.

Note that QUEUE is a function returning a queue containing all the elements of a given set (the order of the element does not matter here). Functions PUSH and POP have the usual semantics. We pop states from Q_{bfs} and if they are less than u distance we push their neighbors onto the queue as long we have found a shorter route to them, updating their distance.

The complexity of REACH-FWD is $O(|\delta| \times (l+1))$, that for a complete DFA corresponds to $O(|Q| \times |\Sigma| \times (l+1))$, since we consider each transition at most once in each iteration of the mandatory region, and at most once in the BFS over the optional region. However, in the case where $b = true$ we could consider a trim DFA δ' with $|\delta'|$ typically far smaller than $|Q| \times |\Sigma|$.

Backward Pass. The backward pass of the algorithm calculates the states which can both reach a final state, and be reached from the start state. The approach of REACH-BWD (see Fig. 5) is analogous to a reversed forward pass, but uses the stored reachability vectors F_i to compute the intersection.

The first step simulates characters in the optional part of the block. It considers all possible ending states Q_E and adds them to a queue. It maintains the least distance to reach each state in *dist*. When it pops a state in $q \in Q_l$ it updates the least possible distance l' required to reach q . If such a distance is at most $u-l$ we collect the characters of the usable transitions into S_{opt} .

If we have reached a state for the first time, we push it onto the BFS queue Q_{bfs} , and update its distance. This creates the optional block with length at least l' and at most $u-l$, with all characters met in S_{opt} .

The remainder is simpler. Given set of states E we could reach after exactly l characters (and can reach a final state) we collect characters c that might reach these states in S_{man} , and the states q' that we could reach this state from, to initialise E for the next iteration. For this step we create a block of unary length with characters in S_{man} . Finally we return the (normalised) dashed string of these $l+1$ blocks.

The complexity of REACH-BWD is similarly $O(|\delta| \times (l+1))$, so the overall worst-case complexity of PROP-REIFIED-REGULAR is $O(|\delta| \times \sum_{i=1}^n l_i)$. This means

```

function REACH-BWD( $\delta, Q, [Q_0, \dots, Q_l, Q_{l+1}], Q_E, S^{l,u}$ )
     $\delta_{bwd} \leftarrow \{q \leftarrow \{(c, q') \mid (q', c, q) \in \delta, c \in S\} \mid q \in Q\}$        $\triangleright$  Backward transitions
     $S_{opt} \leftarrow \emptyset$ 
     $Q_{bfs} \leftarrow \text{QUEUE}(Q_E)$ 
     $l' \leftarrow +\infty$ 
     $dist \leftarrow \left\{ q \mapsto \begin{array}{ll} 0 & \text{if } q \in Q_E \\ +\infty & \text{if } q \in Q - Q_E \end{array} \right\}$ 
    while  $Q_{bfs} \neq []$  do       $\triangleright$  BFS over optional region.
         $q \leftarrow \text{POP}(Q_{bfs})$ 
        if  $q \in Q_l$  then
             $l' \leftarrow \min(l', dist[q])$ 
         $d \leftarrow dist[q] + 1$ 
        if  $d \leq u - l$  then
            for  $(c, q') \in \delta_{bwd}(q)$  where  $q' \in Q_{l+1}$  do
                 $S_{opt} \leftarrow S_{opt} \cup \{c\}$ 
                if  $dist[q'] > d$  then
                     $\text{PUSH}(Q_{bfs}, q')$ 
                     $Q_E \leftarrow Q_E \cup \{q'\}$ 
                     $dist[q'] \leftarrow d$ 
     $X_{l+1} = S_{opt}^{l', u-l}$ 
     $E \leftarrow Q_E \cap Q_l$ 
    for  $i \in l, l-1, \dots, 1$  do       $\triangleright$  Mandatory region
         $E' \leftarrow S_{man} \leftarrow \emptyset$ 
        for  $q \in E$  do
            for  $(c, q') \in \delta_{bwd}(q), q' \in Q_{i-1}$  do
                 $S_{man} \leftarrow S_{man} \cup \{c\}$ 
                 $E' \leftarrow E' \cup \{q'\}$ 
         $E \leftarrow E'$ 
         $X_i \leftarrow S_{man}^{1,1}$ 
    return  $E, \text{NORM}([X_1, \dots, X_{l+1}])$ 
    
```

Fig. 5. Backward pass of the algorithm. Returns the feasible starting states, and a dashed string corresponding to the refined block.

that, apart from $|\delta|$, the complexity of the propagation asymptotically depends on the characters that *must* occur in the string, and not on those that *may* appear. This makes a big difference when λ is big.

As mentioned in Sect. 2, for some set of strings we cannot define a best dashed string representation. It is therefore unlikely to have propagators maintaining consistency notions like, e.g., Generalised Arc Consistency.

If the domain of x has no optional parts, i.e., $D(x) = S_1^{l_1, l_1} \dots S_n^{l_n, l_n}$, then our approach is equivalent to the “standard” CP propagation of [17], where x corresponds to a vector of $l_1 + \dots + l_n$ integer variables $x_{i,j}$ such that $D(x_{i,j}) = S_i$ for $i = 1, \dots, n$ and $j = 1, \dots, l_i$. This is however not very interesting for string solving, where typically lengths are unknown and potentially very long.

Reverse Propagation. We can run the regular propagator in reverse, assuming we know $b = true$. In practice, we run `PROP-REIFIED-REGULAR(true, X^{-1} , R^{-1})` by reversing the dashed string $X^{-1} = S_n^{l_n, u_n} \dots S_1^{l_1, u_1}$ and the automaton $R^{-1} = \langle \Sigma, Q, F, \{(q', c, q) \mid (q, c, q') \in \delta\}, \{q_0\} \rangle$. This has a set of initial states F rather than a single state q_0 , but this only requires to initialize F_0 with $[F]$.

The reversed automaton is not a DFA hence we must omit the greyed out code. The advantage of the reversed automaton is that because the propagator is directional it may propagate where the other direction does not. Using the reversed automaton effectively doubles the time for propagation, but if it generates more propagation this can substantially reduce the total solving time, hence we leave it on by default in G-STRINGS (however the user can override this option).

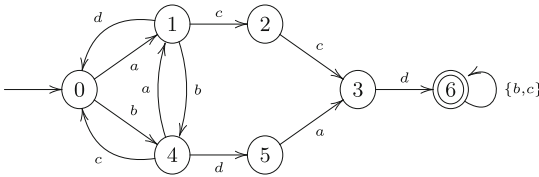


Fig. 6. Example automata R for propagating in Example 1.

Example 1. Consider the propagation of the positive constraint ($b = true$) when $D(x) = \{\mathbf{a}, \mathbf{b}\}^{0,4} \{\mathbf{c}, \mathbf{d}\}^{2,5} \{\mathbf{a}, \mathbf{b}\}^{0,5}$ and automata given by the DFA shown in Fig. 6. The forward propagation determines sets of states $F_1 = [\{0\}, \{0, 1, 4\}]$, $F_2 = [\{0, 1, 4\}, \{2, 5\}, \{3\}, \{3, 6\}]$, and $F_3 = [\{3, 6\}, \{3, 6\}]$.

The backwards pass for block 3 starts from state $\{6\}$ and determines it can reach only set of states $\{6\}$ using $\{\mathbf{b}\}$. It returns $\{6\}, \{\mathbf{b}\}^{0,5}$.

The backwards pass for block 2 starts from state $\{6\}$, It sets $dist[6] = 0$ and the rest to infinity. It then determines it can reach 3 at distance 1, 2 at distance 2, and 1 at distance 3 (node 0 at distance 4 is not considered since $4 > 5 - 2$). Since $3 \in Q_2 = \{3\}$ is only reachable at distance 1, we find $l' = 1$. This creates the optional block $\{\mathbf{c}, \mathbf{d}\}^{1,3}$. We then consider one step backwards from $\{3\}$ which reaches $\{2\}$ and creates block $\{\mathbf{c}\}^{1,1}$, then one step backwards from $\{2\}$, which reaches $\{1\}$ and creates block $\{\mathbf{c}\}^{1,1}$. The function returns $\{1\}, \{\mathbf{c}\}^{2,2} \{\mathbf{c}, \mathbf{d}\}^{1,3}$.

The backwards pass for block 1 starts from state $\{1\}$ and determines it can reach $\{0, 4\}$ at distance 1 and $\{1\}$ at distance 2. We find $l' = 1$ and the function returns $\{0\}, \{\mathbf{a}, \mathbf{b}\}^{1,4}$. So, $D(x)$ becomes $\{\mathbf{a}, \mathbf{b}\}^{1,4} \{\mathbf{c}\}^{2,2} \{\mathbf{c}, \mathbf{d}\}^{1,3} \{\mathbf{b}\}^{0,5}$.

Note the propagator is *not idempotent*. Running it again will determine the finer domain $\{\mathbf{a}, \mathbf{b}\}^{1,4} \{\mathbf{c}\}^{2,2} \{\mathbf{d}\}^{1,1} \{\mathbf{c}\}^{0,2} \{\mathbf{b}\}^{0,5}$. Running the reverse propagator will split the first block into $\{\mathbf{a}, \mathbf{b}\}^{0,3} \{\mathbf{a}\}^{1,1}$. We can thus infer that substring `accd` must occur in x .

□

4 Regular Expressions Decomposition

A natural way to express a constraint of the form $x \in L(R)$, where R is an automaton, is to give an equivalent formulation $x \in L(r)$ in terms of an equivalent regular expression $r \in \mathcal{RE}$. We observed that these kind of constraints often occur in SMTLIB instances derived from real-world program analysis.

We can easily deal with constraints of the form $b \Leftrightarrow x \in L(r)$ by simply propagating $b = \text{REGULAR}(x, \text{DFA}(r))$, where DFA is the function introduced in Sect. 2.3 for converting a given regular expression into a DFA. However, if $b = \text{true}$, we could avoid instantiating a propagator entirely.

First, we observe that dashed strings are themselves a particular class of regular expressions: the language $\gamma(X)$ denoted by $X = S_1^{l_1, u_1} \dots S_k^{l_k, u_k}$ actually corresponds to $L(r) \cap \mathbb{S}$ where $r = r_1^{l_1, u_1} \dots r_k^{l_k, u_k}$ with $r_i = \underbrace{(c_{i,1} | \dots | c_{i, n_i})}_{l \text{ times}} \underbrace{((r_i | \epsilon) \dots (r_i | \epsilon))}_{u-l \text{ times}}$,

$S_i = \{c_{i,1}, \dots, c_{i, n_i}\}$, and $r_i^{l,u}$ is a shorthand for $(r_i \dots r_i)((r_i | \epsilon) \dots (r_i | \epsilon))$ for $i = 1, \dots, k$. Hence we can directly encode some classes of REGULAR constraints as domain constraints on dashed strings (e.g., the occurrence of characters or substrings in a given string). Moreover, with the help of auxiliary string constraints we can also encode more complex regular expressions. For example, $x \in L(\text{fee|foo}\bar{\text{}})$ can be reformulated into $x = yz \wedge y \in \{\text{fee}, \text{foo}\} \wedge z = \bar{\text{}}$.

Unfortunately, not all regular expressions are easily decomposable into basic string constraints. For example, we can easily map $x \in L((\mathbf{a|bc})^*)$ into the domain constraint $x :: \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}^{0,\lambda}$ but we cannot do the same for the constraint $x \in L(\mathbf{a|bc}^*)$ because this would require to have a propagator for the iterated concatenation of sets of strings $\{\mathbf{a}, \mathbf{bc}\}^n, n \geq 0$.²

Hence, we use simple syntactic pre-checks to identify opportunities for decomposing a constraint $\text{true} \Leftrightarrow x \in L(r)$ into a conjunction $C_1 \wedge \dots \wedge C_k$ of basic string constraints (if $D(b) \neq \{\text{true}\}$, we simply propagate $b = \text{REGULAR}(x, \text{DFA}(r))$). We indicate with $x \in L(r) \models C_1 \wedge \dots \wedge C_k$ such a decomposition.

$$x \in L(\emptyset) \models \text{false} \qquad x \in L(c) \models x = c \quad \text{if } c \in \Sigma \cup \{\epsilon\} \quad (1)$$

$$x \in L(r_1 \cdot r_2) \models x_1 \in L(r_1) \wedge x_2 \in L(r_2) \wedge x = x_1 \cdot x_2 \quad (2)$$

$$x \in L(r_1 | r_2) \models x_1 \in L(r_1) \wedge x_2 \in L(r_2) \wedge n \in \{1, 2\} \wedge x = [x_1, x_2][n] \quad (3)$$

$$x \in L((r_1 | \dots | r_k)^*) \models x :: \{c_1, \dots, c_k\}^{0,\lambda} \quad \text{if } L(r_i) = \{c_i\} \subseteq \Sigma \text{ for } i = 1, \dots, k \quad (4)$$

$$x \in L(r^*) \models n \in [0, \lambda] \wedge x = w^n \quad \text{if } L(r) = \{w\} \subseteq \Sigma^* \quad (5)$$

Fig. 7. Decomposition rules. Variables x_1, x_2 are new string variables, n a new integer variable, and r, r_1, r_2, \dots, r_k are regular expressions.

² Note this is different from the iterated concatenation of strings. For example, encoding $x \in L(\mathbf{a|bc}^*)$ with $x = y^n \wedge n \geq 0 \wedge y \in \{\mathbf{a}, \mathbf{bc}\}$ is unsound because this actually encodes the constraint $x \in L(\mathbf{a}^* | (\mathbf{bc})^*)$ (e.g., $\mathbf{abc} \in L(\mathbf{a|bc}^*) - L(\mathbf{a}^* | (\mathbf{bc})^*)$).

Figure 7 summarises the decomposition rules we implemented. Rules 1–2 are straight rewritings into equality/concatenation. Rule 3 encodes the construct $x \in \{x_1, x_2\}$ by means of the ELEMENT global constraint [8]. Rule 4 decomposes into a domain constraint when a regular expression r_i denotes a single character c_i , while rule 5 takes advantage of iterated concatenation when r denotes a single string w . In addition to these rules, we also implemented a number of other rules to cope with SMTLIB syntax (e.g., we also decompose $x \in L([a, b])$ and $x \in L([a, b]^*)$, where $[a, b]$ is the range of characters denoted by $(a|\dots|b)$).

Note that this reformulation does not detect all opportunities for reformulation. Indeed, deciding DFA *primality* – that is, whether there exist non-trivial L_1, L_2 such that $L(R) = L_1L_2$ for a given DFA R – is PSPACE-hard [15], so an efficient complete method is vanishingly unlikely. Nevertheless, devising a ‘good enough’ decomposition method remains an interesting challenge.

As we shall see in Sect. 5, decomposing a regular expression r can be advantageous since it can significantly reduce the number of states of $\text{DFA}(r)$, especially when dealing with expressions involving very long fixed strings, and the number of propagations performed. Let us clarify this by providing an example extracted from the empirical evaluation of Sect. 5.

Example 2. Consider the constraint $x \in L(\mathbf{a}^*\mathbf{b}\mathbf{b}^*) \wedge x \in L((\mathbf{a}|\mathbf{b})^*\mathbf{b}\mathbf{a}(\mathbf{a}|\mathbf{b})^*)$, which is clearly unsatisfiable since the $\mathbf{b}\mathbf{a}$ sub-string in the second expression conflicts with the first expression, imposing each character \mathbf{b} to be followed by only \mathbf{b} ’s.

If we do not decompose the regular expressions, the propagation algorithm is only able to infer that $x :: \{\mathbf{a}, \mathbf{b}\}^{2;\lambda-1}\{\mathbf{b}\}^{1,1}$ from the corresponding DFAs. This entails that we have to branch on x and explore all the possible alternatives to detect the unsatisfiability: the solving time clearly depends on λ .

Conversely, by decomposing the expressions into basic string constraints we can infer that $x :: \{\mathbf{a}\}^{0,\lambda-1}\{\mathbf{b}\}^{1,\lambda} \wedge x :: \{\mathbf{a}, \mathbf{b}\}^{0,\lambda-2}\{\mathbf{b}\}^{1,1}\{\mathbf{a}\}^{1,1}\{\mathbf{a}, \mathbf{b}\}^{0,\lambda-2}$. In this case, no search is performed because the sweep-based equate algorithm [5] implemented in G-STRINGS instantaneously triggers a failure since the two dashed strings are not equatable.

□

5 Evaluation

We implemented the REGULAR propagator in the G-STRINGS solver, and we also implemented a MiniZinc/FlatZinc interface for it [3]. The user can either specify a (reified) REGULAR constraint in terms of a regular expression or a finite state automaton. We tested G-STRINGS on three well-known SMTLIB string benchmarks containing regular expressions:

- APPSCAN: 8 satisfiable instances derived from security analysis performed by IBM AppScan tool [13]. We discarded two of them (namely, `t01` and `t06`) since they do not contain regular. The remaining 6 instances contain only decomposable and non-reified regular expressions.

- STRANGER: 3392 instances derived by Stranger [22] tool from real-world PHP programs. These instances are not publicly available and the authors sent them to us privately, but 56 of them were malformed. We thus ended up with 3336 SMTLIB instances, containing only decomposable and non-reified regular expressions.
- NORN: 1027 instances generated by a model checker based on CEGAR refinement [2], from which we discarded 24 instances not containing regular expressions. In the remaining 1003 instances, 942 contain at least a non-decomposable expression and 870 contain at least a reified regular expressions (all of these are negated expressions of the form $false \Leftrightarrow x \in L(r)$).

We compared G-DECOMP (the version of G-STRINGS that always unfolds decomposable, not-reified regular expressions using the method of Sect. 4) and G-NOTDEC (the version that never unfolds them) against three state-of-the-art string solvers: two SMT solvers supporting the theory of strings (namely, Z3STR3 [9] and CVC4 [14]), and a CP solver that extends GECODE for supporting bounded-length string variables, i.e., GECODE+S [18].³

Note that GECODE+S is no longer actively developed. So, since its FlatZinc support is incomplete, we used a compiler from SMTLIB to C++ that the authors used for their previous experiments (here we call it `smt2cpp`). For G-DECOMP and G-NOTDEC we instead took advantage of the SMTLIB to MiniZinc compiler introduced in [4]. We run all the experiments on a Ubuntu 15.10 machine with 16 GB of RAM and 2.60 GHz Intel[®] i7 CPU by setting a solving timeout of $T = 300$ s and varying $\lambda \in \{500, 1000, 10000\}$.

Table 1. APPSCAN results. Times are in seconds.

Instances	t02	t03	t04	t05	t07	t08
G-STRINGS	0.00	0.00	0.00	0.00	0.00	0.00
Z3STR3	0.10	0.29	0.38	0.97	2.09	0.02
CVC4	0.01	6.12	T	5.75	0.00	0.27

5.1 AppScan and Stranger Benchmarks

Results on APPSCAN benchmark are shown in Table 1. GECODE+S is not included in the comparison since `smt2cpp` could not process these instances (statements like logical implication and `if-then-else` are not supported).

APPSCAN is not a challenging benchmark: SMT solvers can solve most of the problems in a short time, while G-STRINGS resolution is instantaneous. Here we do not discriminate between G-DECOMP and G-NOTDEC since they both find a solution in 0 s, regardless of maximum length λ .

³ We used Z3STR3 4.6.2 and CVC4 1.5. The source code of the experiments is publicly available at: <https://bitbucket.org/robama/exp.cp.2018>.

Table 2. STRANGER results. Times are in seconds.

	<i>SAT</i>		<i>UNS</i>		<i>TOT</i>	
	Solved	Runtime	Solved	Runtime	Solved	Runtime
G-DECOMP	1254	0.20	2082	0.00	3336	0.08
G-NOTDEC	1254	0.65	2082	0.01	3336	0.25
CVC4	1247	2.19	2082	0.01	3329	0.88
Z3STR3	936	76.27	2082	0.09	3018	28.68

Results on the STRANGER benchmark are shown in Table 2. Solving time is set to T when a solver can not solve an instance. Note that we are considering the simplified instances used also in [12], where the `str.replaceall` operation is replaced by `str.replace`. GECODE+S is not included in the results because of unsupported constraints (e.g., string replacement) and characters (all the STRANGER instances contain extended ASCII characters, while the alphabet size of GECODE+S is limited to 64 characters).

For G-DECOMP and G-NOTDEC, only the results with $\lambda = 10000$ are presented: here strings can be very long, so with $\lambda = 500$ we have 107 unsound results: we instantaneously detect the unsatisfiability because at least one string has length greater than 500. With $\lambda = 1000$, we have 50 unsound results. For both G-DECOMP and G-NOTDEC, we branched on binary variables first.

The two versions of G-STRINGS outperform the SMT solvers, especially on the satisfiable instances (the unsatisfiable ones appear very easy to solve). We can also observe the benefits of decomposition (all the REGULAR constraints of this benchmark can be rewritten into concatenation constraints). On average, G-DECOMP is more than three times faster than G-NOTDEC.

5.2 Norn Benchmark

Results on the 1003 instances of the NORN benchmark are shown in Table 3. Note that `smt2cpp` can process only 150 of them, mainly because GECODE+S does not support negated regular expressions. CVC4 and Z3STR3 work on unbounded strings, so they do not use a maximum string size.

The results clearly show that G-STRINGS is faster and more powerful than alternative approaches. The advantages of decomposition are also illustrated, although the performance of G-NOTDEC and G-DECOMP is not so different (in particular on satisfiable instances they are equivalent).

Conversely to the STRANGER benchmark here satisfiable instances are trivial, while unsatisfiable ones are harder to solve. This is in general not surprising for CP solvers – especially those like GECODE not employing nogood learning – and this in particular holds for G-STRINGS, which is based on GECODE and for which the resolution is obviously influenced by λ size. As an example, let us

consider the only instance that no solver can solve within the time limit T .⁴ This unsatisfiable instance is hard to solve since it contains a pattern of the form:

$$x, y :: \{\mathbf{b}\}^{1,\lambda} \wedge x \cdot \mathbf{z} \cdot y \in L((\mathbf{bz}^*\mathbf{b})^*) \wedge x \cdot \mathbf{z} \cdot y \cdot \mathbf{b} \notin L((\mathbf{bz}^*\mathbf{b})^*\mathbf{b})$$

with x, y string variables and \mathbf{b}, \mathbf{z} characters of Σ . Unfortunately, our propagation algorithms cannot further narrow the domains of x and y , and thus we have to rely on branching. This means that $O(|D(x) \times D(y)|) = O(\lambda^2)$ nodes must be explored to detect the inconsistency.

Table 3. NORN results. Times are in seconds.

	Solver	G-DECOMP			G-NOTDEC			CVC4	Z3STR3	GECODE+S		
	λ	500	1000	10000	500	1000	10000			500	1000	10000
<i>SAT</i>	Solved	688	688	688	688	688	688	627	178	75	75	43
	Runtime	0.00	0.00	0.00	0.00	0.00	0.00	29.82	223.10	267.65	268.43	283.90
<i>UNS</i>	Solved	314	314	312	312	309	307	182	89	41	41	25
	Runtime	0.96	1.00	3.81	4.82	5.75	7.62	123.64	214.02	260.3	261.54	276.88
<i>TOT</i>	Solved	1002	1002	1000	1000	997	995	809	267	116	116	68
	Runtime	0.30	0.31	1.20	1.51	1.81	2.39	58.72	265.38	266.31	248.31	281.74

Table 4. NORN results on the 116 instances that GECODE+S can solve.

Solver	G-DECOMP			G-NOTDEC			GECODE+S			CVC4	Z3STR3
	λ	500	1000	10000	500	1000	10000	500	1000		
Solved	116	116	116	116	115	115	116	116	68	104	46
Runtime	0.00	0.00	0.00	1.20	2.59	2.59	0.69	8.69	142.09	32.72	181.41

GECODE+S is the closest approach to G-STRINGS. It implements the CP approach of [17] with a termination character for strings with length less than λ . If we consider only the 116 instances that GECODE+S can correctly solve (see Table 4) G-STRINGS is on average still faster and, as already observed in [5, 6], its performance decay is less pronounced as λ grows.

6 Conclusion

We have presented a propagation algorithm for enforcing REGULAR constraints over dashed strings. Unlike existing propagators for REGULAR, the algorithm runs in time independent of the upper bound on the string length. We have also identified a sub-class of regular expressions which may be translated directly into dashed string domain constraints.

⁴ Precisely, this is the instance 489 of the `HammingDistance` class. G-DECOMP with $\lambda = 500$ takes 454.6 s to detect the unsatisfiability. Clearly, we can only prove that there is no solution where all string variables x have length $|x| \leq \lambda$.

We have demonstrated the effectiveness of the propagator on three sets of existing string constraint problems. On these benchmarks, G-STRINGS is considerably faster and more robust than existing solvers.

We also defined the first propagator we are aware of for reified REGULAR. The same modifications we use here could be adapted to create a reified REGULAR propagator on fixed length arrays, as is standard in CP.

Acknowledgments. This work is supported by the Australian Research Council (ARC) through Linkage Project Grant LP140100437 and Discovery Early Career Researcher Award DE160100568.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Flatten and conquer: a framework for efficient analysis of string constraints. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI 2017, pp. 602–617, Barcelona, Spain, 18–23 June 2017 (2017)
2. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: an SMT solver for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 462–469. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_29
3. Amadini, R., Flener, P., Pearson, J., Scott, J.D., Stuckey, P.J., Tack, G.: MiniZinc with Strings. In: Logic-Based Program Synthesis and Transformation - 25th International Symposium LOPSTR 2016 (2016). <https://arxiv.org/abs/1608.03650>
4. Amadini, R., Gange, G., Stuckey, P.J.: Propagating LEX, FIND and REPLACE with dashed strings. In: van Hove, W.-J. (ed.) CPAIOR 2018. LNCS, vol. 10848, pp. 18–34. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93031-2_2
5. Amadini, R., Gange, G., Stuckey, P.J.: Sweep-based propagation for string constraint solving. In: To appear in AAAI 2018 (2018)
6. Amadini, R., Gange, G., Stuckey, P.J., Tack, G.: A novel approach to string constraint solving. In: Beck, J.C. (ed.) CP 2017. LNCS, vol. 10416, pp. 3–20. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66158-2_1
7. Barták, R.: Modelling resource transitions in constraint-based scheduling. In: Grosky, W.I., Plášil, F. (eds.) SOFSEM 2002. LNCS, vol. 2540, pp. 186–194. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36137-5_13
8. Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T.: Global constraint catalogue: past, present and future. Constraints **12**(1), 21–62 (2007). <http://sofdem.github.io/gccat/>
9. Berzish, M., Zheng, Y., Ganesh, V.: Z3str3: A string solver with theory-aware branching. CoRR abs/1704.07935 (2017). <http://arxiv.org/abs/1704.07935>
10. Cheng, K.C.K., Yap, R.H.C.: Maintaining generalized arc consistency on Ad Hoc r -ary constraints. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 509–523. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85958-1_34
11. Gecode Team: Gecode: Generic constraint development environment (2016). <http://www.gecode.org>
12. Holík, L., Janku, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. PACMPL **2**(POPL), 4:1–4:32 (2018)

13. IBM: Security AppScan (2018). <https://www.ibm.com/security/application-security/appscan>
14. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 646–662. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_43
15. Martens, W., Niewerth, M., Schwentick, T.: Schema design for XML repositories: complexity and tractability. In: Paredaens, J., Gucht, D.V. (eds.) Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, Indianapolis, Indiana, USA, pp. 239–250. ACM, 6–11 June 2010 (2010)
16. Perez, G., Régim, J.-C.: Improving GAC-4 for table and MDD constraints. In: O’Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 606–621. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_44
17. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 482–495. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30201-8_36
18. Scott, J.D., Flener, P., Pearson, J., Schulte, C.: Design and implementation of bounded-length sequence variables. In: Salvagnin, D., Lombardi, M. (eds.) CPAIOR 2017. LNCS, vol. 10335, pp. 51–67. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59776-8_5
19. Tateishi, T., Pistoia, M., Tripp, O.: Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Trans. Softw. Eng. Methodol.* **22**(4), 33 (2013)
20. Thomé, J., Shar, L.K., Bianculli, D., Briand, L.C.: Search-driven string constraint solving for vulnerability detection. In: Proceedings of the 39th International Conference on Software Engineering ICSE 2017, Buenos Aires, Argentina, pp. 198–208, 20–28 May 2017 (2017)
21. Trinh, M., Chu, D., Jaffar, J.: S3: a symbolic string solver for vulnerability detection in web applications. In: SIGSAC, pp. 1232–1243. ACM (2014)
22. Yu, F., Alkhalaf, M., Bultan, T.: STRANGER: an automata-based string analysis tool for PHP. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 154–157. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_13



A Constraint-Based Encoding for Domain-Independent Temporal Planning

Arthur Bit-Monnot^{1,2}(✉)

¹ University of Genoa, Genoa, Italy
afbit@uniss.it

² University of Sassari, Sassari, Italy

Abstract. We present a general constraint-based encoding for domain-independent task planning. Task planning is characterized by causal relationships expressed as conditions and effects of optional actions. Possible actions are typically represented by templates, where each template can be instantiated into a number of primitive actions.

While most previous work for domain-independent task planning has focused on primitive actions in a state-oriented view, our encoding uses a fully lifted representation at the level of action templates. It follows a time-oriented view in the spirit of previous work in constraint-based scheduling.

As a result, the proposed encoding is simple and compact as it grows with the number of actions in a solution plan rather than the number of possible primitive actions. When solved with an SMT solver, we show that the proposed encoding is slightly more efficient than state-of-the-art methods on temporally constrained planning benchmarks while clearly outperforming other fully constraint-based approaches.

1 Introduction

Task planning is a field of Artificial Intelligence concerned with finding a set of actions that would result in desirable state. Its key difficulty lies in the handling of causal relationships between a large number of potential actions. Research in the field has focused primarily on the definition of relaxations of the classical planning problem in order to define accurate heuristic functions to guide tree search algorithms. Those heuristics have proved to be highly effective for reasoning on the causal relationships that occur in task planning and have driven most research to focus on state-based heuristic search.

Despite their success in classical planning, they have proved to be difficult to extend to more expressive models including time, resources or continuous changes. This led to a renewal of interest into constraint-based models and search as a way to increase the expressiveness of domain-independent task planners [10, 18, 31].

In this paper, we propose an encoding for the causal relationships that are at the core of domain-independent planning. Unlike the most recent work proposing compilations of domain-independent temporal planning to Constraint Satisfaction Problems (CSPs) which follow a state-oriented view [9, 10], we use a time-oriented view inspired by the work in the field of constraint-based planning and scheduling [11, 23, 24, 26]. The key benefit of this encoding is its simplicity and its good fit for integration into more general constraint satisfaction problems, going one step closer to bridging the gap between planning and scheduling.

We start by giving a background on task planning and the representation of temporal planning problems as chronicles. We then show how such planning problems can be concisely encoded into CSPs, using a fully lifted representation based on chronicles. The resulting encoding is leveraged in a domain-independent planner for ANML, an expressive language for the specification of planning problems [32]. Using an off-the-shelf SMT solver as a backend, the planner is shown to outperform state-of-the-art temporal planners on temporally constrained planning benchmarks. Last, we discuss the related work in constraint-based planning and scheduling, especially analyzing other domain-independent planners that rely on SMT.

2 Background

2.1 A Distilled Planning Problem

In its simplest formulation, a planning problem is composed of (i) an initial state, (ii) a goal state and (iii) a set of primitive actions.

A state is an assignment to a set of *state variables*, each denoting one particular feature of the environment (e.g. the location of a truck). We denote as S the set of possible states.

A primitive action a is composed of:

- a set of conditions over state variables. It characterizes the set of states $S_a^{app} \subseteq S$ in which the action is applicable.
- a set of effects from which one can construct a state transition function $f_a : S_a^{app} \rightarrow S$.

Given an initial state $s_0 \in S$, a goal state $s_g \in S$ and a set of primitive actions, a plan is a sequence of primitive actions $\langle a_1, \dots, a_n \rangle$. For a given plan, one can build the resulting sequence of states $\langle s_1, \dots, s_n \rangle$ such that $s_i = f_{a_i}(s_{i-1})$. A plan is a solution to a planning problem if all actions are applicable in the previous state ($\forall_{i \in [1, n]} s_{i-1} \in S_{a_i}^{app}$) and the final state is the goal state ($s_n = s_g$).

This formulation of planning as a state transition system has been at the core of domain-independent planning research, and of PDDL, the de-facto standard language for describing planning problems [29]. Several extensions have been devised for extending the expressiveness of PDDL, most notably to handle a limited form of temporal constructs and numeric variables [19, 21]. Nevertheless, the focus of the language and the benchmarks of the International Planning Competition remains heavily focused on handling the causal relationships derived from the conditions and effects of primitive actions.

2.2 Temporal Planning as Chronicles

An other notable representation for planning follows a time-oriented view as opposed to the state-oriented view above. We here detail the representation of planning problems as chronicles [25], which was first introduced in the IxTeT planner [24]. This representation avoids direct references to states and instead describes the evolution of the environment through temporally qualified assertions over the state variables.

A *type* is defined by a set of values. A type is either (i) a set of domain constants (e.g. the type $Truck = \{ R_1, R_2 \}$ defines two truck objects R_1 and R_2 in the planning problem), (ii) a discrete or continuous set of numeric values, or (iii) a representation of temporal instants, for which we typically consider the set of natural numbers.¹

A (decision) *variable* is associated with a type that defines its initial domain. We denote as *timepoints* the subset of variables that represent a temporal instant.

A *state variable* denotes the evolution of a particular state feature over time. A state variable is typically parameterized by one or multiple domain objects to represent the state of a particular object in a planning problem. For instance $loc(R_1)$ denotes the evolution of the location of the truck R_1 over time. A state variable expression can be parameterized by decision variables, in which case the actual state variable it refers to depends on the value taken by its parameters, e.g., $loc(r)$ will refer to $loc(R_1)$ or $loc(R_2)$ depending on the value taken by the variable r of type *Truck*.

Chronicle. In line with previous work in temporal planning, we represent the core constructs of a planning problem with chronicles [25]. A chronicle is a tuple (V, X, C, E) where:

- V is a set of variables.
- X is a set of constraints over the variables in V .
- C is a set of conditions. Each condition is of the form $[s, e] sv(p_1, \dots, p_n) = v$ where s and e are timepoints in V , $sv(p_1, \dots, p_n)$ is a parameterized state variable (with each $p_i \in V$) and $v \in V$ is a variable. A condition states that the state variable $sv(p_1, \dots, p_n)$ must have the value v over the temporal interval $[s, e]$.
- E is a set of effects. Each effect is of the form $[s, e] sv(p_1, \dots, p_n) \leftarrow v$ where s and e are timepoints, $sv(p_1, \dots, p_n)$ is a parameterized state variable (with each $p_i \in V$) and $v \in V$ is a variable. Such an effect states that the state variable $sv(p_1, \dots, p_n)$ will take the value v at time e . Over the temporal interval $]s, e[$ the state variable is transitioning from its previous value to v : it has an undefined value and cannot be further constrained.

¹ Note that this integer based representation of time is no less expressive than a real-valued representation when forbidding instantaneous changes, as common in temporal planning [15].

A chronicle is thus a constraint satisfaction problem (V, X) extended with additional constructs to represent the conditions and effects that are at the core of planning problems.²

Action Chronicle. A planning problem defines a set of action templates \mathcal{T} where each action template can be instantiated into chronicles. We denote as \mathcal{C}_a^i a chronicle instantiated from an action template $a \in \mathcal{T}$, where i distinguishes chronicles instantiated from the same template.

Considering an action template $Go \in \mathcal{T}$ that moves a truck r from a location ℓ_s to a location ℓ_e , its instantiation as a chronicle \mathcal{C}_{Go} could have the following components:

- $V = \{ r, \ell_s, \ell_e, t_s, t_e \}$ where r, ℓ_s, ℓ_e are variables representing the parameters of the action (truck, start location and end location) and t_s and t_e are timepoints representing the start and end times of the action.
- $X = \{ t_e = t_s + 10, \ell_s \neq \ell_e \}$, constraints stating that the action takes 10 time units and that the origin ℓ_s and destination ℓ_e must be different.
- $C = \{ [t_s, t_s] \text{ loc}(r) = \ell_s \}$, i.e., the truck r must be in location ℓ_s at the action's start t_s .
- $E = \{ [t_s, t_e] \text{ loc}(r) \leftarrow \ell_e \}$, i.e., the truck r will be in location ℓ_e at the action's end t_e . Its location is undefined for the duration of the action $]t_s, t_e[$.

Such a chronicle is called an *action chronicle*.

Initial Chronicle. We distinguish action chronicles from the initial chronicle \mathcal{C}_0 that represents the initial state and the planning objectives. In addition, the initial chronicle might provide a partial view of the expected state evolution, e.g., that a cargo ship (outside the control of the planner) will arrive at 5pm.

Specifying a problem where the truck R_1 is initially in location L_0 and must be in location L_2 or L_3 before time 100 is encoded with the chronicle $\mathcal{C}_0 = (V_0, X_0, C_0, E_0)$ where:

- $V_0 = \{ t, \ell \}$.
- $X_0 = \{ t < 100, \ell = L_2 \vee \ell = L_3 \}$, constraints restricting the solution set.
- $C_0 = \{ [t, t] \text{ loc}(R_1) = \ell \}$, condition specifying the goal.
- $E_0 = \{ [0, 0] \text{ loc}(R_1) \leftarrow L_0 \}$, effect defining the initial state: the truck R_1 is in L_0 at the initial time.

A planning problem can be represented as a pair $(\mathcal{C}_0, \mathcal{T})$ where \mathcal{C}_0 defines the initial state and objectives and \mathcal{T} is a set of action templates that can be instantiated into action chronicles.

² The original chronicle model used transitions instead of effects. We use effects to more closely match the classical definition of planning problems and simplify the presentation. Note that transitions can still be straightforwardly encoded by combining a condition and an effect.

3 Planning as a Constraint Satisfaction Problem

One of the difficulty that arises in planning is that the number of actions is not known beforehand nor can it be given tight bounds. As a first step, we consider a *bounded* planning problem composed of an initial chronicle \mathcal{C}_0 and finite set of *optional* action chronicles AC . An optional action chronicle $\mathcal{C}_a^i \in AC$ is a action chronicle associated to a boolean variable o_a^i that is true (\top) if \mathcal{C}_a^i is part of the solution plan and false (\perp) otherwise.

Consider the planning problem of moving the truck R to either $L2$ or $L3$ from the previous section. Considering a set of optional action chronicles $\{\mathcal{C}_{Go}^1, \mathcal{C}_{Go}^2\}$, we can represent all the plans composed of 0, 1 or 2 instances of the Go action. The actual plan depends on the instantiation of the decision variables. One could build a satisfying solution from the partial assignment $\{o_{Go}^1 \leftarrow \top, r^1 \leftarrow R_1, \ell_s^1 \leftarrow L_0, \ell_e^1 \leftarrow L_2, o_{Go}^2 \leftarrow \perp\}$ where r^1, ℓ_s^1 and ℓ_e^2 are the eponymous variables in \mathcal{C}_{Go}^1 . Such an assignment states that only the first action would be present and would move the truck R_1 from L_0 to L_2 . Note that this only partially defines the solution plan and other assignments would need to be made, notably to the action timepoints and to the variables of \mathcal{C}_0 .

A *bounded* planning problem Π is the set of chronicles $\{\mathcal{C}_0\} \cup AC$ where each $\mathcal{C} \in \Pi$ is associated to a boolean variable $present(\mathcal{C})$ that is: o_a^i if \mathcal{C} is an action chronicle $\mathcal{C}_a^i \in AC$; or \top if $\mathcal{C} = \mathcal{C}_0$. This planning problem is called *bounded* as there is an upper limit on the number of actions in a solution plan defined by the number of action chronicles.

3.1 Building Blocks

Given a bounded planning problem Π , we now describe the core structures for compiling it to a constraint satisfaction problem.

Condition Token. Given a chronicle $\mathcal{C} \in \Pi$, each condition $[s, e] sv(p_1, \dots, p_n) = v$ in \mathcal{C} is associated to a *condition token*:

$$present(\mathcal{C}) : [s, e] sv(p_1, \dots, p_n) = v$$

This token represents that, if \mathcal{C} is part of the solution ($present(\mathcal{C}) = \top$), then the state variable $sv(p_1, \dots, p_n)$ must have the value v over the temporal interval $[s, e]$.

We denote as C_Π the set of condition tokens in Π .

Effect Token. Given a chronicle $\mathcal{C} \in \Pi$, we associate to each effect $[s, e] sv(p_1, \dots, p_n) \leftarrow v$ in \mathcal{C} an *effect token*:

$$present(\mathcal{C}) : [s, e, t] sv(p_1, \dots, p_n) \leftarrow v$$

where t is a new timepoint variable. This token states that, if \mathcal{C} is part of the solution ($present(\mathcal{C}) = \top$), then the state variable $sv(p_1, \dots, p_n)$ is undefined

over the temporal interval $]s, e[$ and has the value v over the temporal interval $[e, t]$. The introduction of the new decision variable t allows us to encode a minimal time for which the effect will persist.

We denote as E_{Π} the set of effect tokens in Π .

Characteristics of Tokens. Effect tokens here represent the evolution of state variables over time. Each (present) effect token imposes a new value to its state variable. This value is constrained to be maintained on a given temporal interval. Effect tokens thus encode the state evolution. On the other hand, condition tokens place constraints on the state evolution by requesting a given state variable to have a given value over a temporal interval. Intuitively, such a condition is achieved if there is a corresponding effect token that imposes the appropriate value.

For a given token, the variables involved allow four degrees of freedom on which a solver might play to build a consistent plan:

- the presence of the token (variable $present(\cdot)$).
- the temporal interval on which it applies, (variables s, e and t).
- the state variable on which it applies (variables p_1, \dots, p_n).
- the value taken by the state variable (variable v).

Those variables are typically shared with other tokens from the same action chronicle. These interdependencies between tokens are at the core of the hardness of planning as having an effect token to establish a condition requires the presence of all other tokens from the same chronicle. This in turn requires new conditions to be established as well as new effect tokens that might interact with the existing ones.

3.2 Constraints for Plan Consistency

Given a bounded planning problem Π , we define a set of constraints that encode the requirements for a plan to be consistent.

Coherence Constraint. State variables are similar to unary resources in that they can only take a single value at a given time. Any two distinct effect tokens α and α' in E_{Π} must be coherent: they may not concurrently impose a value to the same state variable.

$$\begin{aligned} \text{Given } \alpha &= \langle o : [s, e, t] \text{ } sv(p_1, \dots, p_n) \leftarrow v \rangle \in E_{\Pi} \\ \alpha' &= \langle o' : [s', e', t'] \text{ } sv(p'_1, \dots, p'_n) \leftarrow v' \rangle \in E_{\Pi} \end{aligned}$$

then the constraint $coherent(\alpha, \alpha')$ is defined as:

$$o \wedge o' \implies t \leq s' \vee t' \leq s \vee p_1 \neq p'_1 \vee \dots \vee p_n \neq p'_n$$

The coherence constraint enforces that a given state variable has at most one value at any point in time. It is done by forcing two effect tokens to be non overlapping which can be enforced along three axes: presence (o), time ($]s, t]$) and state variable ($sv(p_1, \dots, p_n)$).

Support Constraint. We say that a condition token $\beta \in C_\Pi$, is *supported by* an effect token $\alpha' \in E_\Pi$ if α' establishes the value required by β and that this value persists for the duration of β .

$$\begin{aligned} \text{Given } \beta &= \langle o : [s, e] \text{ } sv(p_1, \dots, p_n) = v \rangle \in C_\Pi \\ \alpha' &= \langle o' : [s', e', t'] \text{ } sv(p'_1, \dots, p'_n) \leftarrow v' \rangle \in E_\Pi \end{aligned}$$

then *supported-by*(β, α') is defined as:

$$o' \wedge e' \leq s \wedge e \leq t' \wedge p_1 = p'_1 \wedge \dots \wedge p_n = p'_n \wedge v = v'$$

Less formally, it means that β is supported by α' if (i) α' is present, (ii) β is contained in the interval $[e', t']$ over which the effect of α' persists, and (iii) α' establishes the right value on the right state variable.

The fact that a condition token $\beta \in C_\Pi$ must be supported by at least one effect token if it is present is encoded by the constraint *supported*(β):

$$\text{present}(\beta) \implies \bigvee_{\alpha' \in E_\Pi} \text{supported-by}(\beta, \alpha')$$

Internal Chronicle Consistency. Given an optional chronicle $\mathcal{C} = (V, X, C, E)$, if it is part of the solution then all its internal constraints must hold, which is represented by the constraint *consistent*(\mathcal{C}):

$$\text{present}(\mathcal{C}) \implies \bigwedge_{c \in X} c$$

Formulation as a CSP. A bounded planning problem Π is encoded as a Constraint Satisfaction Problem (V_Π, X_Π) where:

$$\begin{aligned} V_\Pi &= \{ V \mid (V, X, C, E) \in \Pi \} \\ &\cup \{ \text{present}(\mathcal{C}) \mid \mathcal{C} \in \Pi \} \\ X_\Pi &= \{ \text{coherent}(\alpha, \alpha') \mid \alpha, \alpha' \in E_\Pi, \alpha \neq \alpha' \} \\ &\cup \{ \text{supported}(\beta) \mid \beta \in C_\Pi \} \\ &\cup \{ \text{consistent}(\mathcal{C}) \mid \mathcal{C} \in \Pi \} \end{aligned}$$

3.3 Symmetry Breaking Constraints

A given bounded planning problem Π might contain several action chronicles of the same action template, e.g., several chronicles being instantiations of the *Go* action template. In the current formulation, nothing distinguishes two action chronicles of the same template and any satisfying assignment can be made into a new one by exchanging the variables of the two chronicles.

Given the set of action templates \mathcal{T} , for any action template $a \in \mathcal{T}$, we refer to all action chronicles of the template a in Π as $\langle \mathcal{C}_a^1, \dots, \mathcal{C}_a^n \rangle$, in an arbitrary order.

From this ordering, we define two symmetry breaking constraints. The first one requires that, if a given chronicle is present, then all its predecessors in the ordering are present as well:

$$\bigwedge_{a \in \mathcal{T}} \bigwedge_{i \in [1, n-1]} \text{present}(\mathcal{C}_a^{i+1}) \implies \text{present}(\mathcal{C}_a^i)$$

The second one requires the ordering on chronicles to match the ordering of actions in the solution plan:

$$\bigwedge_{a \in \mathcal{T}} \bigwedge_{i \in [1, n-1]} \text{start}(\mathcal{C}_a^{i+1}) \geq \text{start}(\mathcal{C}_a^i)$$

where $\text{start}(\mathcal{C})$ is the start timepoint of the action chronicle \mathcal{C} .

4 Instantiation in a Domain-Independent Planner

We now describe how the presented encoding can be leveraged in a fully domain-independent temporal planner. We present LCP (Lifted Constraint Planner), a planner that solves ANML planning problems using an SMT solver as a backend. Rather than providing a fully fledged temporal planner, LCP aims at validating the effectiveness of our encoding on standard temporal planning benchmarks.

4.1 The ANML Language

The Action Notation Modeling Language (ANML) is a proposal by NASA Ames Research Center to represent rich temporal planning and scheduling problems [32]. ANML features a clear notion of actions with conditions and effects integrated with a rich representation of time.

Compared to the temporal variants of PDDL, the ANML language has two important features. The support of multi-valued state variables, as opposed to the boolean state variables of PDDL, allows for a more compact representation of the state. In addition, ANML supports referencing timepoints other than the start and end of an action, thus increasing the capabilities to model complex temporal actions. Temporal PDDL planning problems are usually easy to express in ANML, with some recent work to provide an automated translation [3, 5].

Translation of ANML planning problems into chronicles is straightforward as all constructs we are interested in have a one to one mapping [6].³ We parse an ANML problem file into the initial chronicle \mathcal{C}_0 and a set of action templates \mathcal{T} that can be instantiated into action chronicles.

³ Omitted in our translations are the hierarchical and resource constructs of ANML that are beyond the scope of this paper.

4.2 Solving with SMT

Our solving procedure works by incrementally generating bounded planning problems Π_k with an increasing number of optional actions. For each problem, an SMT solver is used to either prove the consistency of Π_k or the absence of a satisfying assignment. If Π_k is consistent, the solution plan is extracted from the found satisfying assignment. Otherwise the process continues with the next bounded problem Π_{k+1} .

The overall procedure is given in Algorithm 1. The planner iteratively generates bounded planning problems of increasing depth k until a solution is found or a depth k_{\max} is reached. Given an input planning problem $(\mathcal{C}_0, \mathcal{T})$, we generate a bounded problem Π_k using the procedure $\text{GENPROBLEM}(\mathcal{C}_0, \mathcal{T}, k)$ as follows:

$$\Pi_k = \{ \mathcal{C}_0 \} \cup \bigcup_{a \in \mathcal{T}} \{ \mathcal{C}_a^i \mid i \in [1, k] \}$$

For a problem Π_k , this formulation allows the presence of k instances of each action template $a \in \mathcal{T}$. The $\text{CHECKSMT}(\Pi_k)$ function encodes Π_k as a CSP using the encoding we presented. The consistency of the resulting CSP is checked with the Z3 SMT solver [16].

Algorithm 1. Planning procedure of LCP (Lifted Constraint Planner) for an input problem $(\mathcal{C}_0, \mathcal{T})$ and maximum depth k_{\max} .

```

function LCP( $\mathcal{C}_0, \mathcal{T}, k_{\max}$ )
   $k \leftarrow 0$ 
  while  $k \leq k_{\max}$  do
     $\Pi_k \leftarrow \text{GENPROBLEM}(\mathcal{C}_0, \mathcal{T}, k)$ 
     $model \leftarrow \text{CHECKSMT}(\Pi_k)$ 
    if  $model$  is a consistent model then
      return  $\text{EXTRACTSOLUTION}(model)$ 
    end if
     $k \leftarrow k + 1$ 
  end while
  return failure
end function

```

4.3 Limitations

As it stands, LCP is a rather naïve instantiation of the proposed encoding into a domain-independent planner. One of the limitation is the absence of reuse of the inconsistent models of previous steps when generating a new bounded planning problem. At the very least, one could analyze the unsatisfiable core provided by SMT solvers to infer which actions must be added to the next bounded planning problem.

More importantly, the use of SMT is certainly suboptimal given the progress made in constraint-based scheduling solvers in tackling closely related problems. Indeed, solvers such as CP Optimizer provide constructs for optional temporal intervals and state functions that closely relate to the building blocks of our encoding [26, 27]. However, as we further discuss in Sect. 6, there is still a mismatch between the available global constraints and the needs for domain-independent planning. In the absence of global constraints for CP solvers, SMT solvers provide a good backup solution for our encoding that features many disjunctive constraints.

LCP here simply aims at a preliminary evaluation of the proposed encoding against existing planners.

5 Experiments

5.1 Comparison with State of the Art Temporal Planners

We evaluate our approach against several state of the art temporal planners on benchmarks from the International Planning Competition (IPC). We focus on problems where time plays an important role in the solution: planning problems that have deadlines or time windows that constrain the occurrence time of actions in the plan. The rationale for doing so is that planning problems without such features can be solved without any explicit handling of time (with the exception of some problem with required concurrency [14]).

We compare ourselves to Temporal Fast Downward (TFD) [20], OPTIC [2], FAPE [6, 18] and SMTPlan+ [10]. TFD is a forward-search state-space planner that ranked second in the last temporal track of the IPC [33]. Its search is based on the addition of new primitive actions at the end of an existing plan and heavily relies on heuristics for guidance. OPTIC is similar in its use of forward-search but uses a different heuristic and a lifted temporal representation handled in an STN. It is an improved version of POPF [13], runner up in the temporal track of the penultimate IPC.

FAPE represents another line of temporal planners that uses a constraint-based representation. FAPE plans in the space of lifted partial plans, where each plan is composed of a set of partially instantiated actions whose variables are embedded in a CSP. Search proceeds by either imposing values on variables or extending the partial plan with additional actions, thus extending the CSP with new constraints, variables and values in previous domains. FAPE is the first planner in this line of work to show good performance in a domain-independent setting [6].

SMTPlan+ [10] is a recent planner that supports the full range of PDDL+ [22] through a compilation to SMT. Unlike LCP, its encoding is state-oriented with additional constructs to support non-linear continuous change. Given SMTPlan+'s usage of SMT to solve a constraint-based encoding, an additional, more detailed, comparison will be made in the next subsection.

We use the *airport-timewindows*, *satellite-timewindows* and *pipesworld-deadlines* domains from IPC4 which feature durative actions and temporal constraints on the plan in the form of deadlines and time-windows in which actions can be performed. The *airport-timewindows* domain focuses on planning the takeoff and landing of several planes in an airport, the movement of planes being temporally constrained by the arrival of other planes. The *satellite-timewindows* domain plans observations and data transmission of a fleet of satellites subject to visibility windows. The *pipesworld-deadlines* domain focuses on planning the transportation of products through pipes, subject to delivery deadlines. TFD, OPTIC and SMTPlan+ use the original PDDL domains while LCP and FAPE use their translation to ANML from FAPE’s benchmark problems.⁴

As standard in the IPC, we evaluate the planners on the number of problems solved with a 30 min timeout. Benchmarks are run on an Intel i5-7200U @ 2.50 GHz CPU with 8 Gb of RAM. LCP and SMTPlan+ use Z3 [16] in version 4.6.3 as a black-box SMT solver.

Results are given in Table 1. It can be seen that, on temporally constrained problems, LCP is slightly ahead of both FAPE and OPTIC in terms of number of problems solved. TFD performs poorly on such problems while SMTPlan+ fails to solve any problem.

LCP performs best in the *pipesworld-deadlines* domain which is characterized by much interference between the available actions, reducing the ability to reason independently on goals. Its worst performance is on the *satellite-timewindows* domain whose difficulty lies in the number of mostly independent goals requiring a large number of actions. In this setting, LCP fails to prove the absence of solution before reaching the subproblem that actually contains one. The *airport-timewindows* domain is an intermediate between those two domains making LCP performs only slightly better than other planners.

Table 1. Number of problems solved by the considered planners within 30 min. Best result is given in boldface.

	LCP	FAPE	OPTIC	TFD	SMTPlan+
<i>airport-timewindows</i>	8	7	7	1	0
<i>satellite-timewindows</i>	4	10	4	0	0
<i>pipesworld-deadlines</i>	17	6	13	2	0
Total	29	23	24	3	0

Note on Expressiveness. Unlike PDDL temporal planners, LCP supports rich temporal constructs, including actions with conditions and effects besides the start and end timepoints of actions. It leverages ANML’s multi-valued state variables (as opposed to the boolean state variables of PDDL) to provide a more

⁴ <https://github.com/laas/fape/tree/master/planning/domains>.

compact representation. Note that FAPE, which also uses the ANML language as input, has the same characteristics.

In addition, LCP readily supports numeric variables both as action parameters and state variables. State of the art temporal planners do not support numeric parameters due to their need to ground the problem to derive heuristics, even though some recent work was done in this direction for forward-search planners [30].

Note on Performance in Non-temporal Domains. We also tested the performance of LCP on non-temporal domains, i.e., domains where time is either absent or limited to the duration of actions and can soundly be omitted [14]. On the domains tested (*blocks*, *logistics* and *rovers* from the IPC), LCP solved around two thirds of the problems solved by FAPE, OPTIC and TFD who all had similar performance. As could be expected, LCP does not reach the performance of state of the art planners on non-temporal problems. It nevertheless showed consistent performance in its handling of causal relationships.

5.2 Comparison with SMTPlan+

Our work was motivated by the recent developments in the planning community to extend the expressiveness of task planners. Research in domain-independent planning has been mostly focused on ground state-based heuristic search. Such planners deliberately choose to cope with very large search spaces because it enables the definition of very effective heuristic functions to guide their exploration.

The will to support more realistic problems through extensions to PDDL is however problematic. Indeed extensions affect the performance of heuristics that become less informative, with dramatic effects on search performance. This state of affairs has motivated the introduction of new planners that rely on SMT solvers to deal with the most expressive extensions of PDDL. Most notably, the most effective approaches for planning with PDDL+ rely on compilations to SMT [9, 10].

SMTPlan+ [10] appears to be the most effective of those planners. It supports planning with continuous non-linear change and processes and the authors showed that those can be efficiently accounted for in an SMT representation.

Of all the planning benchmarks we tested it on, SMTPlan+ was only able to solve the two simplest instances of the (non-temporal) *blocks* domain. Its performance on domain-independent temporal or non-temporal planning is thus largely below the ones of LCP and state-of-the-art planners.

SMTPlan+ uses a state-oriented representation where each state is associated to a given happening that alters the state (corresponding to the start or end of an action). Each state is represented by a set of boolean variables. To each happening is associated the choice of a primitive action that implies constraints on the happening's state variables.

To understand the implication of this representation, we compare the encoding of both LCP and SMTPlan+ on two problems of the *rovers* domains. The two problems mainly differ by the number of objects in the domain, with a direct impact on the number of primitive actions. The first instance (p01) has only 63 primitive actions while the second instance considered (p10) has 382. Note that these numbers are low compared to standard planning benchmarks (e.g. the most difficult problem of the *rovers* domain has 32,437 reachable primitive actions).

The number of disjunctive constraints (i.e. clauses in the SMT formula) as well as the number of variables is given in Table 2 for both LCP and SMTPlan+. It can be seen that the size of the SMT formula at a given depth in SMTPlan+ is directly impacted by the number of primitive actions, making it quickly untractable even on problems of modest size. The size of the encoding grows linearly with the number of happenings (depth). On the other hand, LCP is mostly unaffected by the growth in problem size thanks to its use of partially instantiated actions. On the down side, the encoding of LCP grows quadratically with the depth. Note that, in the case of LCP, a depth of 4 means that each action template might be duplicated 4 times (in the *rovers* domain, which has 9 action templates, the plan at depth 4 might contain up to 36 actions). On the other hand, a durative action requires two happenings in SMTPlan+'s representation, i.e., a depth of 4 would only allow two sequenced actions.

Table 2. Number of disjunctive constraints (left) and variables (right) in the SMT formulas of LCP and SMTPlan+ at various depths. Depth is the number k of duplicated actions in LCP and the number of happenings in SMTPlan+.

Depth	Rovers p01		Rovers p10	
	LCP	SMTPlan+	LCP	SMTPlan+
1	109/115	1 987/332	165/132	28 760/1 828
2	295/218	3 987/664	399/235	59 293/3 656
3	561/321	6 161/996	713/338	89 196/5 484
4	907/424	8 248/1 328	1 107/441	119 099/7 312

6 Related Work

Encodings for Graph-Plan. A historical application of CP solvers to planning has been based on the graph-plan framework [8]. Such planners build a synthetic data structure that captures causal relationships between primitive actions up to a given plan length. While building such a data structure can be done in polynomial time, extracting a solution (or proving its absence) is combinatorial. CP solvers have been leveraged to perform the plan extraction step in planners such as GP-CSP [17] and CSP-PLAN [28]. Both planners use a grounded state-based encoding where the value of variables in each state is related to the previous state

and primitive actions through a set of constraints. Despite work on improving the encoding of the CSP with table constraints [1], the performance of such planners has been superseded by forward search planners.

Plan-Space and Timeline-Based Planners. Another line of research is the work on timeline-based planners [11, 12, 23] and lifted plan-space planners [6, 24]. Both kind of planners use a time-oriented representation, and the chronicle model that we use originated in lifted plan-space planners [24]. Those planners search in the space of partial plans where the current partial plan is extended with new actions during search. Key aspects of the partial plan are represented in a CSP, including the actions' parameters and timepoints.

One difficulty is that the CSP is constructed online – both constraints and variables are added to the CSP – limiting the ability to use existing constraint solvers. More problematic is that the set of effects that can be used to support a given condition is not fixed *a priori* as new actions can be inserted in the partial plan. As a result, the equivalent of our *supported-by* constraints are often implemented in an *ad hoc* way, outside of the main constraint engine.

The internal representation of those planners is however closely related to the encoding proposed in this paper. One important difference is that plan-space planners use *threats* to encode the consistency between support decisions. We avoid the explicit handling of threats by the introduction of a new timepoint in effect tokens to represent the minimal persistence of an effect which is accounted for directly in the coherence constraint. This is important to limit the size of the CSP, as an explicit encoding of threats would be cubic in the number of possible actions.

The proximity with those planners might make it possible to adapt the relaxations, heuristics and propagators developed into global constraints to our setting, such as the one tailored to reason on causal relationships in EUROPA and FAPE [4, 7].

SMT-based Planners. The development of SMT encodings for task planning in the last years has been motivated by the need to support richer planning problems, notably involving continuous change on numeric state variables in planners such as dReach [9] and SMTPlan+ [10]. As highlighted in Subsection 5.2, it is our feeling that the support of more complex problems has been done at the detriment of those planners scalability. Our objective with this paper is precisely to propose an alternative encoding that can (*i*) be used to efficiently represent planning problems and (*ii*) does not prevent its extension to problems with continuous change. Our encoding already supports continuous state variables and – unlike dReach and SMTPlan+ – action parameters with continuous domains. As for SMTPlan+ and dReach, our use of an SMT solver with non-linear arithmetic theories opens up the possibility of reasoning with non-linear continuous changes and processes.

Constraint-Based Scheduling. The consideration of optional activities in constraint solvers slowly narrows the gap that existed between planning and scheduling. Our formulation of a bounded planning problem is indeed very close to a scheduling problem with optional activities over given temporal intervals, in the spirit of those in CP Optimizer [26].

Achieving good, domain-independent, performance in a such a setting would certainly require specialized global constraints. This is a promising direction to tackle the current quadratic growth in the number of constraints of our representation.

Our coherence constraints are closely related to scheduling optional activities on unary resources and state-functions for which existing techniques could be easily adapted [26,27,35]. The main difference here being the implicit choice of the state variable that is governed by several variables. Support constraints are more challenging and to our knowledge have no straightforward mapping to global constraints in scheduling. Indeed, the closest equivalent (state-functions in CP Optimizer) lack the notions of conditions and effects. While plan-space planners do provide propagators for such support constraints [7,34], those are highly tailored to their search mechanism and adapting them would require more work.

7 Conclusion

In this paper, we presented a constraint-based encoding for temporal planning. The encoding is focused on handling the conditions and effects appearing in typical planning problems. It leverages a fully lifted representation and allows the addition of arbitrary constraints. These characteristics make it a good fit for a usage at the core of more general constraint-based planners or for embedding planning subproblems in CSPs.

Its usage in a simple planner, using an off-the-shelf SMT solver, shows improved performance with respect to state of the art temporal planners on planning problems with deadlines and time-windows. The resulting planner largely outperforms existing SMT-based temporal planners on planning benchmarks.

One important contribution is the identification the small discrepancies that remain between the available global constraints in constraint-based scheduling and the requirements for efficiently handling task planning problems; going one step closer to closing the long standing gap between planning and scheduling.

References

1. Barták, R., Toropila, D.: Reformulating constraint models for classical planning. In: International Florida Artificial Intelligence Research Society Conference (FLAIRS) (2008)
2. Benton, J., Coles, A., Coles, A.: Temporal planning with preferences and time-dependent continuous costs. In: International Conference on Automated Planning and Scheduling (ICAPS) (2012)
3. Bernardini, S., Fagnani, F., Smith, D.E.: Extracting lifted mutual exclusion invariants from temporal planning domains. *Artif. Intell.* **258**, 1–65 (2018)

4. Bernardini, S., Smith, D.E.: Developing Lfor EUROPA2. In: ICAPS Workshop on Heuristics and Search for Domain-Independent Planning (HSDIP) (2007)
5. Bernardini, S., Smith, D.E.: Automatic synthesis of temporal invariants. In: Symposium on Abstraction, Reformulation and Approximation (SARA) (2011)
6. Bit-Monnot, A.: Temporal and hierarchical models for planning and acting in robotics. Ph.D. thesis, Université de Toulouse (2016)
7. Bit-Monnot, A., Smith, D.E., Do, M.B.: Delete-free reachability analysis for temporal and hierarchical planning. In: European Conference on Artificial Intelligence (ECAI) (2016)
8. Blum, A.L., Furst, M.L.: Fast planning through planning graph analysis. *Artif. Intell.* **90**(1–2) (1997)
9. Bryce, D., Gao, S., Musliner, D., Goldman, R.: SMT-based nonlinear PDDL+ Planning. In: AAAI Conference on Artificial Intelligence (2015)
10. Cashmore, M., Fox, M., Long, D., Magazzeni, D.: A compilation of the full PDDL+ language into SMT. In: International Conference on Automated Planning and Scheduling (ICAPS) (2016)
11. Cesta, A., Cortellessa, G., Fratini, S., Oddi, A.: Developing an end-to-end planning application from a timeline representation framework. In: Innovative Applications of Artificial Intelligence Conference (IAAI) (2009)
12. Chien, S., et al.: ASPEN: automated planning and scheduling for space mission operations. In: International Conference on Space Operations (SpaceOps) (2000)
13. Coles, A., Coles, A., Fox, M., Long, D.: Forward-chaining partial-order planning. In: International Conference on Automated Planning and Scheduling (ICAPS) (2010)
14. Cushing, W., Kambhampati, S., Mausam, Weld, D.S.: When is temporal planning really temporal? In: International Joint Conference on Artificial Intelligence (IJCAI) (2007)
15. Cushing, W.A.: When is temporal planning really temporal? Ph.D. thesis, Arizona State University (2012)
16. de Moura, L., Björner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
17. Do, M.B., Kambhampati, S.: Solving planning-graph by compiling it into CSP. In: International Conference on Automated Planning and Scheduling (ICAPS) (2000)
18. Dvorák, F., Barták, R., Bit-Monnot, A., Ingrand, F., Ghallab, M.: Planning and acting with temporal and hierarchical decomposition models. In: IEEE International Conference on Tools with Artificial Intelligence (ICTAI) (2014)
19. Edelkamp, S., Hoffmann, J.: PDDL2.2: the language for the classical part of the 4th international planning competition. In: International Planning Competition (IPC-2004) (2004)
20. Eyerich, P., Mattmüller, R., Röger, G.: Using the context-enhanced additive heuristic for temporal and numeric planning. In: Prassler, E., et al. (eds.) Springer Tracts in Advanced Robotics (STAR), pp. 49–64. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-25116-0_6
21. Fox, M., Long, D.: PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)* **20** (2003)
22. Fox, M., Long, D.: Modelling mixed discrete-continuous domains for planning. *J. Artif. Intell. Res. (JAIR)* **27** (2006)
23. Frank, J., Jónsson, A.: Constraint-based attribute and interval planning. *Constraints* **8**(4) (2003)

24. Ghallab, M., Laruelle, H.: Representation and control in IxTeT, a temporal planner. In: International Conference on Artificial Intelligence Planning and Scheduling (AIPS) (1994)
25. Ghallab, M., Nau, D.S., Traverso, P.: Automated Planning: Theory and Practice (2004)
26. Laborie, P., Rogerie, J.: Reasoning with conditional time-intervals. In: International Florida Artificial Intelligence Research Society Conference (FLAIRS) (2008)
27. Laborie, P., Rogerie, J., Shaw, P., Vilím, P.: Reasoning with conditional time-intervals. Part II: an algebraical model for resources. In: International Florida Artificial Intelligence Research Society Conference (FLAIRS) (2009)
28. Lopez, A., Bacchus, F.: Generalizing graphplan by formulating planning as a CSP. In: International Joint Conference on Artificial Intelligence (IJCAI) (2003)
29. McDermott, D., et al.: PDDL: the Planning Domain Definition Language. Technical report (1998)
30. Savas, E., Fox, M., Long, D., Magazzeni, D.: Planning using actions with control parameters. In: European Conference on Artificial Intelligence (ECAI) (2016)
31. Scala, E., Ramirez, M., Haslum, P., Thiebaux, S.: Numeric planning with disjunctive global constraints via SMT. In: International Conference on Automated Planning and Scheduling (ICAPS) (2016)
32. Smith, D.E., Frank, J., Cushing, W.: The ANML language. In: International Conference on Automated Planning and Scheduling (ICAPS) (2008)
33. Vallati, M., Chrapa, L., Grześ, M., McCluskey, T.L., Roberts, M., Sanner, S., Managing Editor: The 2014 international planning competition: progress and trends. *AI Mag* **36**(3) (2015)
34. Vidal, V., Geffner, H.: Branching and pruning: an optimal temporal POCL planner based on constraint programming. *Artif. Intell.* **170**(3) (2006)
35. Vilím, P., Barták, R., Čepek, O.: Extension of $O(n \log n)$ filtering algorithms for the unary resource constraint to optional activities. *Constraints* **10**(4) (2005)



Decremental Consistency Checking of Temporal Constraints: Algorithms for the Point Algebra and the ORD-Horn Class

Massimo Bono and Alfonso Emilio Gerevini^(✉)

Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Brescia,
Via Branze 38, 25123 Brescia, Italy
{mbono,alfonso.gerevini}@unibs.it

Abstract. Deciding consistency of a set of temporal constraints (CSP) over either the Point Algebra (PA) or the Interval Algebra (IA) is a fundamental problem in qualitative temporal reasoning. Given an *inconsistent* temporal CSP and a sequence of constraint relaxations to perform, incrementally decided by the user or an application, decremental consistency checking is the problem of determining if the revised CSP becomes consistent after each relaxation. We propose new algorithms for decremental consistency checking of a CSP over either PA or the ORD-Horn subalgebra of IA. These techniques exploit a graph representation of the CSP and some data structures that are maintained at each relaxation. An experimental analysis shows that solving decremental consistency checking using our algorithms can be significantly faster than using existing static algorithms.

1 Introduction

Constraint-based qualitative temporal reasoning is a widely studied area of AI in which the most prominent approaches are Allen's Interval Algebra (IA) [1], Vilain and Kautz's Point Algebra (PA) [30]. PA and (the full or fragments of) IA are used in many contexts, including temporal reasoning, knowledge representation, planning, diagnosis, and spatial reasoning. Given a CSP of temporal constraints over either PA or IA, a fundamental reasoning problem is deciding its satisfiability, also called consistency checking of the CSP. This problem is NP-hard for the full IA, while it is polynomial for PA and for several fragments of IA, such as Nebel and Bürckert [26] ORD-Horn subalgebra, the unique maximal tractable subclass of IA containing all the basic relations.

In many applications of temporal reasoning, and constraint-based reasoning in general, we need to deal with an *inconsistent* (over-constrained) CSP (e.g., [12]) by performing some constraint relaxations in order to make the CSP consistent. For instance, an inconsistent CSP can be generated when merging qualitative temporal constraints that express beliefs, preferences or possibly noisy information from different sources (e.g., [6]), or that represent the temporal structure of different partially ordered plans in the context of plan merging (e.g., [32]).

As shown in our paper, finding a minimal set of constraints to relax in order to make an inconsistent temporal CSP consistent is NP-hard even for PA restricted to contain only the ‘<’ relation.

In the *decremental* version of consistency checking, we start from a non-empty inconsistent temporal CSP, and we iteratively *relax* (weaken) one of its constraints, possibly decided by the user or an application. After each constraint relaxation, we want to check if the revised CSP remains inconsistent, which is polynomial for CSPs over either PA or ORD-Horn. This process can be repeated until the temporal CSP becomes consistent. The incremental version of consistency checking is similar; starting from a consistent temporal CSP, we want to check if it remains consistent when constraints are made stronger. These decremental/incremental versions can be seen as dynamic polynomial problems, that have been largely investigated in the context of graph algorithms in order to efficiently maintain certain graph properties when edges are added/removed to the graph (e.g., [11, 14, 23]). Instead of applying a static algorithm to recompute the property at each graph revision, specialized algorithms and data structures are used to reduce the total computational effort of maintaining the property over a sequence of revisions. Dynamic polynomial algorithms have been studied also for some special classes of tractable binary CSPs over discrete variables [13].

While the static (e.g., [4, 9, 19, 22, 26]) and incremental (e.g., [8, 17]) versions of consistency checking have been widely studied for PA and tractable fragments of IA, efficient decremental consistency checking for these classes of constraints has not been addressed yet. In this paper, we study decremental consistency checking for CSPs over either PA or ORD-Horn. First we propose new decremental algorithms that exploit a graph representation of the CSP and some metadata that are maintained at each performed relaxation. Then our algorithms and the existing static algorithms, used in the context of decremental consistency checking, are experimentally evaluated. The results of this analysis show that substantial performance gains can be obtained by the new algorithms.

2 Background, Terminology and Notation

Vilain and Kautz’s Point Algebra (PA) [30] consists of three base relations between time points ($<$, $>$, $=$), all possible unions of them (\leq , \geq , \neq , \top , where \top is the universal relation), and of the empty relation (\perp). Allen’s Interval Algebra (IA) [1] consists of thirteen base relations between temporal intervals, all possible unions of these relations, and \perp .

In our context, a *temporal constraint satisfaction problem* (or briefly *temporal CSP*) is a set of constraints of the kind xRy , where x and y are either *point variables* or *interval variables*, and R is either a PA relation (if x and y are point variables) or an IA relation (otherwise). A temporal CSP will be denoted by Σ , if its constraints are over PA, by Ω if they are over IA, and by Θ if they are all over either PA or IA. Without loss of generality we assume that if $xRy \in \Theta$ then $yR^{-1}x \in \Theta$, with R^{-1} inverse of R , and that \top and \perp are not explicitly used in an input constraint.

Given a temporal CSP Θ over either PA or IA, a fundamental reasoning problem is deciding the *satisfiability* (or *consistency*) of Θ , where Θ is satisfiable (consistent) if and only if there exists an assignment of temporal values to the variables of Θ (rational numbers for point variables, pairs of rational numbers for interval variables) such that all the constraints in Θ are satisfied. In [17] the problem of deciding the satisfiability of Θ is called PSAT, if Θ is over PA, and ISAT if Θ is over IA.

While ISAT is NP-complete [30], PSAT can be solved in $O(n + c)$ time using van Beek's method [4], where n is the number of variables in the temporal CSP and c is the number of PA-constraints. Van Beek's techniques for PA uses a graph-based representation of the temporal CSP that in [16] is called *temporally labeled graph* (TL-graph).

A TL-graph $\mathcal{G} = (V, E)$ is a graph where each vertex in V represents a variable of the temporal CSP, and each edge in E represents a PA constraint in the CSP. Edges are either directed and labeled ' \leq ' or '<', or undirected and labeled ' \neq '. An edge between v and w labeled R is denoted by (v, R, w) . Every constraint $x = y$ is represented by the pair of edges (x, \leq, y) and (y, \leq, x) .

Given a temporal CSP Σ over PA represented by a TL-graph \mathcal{G} , van Beek's method solves PSAT for Σ by first computing the strongly connected components (SCCs) of \mathcal{G} considering only the directed edges in E . Σ is satisfiable if and only if \mathcal{G} has no edge labeled ' \neq ' or '<' connecting two vertices in the same SCC [4].

A *strongly connected component* (SCC) of \mathcal{G} is a maximal subset of vertices σ in \mathcal{G} such that $I(\sigma, \mathcal{G})$ is strongly connected, where: $I(\sigma, \mathcal{G})$ is the *graph induced* by σ in \mathcal{G} , i.e., the subgraph of \mathcal{G} with vertices σ and the subset of its directed edges connecting vertices in σ ; a directed graph is *strongly connected* if for each pair v, w of vertices it has a directed path from v to w [7]. The SCCs of \mathcal{G} can be computed in $O(|V| + |E|)$ time [29].

Given a TL-graph \mathcal{G} and an edge $(v, R, w) \in \mathcal{G}$ such that $R \in \{ '<', '\neq' \}$, we say that (v, R, w) is a *flaw* in \mathcal{G} if v and w belong to the same SCC of \mathcal{G} , and that a SCC σ of \mathcal{G} is *flawed* (*valid*) if $I(\sigma, \mathcal{G})$ contains (does not contain) a flawed edge, i.e., an edge labeled either '<' or ' \neq '. The flawed SCCs of a TL-graph can be computed in $O(|V| + |E|)$ time [4]. A valid SCC in \mathcal{G} represents an equivalence class of vertices (CSP variables) that should be interpreted with the same value (because the constraints between the variables of the SCC imply their equality [4]).

A TL-graph is *inconsistent* if it contains a flawed SCC. The CSP represented by a TL-graph \mathcal{G} is inconsistent if and only if \mathcal{G} is inconsistent [4, 16].

Several tractable fragments of IA have been identified [4, 10, 19, 21, 22], of which the most interesting and popular are the Simple Interval Algebra (SIA) [3, 22] and Nebel and Bürckert's ORD-Horn class [26]. SIA consists of the IA-relations that can be translated into a conjunction of PA-constraints between interval endpoints. Hence, ISAT for a temporal CSP over SIA can be reduced to PSAT for the PA-translation of the CSP. ORD-Horn subsumes SIA and is the unique maximal tractable sub-algebra of IA containing all the basic relations. Each constraint over ORD-Horn can be translated in constant time into an

equivalent set of disjunctions of PA-constraints called *ORD-Horn clauses* [26], where (1) each literal is of the form $p = q$, $p \leq q$ or $p \neq q$ (with p and q endpoints of an interval mentioned by the ORD-Horn constraint); (2) at most one literal is of type ‘=’ or ‘ \leq ’; (3) each disjunction is at most binary. $\pi(\Omega)$ denotes a set of ORD-Horn clauses translating Ω , called a *ORD-Horn clause translation* of Ω (this translation is not unique); $\pi_1(\Omega)$ denotes the CSP of PA-constraints formed by the unary clauses in $\pi(\Omega)$; finally, $\pi_2(\Omega)$ denotes the set of the binary clauses in $\pi(\Omega)$ ($\pi(\Omega) = \pi_1(\Omega) \cup \pi_2(\Omega)$).

An example of relation in ORD-Horn and not in SIA is the inequality of two intervals (i.e., all 13 base relations except ‘*equal*’ are possible). ISAT for a temporal CSP Ω over ORD-Horn can be decided in cubic time by using a path-consistency algorithm [26] and, when Ω is sparse, in quadratic time [17].

A *relaxation* of xRy is a constraint $xR'y$ such that $R' \supset R$.

3 Decremental Consistency Checking

An inconsistent temporal CSP can be made consistent by relaxing one or more constraints, but identifying a minimal set of relaxations making the CSP consistent is NP-hard. As shown in the proof of Theorem 1, NP-hardness holds even if every CSP constraint is of type either ‘<’ (for PA) or ‘*before*’ (for IA).

Theorem 1. *Given a temporal CSP Θ over either PA, SIA or ORD-Horn, computing a minimal set of relaxations making Θ consistent is NP-hard.*

Proof. Reduction from the Minimum Feedback Arc Set problem [20]. Given a directed graph $G = (V, E)$ we build a CSP $\Theta = \{(v < w) | (v, w) \in E\}$ and a *TL*-graph \mathcal{G} with vertices V and edges $\{(v, <, w) | (v, w) \in E\}$. If we remove an edge from G , then we relax $v < w$ to $v \top w$ and we remove edge $(v, <, w)$ from \mathcal{G} . Since Θ is consistent iff \mathcal{G} has no cycles [4], it is easy to see that G has a feedback arc set S of size k (i.e., G becomes acyclic without S) iff removing k edges from \mathcal{G} (relaxing k constraints in Θ) makes \mathcal{G} (Θ) consistent. A similar reduction can be constructed with Θ formed by constraints $\{I_v \text{ before } I_w | (v, w) \in E\}$ and \mathcal{G} constructed from $\pi_1(\Theta)$. NP-hardness follows.

Decremental consistency checking can be seen as a complementary approach to finding optimal relaxation sets, in which the user(s) or the application (e.g., a planner dealing with plan merging, or a system that searches for minimal conflict sets by iteratively relaxing constraints from a non minimal one [2]) specifies the relaxations to perform, and the system checks the consistency of the revised CSP after each relaxation.

Example. As a very simple example (illustrated in Fig. 1), consider organising the touristic activities for a trip of three tourists $T1$, $T2$ and $T3$. Some site visits can be performed only in the morning (Tower, Church and Museum), some only in the afternoon (Temple and Castle), and the others during all the day (Botanic Gardens). Some activities have a predefined order, such as Lunch, that should be in between the morning and the afternoon activities. Each tourist specifies a set of preferences about the order of some activities (not necessarily the same ones):

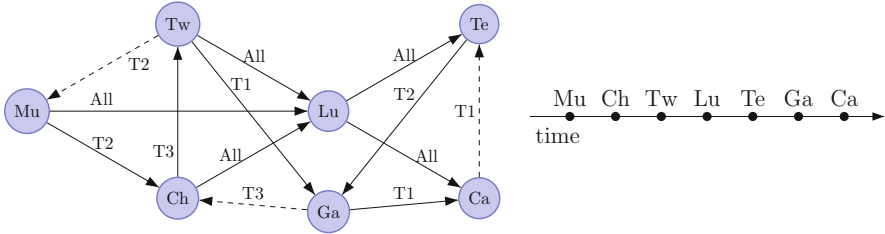


Fig. 1. CSP of precedence constraints for the touristic example and corresponding possible solution of the relaxed CSP. Graph vertices are activities to perform, and edges are precedence ($<$) constraints. Edges are labeled by the tourist(s) expressing the corresponding constraint (“All” for all tourists). Dashed edges indicate the relaxations specified by the three tourists.

$T1$: Tower before Garden, Garden before Castle, Castle before Temple;
 $T2$: Tower before Museum, Museum before Church, Temple before Gardens;
 $T3$: Gardens before Church, Church before Tower.

Although the set of constraints of every tourist is consistent, their union leads to an inconsistent CSP. In order to make it consistent, we can solve a decremental consistency checking problem where the relaxations are obtained by asking, in turn, each tourist to relax one of their personal constraints until the CSP becomes consistent. E.g., if $T1$ decides to remove “Castle before Temple”, then $T2$ removes “Tower before Museum”, and finally $T3$ removes “Gardens before Church”, a sequence of three relaxed CSPs is generated, the last of which is consistent.

More formally, we define the (dynamic) decremental version of consistency checking for a temporal CSP as follows:

Definition 1. Given an inconsistent temporal CSP Θ over a class of constraints \mathcal{C} and a sequence $\Theta_0, \dots, \Theta_k$ of CSPs over \mathcal{C} such that $\Theta = \Theta_0$ and Θ_i is obtained from Θ_{i-1} by making one constraint relaxation in Θ_{i-1} , for $i = 1, \dots, k$, **decremental consistency checking** is the problem of iteratively deciding the satisfiability of every Θ_i starting from Θ_1 until $i = k$ or Θ_i becomes consistent.¹

We call this problem D-PSAT, if the CSP is over PA, and D-OHSAT if it is over ORD-Horn.

3.1 An Algorithm for D-PSAT: DPASAT

Table 1 shows all possible kinds of relaxation in PA. Note that in the CSP $x > y$ and $x \geq y$ are assumed to be specified as $y < x$ and $y \leq x$, respectively, while $x = y$ as the pair $x \leq y$ and $y \leq x$. Without loss of generality, we also assume that every CSP Σ_i ($i > 0$) in the relaxed CSP sequence is represented by the corresponding TL-graph $\mathcal{G}_i = \mathcal{G}_{i-1} - \{(v, R, w)\} \cup \{(v, R', w)\}$ where $vR'w$ is the relaxation made in CSP Σ_{i-1} .

¹ If Θ_i become consistent all successor CSPs are consistent.

Table 1. Possible relaxations $xR'y$ of a constraint xRy in a CSP over PA with $R \in \{<, \leq, \neq\}$. Relaxing $x = y$ corresponds to relaxing $x \leq y$, $y \leq x$ or both.

R	\leq	$<$	$<$	$<$	\neq
R'	\top	\leq	\neq	\top	\top

For every relaxed CSP Σ_i , the performed relaxation $vR'w$ can make Σ_i (\mathcal{G}_i) consistent only if v and w belong to the same flawed SCC of the TL -graph of Σ_{i-1} . This important property is a direct consequence of the following lemmas.

Lemma 1. *Given an inconsistent TL -graph \mathcal{G} , revising the label of an edge contained in no induced graph of a SCC in \mathcal{G} leaves the revised TL -graph inconsistent.*

Proof. Since \mathcal{G} is inconsistent it contains at least a flawed SCC that is not altered by the revision. Inconsistency follows.

Lemma 2. *Given an inconsistent TL -graph \mathcal{G} and two vertices v, w in a valid SCC of \mathcal{G} , revising the label R of edge (v, R, w) in \mathcal{G} from R to R' with $R' \supset R$ leaves the revised TL -graph inconsistent.*

Proof. Same arguments as in the proof of Lemma 1.

Figure 2 shows the pseudocode of our algorithm for D-PSAT (DPASAT). The input is the TL -graph \mathcal{G} of the current (inconsistent) CSP Σ , the set S of flawed SCCs in \mathcal{G} , and a relaxation $vR'w$ of a constraint $vRw \in \Sigma$; the output is the TL -graph \mathcal{G}' of the revised input CSP and its flawed SCC set S' (possibly equal to S), if the revised CSP is inconsistent, **true** otherwise. The SCCs of the initial TL -graph \mathcal{G}_0 (input of the first run of DPASAT) can be computed by Tarjan's [29] or Sharir's [28] algorithms.

Algorithm DPASAT exploits Lemmas 1–2 and a simple data structure, that is maintained by the algorithm, storing for each flawed SCC σ the sets of pairs of its vertices that in \mathcal{G} are connected by edges labeled either ' $<$ ' or ' \neq ':

$$E_{<}[\sigma] = \{\langle v, w \rangle : v, w \in \sigma \text{ and } (v, <, w) \in \mathcal{G}\}$$

$$E_{\neq}[\sigma] = \{\langle v, w \rangle : v, w \in \sigma \text{ and } (v, \neq, w) \in \mathcal{G}\}.$$

Clearly a SCC σ is flawed *iff* $|E_{<}[\sigma]| + |E_{\neq}[\sigma]| > 0$, and \mathcal{G}' is inconsistent *iff* $|S'| > 0$.

After revising edge (v, R, w) in \mathcal{G}' according to relaxation $vR'w$, DPASAT checks if v and w belongs to the same SCC σ in S . If this is *not* the case, by Lemmas 1–2, \mathcal{G}' remains inconsistent and DPASAT outputs $\langle \mathcal{G}', S \rangle$ (line 3); otherwise S' is initialized to S , and each possible kind of relaxation is considered (see Table 1).

In every relaxation case, sets S' , $E_{<}$ and E_{\neq} are updated according to the kind of performed relaxation (lines 5–18). In some cases, this update (and

Algorithm 1. DPASAT**Input:**

- The TL -graph \mathcal{G} of an inconsistent CSP Σ over PA;
- The set S of flawed SCCs in \mathcal{G} ;
- A constraint relaxation $vR'w$ of a constraint $vRw \in \Sigma$ ($R' \supset R \in \{<, \leq, \neq\}$);

Output: true if $\Sigma' = \Sigma - \{vRw\} \cup \{vR'w\}$ is consistent; the pair $\langle \mathcal{G}', S' \rangle$, where \mathcal{G}' is the TL -graph of Σ' and S' is the set of flawed SCCs in \mathcal{G}' (possibly equal to S), otherwise.

```

1: if  $R' = \top$  then  $\mathcal{G}' \leftarrow \mathcal{G} - \{(v, R, w)\}$ 
2: else  $\mathcal{G}' \leftarrow \mathcal{G} - \{(v, R, w)\} \cup \{(v, R', w)\}$ 
3: if  $v$  and  $w$  are not both in a SCC of  $S$  then return  $\langle \mathcal{G}', S \rangle$ 
4:  $\sigma \leftarrow$  the SCC in  $S$  containing  $v$  and  $w$ ;  $S' \leftarrow S$ 
5: if  $R = \leq$  then  $\triangleright R = \leq, R' = \top$ 
6:   |  $S' \leftarrow (S' - \{\sigma\}) \cup \text{FlawedSCCs}(I(\sigma, \mathcal{G}'), v, w)$ 
7: else if  $R = <$  then  $\triangleright R = <, R' \in \{\leq, \neq, \top\}$ 
8:   | remove  $\langle v, w \rangle$  from  $E_{<}[\sigma]$ 
9:   | if  $R' = \neq$  then  $\triangleright R = <, R' = \neq$ 
10:  |   | add  $\langle v, w \rangle$  to  $E_{\neq}[\sigma]$ 
11:  |   | if  $|E_{<}[\sigma]| + |E_{\neq}[\sigma]| = 0$  then  $\triangleright R = <, R' \in \{\leq, \top\}$ 
12:  |   |   |  $S' \leftarrow S' - \{\sigma\}$ 
13:  |   |   | else if  $R' \in \{\neq, \top\}$  then  $\triangleright R = <, R' \in \{\neq, \top\}$ 
14:  |   |   |   |  $S' \leftarrow (S' - \{\sigma\}) \cup \text{FlawedSCCs}(I(\sigma, \mathcal{G}'), v, w)$ 
15: else  $\triangleright R = \neq, R' = \top$ 
16:   | remove  $\langle v, w \rangle$  from  $E_{\neq}[\sigma]$ 
17:   | if  $|E_{<}[\sigma]| + |E_{\neq}[\sigma]| = 0$  then
18:   |   |  $S' \leftarrow S' - \{\sigma\}$ 
19: if  $|S'| > 0$  then return  $\langle \mathcal{G}', S' \rangle$ 
20: else return true

```

Fig. 2. Pseudocode of algorithm DPASAT. We assume that every vertex of \mathcal{G} is marked with the SCC it belongs to, and every set $E_{<}[\sigma]$ and $E_{\neq}[\sigma]$ is decorated with its cardinality.

DPASAT) has constant time cost; other cases require more elaborated processing, since in \mathcal{G}' σ may be broken into multiple, smaller SCCs (see Fig. 4 for an example). After updating the algorithm metadata, consistency is decided by checking if $|S'| > 0$ holds (line 19). We now consider each relaxation case in turn.

$R = \leq$ and $R' = \top$ (lines 5–6). Since $I(\sigma, \mathcal{G}')$ may not be strongly connected anymore, we need to compute the possibly new SCCs in $I(\sigma, \mathcal{G}')$. If σ remains integral, we need to do nothing; otherwise we need to compute the $E_{<}$ and E_{\neq} sets of each flawed SCC in $I(\sigma, \mathcal{G}')$, remove σ from S' , and add the new flawed SCCs in $I(\sigma, \mathcal{G}')$ to S' . The computation of such SCCs and of the relative $E_{<}$ and E_{\neq} sets is performed by function `FlawedSCCs` (Fig. 3). This function first checks (line 2) if there is a directed path from v to w ; if this is the case, $I(\sigma, \mathcal{G}')$ is still strongly connected and σ is integral; so `FlawedSCCs` returns $\{\sigma\}$

Function 1: FlawedSCCs**Input:** A TL -graph \mathcal{F} and two vertices v and w in \mathcal{F} ;**Output:** The set of flawed SCCs in \mathcal{F} .

```

1:  $\sigma' \leftarrow$  set of vertices in  $\mathcal{F}$ 
2: if  $\mathcal{F}$  has a directed path from  $v$  to  $w$  then return  $\{\sigma'\}$ 
3: compute the set  $S_{\mathcal{F}}$  of all SCCs in  $\mathcal{F}$ 
4:  $S_{new} \leftarrow \emptyset$ 
5: foreach  $\sigma_{\mathcal{F}} \in S_{\mathcal{F}}$  do
6:   compute  $E_{<}[\sigma_{\mathcal{F}}]$  and  $E_{\neq}[\sigma_{\mathcal{F}}]$ 
7:   if  $|E_{<}[\sigma_{\mathcal{F}}]| + |E_{\neq}[\sigma_{\mathcal{F}}]| > 0$  then
8:      $S_{new} \leftarrow S_{new} \cup \{\sigma_{\mathcal{F}}\}$ 
9: return  $S_{new}$ 

```

Fig. 3. Pseudocode of **FlawedSCCs**. Step 6 can be efficiently computed by iterating over the vertex pairs $\langle x, y \rangle$ in sets $E_{<}[\sigma']$ and $E_{\neq}[\sigma']$ as follows: we add $\langle x, y \rangle$ to $E_{<}[\sigma_{\mathcal{F}}]$ if $(x, <, y) \in I(\sigma_{\mathcal{F}}, \mathcal{F})$; we add it to $E_{\neq}[\sigma_{\mathcal{F}}]$ if $(x, \neq, y) \in I(\sigma_{\mathcal{F}}, \mathcal{F})$.

and DPASAT algorithm does not change S' . Otherwise the metadata of DPASAT needs to be updated: **FlawedSCCs** computes the set S_{new} of new flawed SCCs and the $E_{<}$ and E_{\neq} sets for each of them (lines 5–8).² Then DPASAT discards σ from S' and it adds the SCCs of S_{new} to S' .

$R = '<'$ and $R' = '\leq'$ (lines 7–8, 11–12). Since $I(\sigma, \mathcal{G}')$ is still strongly connected, it suffices to remove $\langle v, w \rangle$ from $E_{<}[\sigma]$ and check if σ becomes a valid SCC, removing σ from S' if this is the case.

$R = '<'$ and $R' = '\top'$ (lines 8, 11–14). After relaxation, $I(\sigma, \mathcal{G}')$ may not be strongly connected anymore. First $\langle v, w \rangle$ is removed from $E_{<}[\sigma]$. Then, if edge $(v, <, w)$ was the only flaw in $I(\sigma, \mathcal{G})$, σ is removed from S' without checking if it has been broken or not, since there cannot be a flawed SCC in $I(\sigma, \mathcal{G}')$; otherwise, DPASAT's metadata are repaired as for case $R = '\leq'$, $R' = '\top'$.

$R = '<'$ and $R' = '\neq'$ (lines 7–10, 13–14). First $\langle v, w \rangle$ is moved from $E_{<}[\sigma]$ to $E_{\neq}[\sigma]$. Then, since $I(\sigma, \mathcal{G}')$ may not be strongly connected anymore, DPASAT's metadata are repaired as for case $R = '\leq'$, $R' = '\top'$.

$R = '\neq'$ and $R' = '\top'$ (lines 16–18). $I(\sigma, \mathcal{G}')$ remains strongly connected, but we need to check if σ is still flawed: $\langle v, w \rangle$ is removed from $E_{\neq}[\sigma]$ and, if it was the only flaw of σ , σ becomes a valid SCC and it is removed from S' .

Theorem 2. DPASAT solves D-PSAT.

Proof. (Sketch) By case analysis of every kind of relaxations we can show that DPASAT correctly updates its metadata (the set of flawed SCCs and all sets $E_{<}$ and E_{\neq}). Hence by Lemmas 1–2 and the fact that a CSP over PA is inconsistent iff its TL -graph contains a flawed SCC, at each run of DPASAT with input a CSP in the sequence of a D-PSAT instance, **true** is returned iff the CSP is consistent.

² Steps 2–3 of **FlawedSCCs** can be efficiently computed by a variant of Tarjan's algorithm for SCCs in which the first DFS starts from vertex v ; if during this search w is reached termination is forced; otherwise the algorithm continues in its normal way.

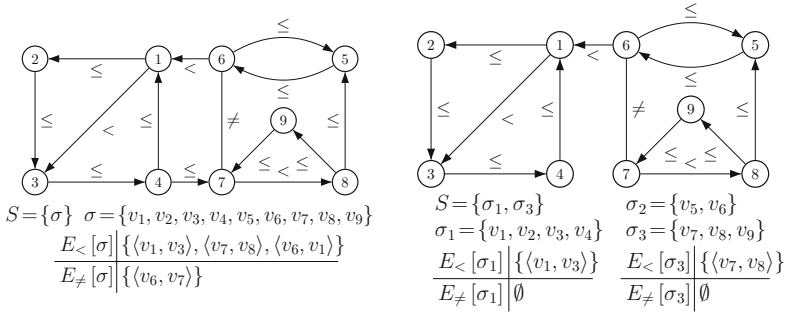


Fig. 4. An example where a flawed SCC σ of the TL -graph breaks due to relaxation $v_4 \top v_7$. In the left/right, the graph and its metadata before/after relaxation. After relaxation 3 new SCCs are created (σ_1 and σ_3 are flawed, σ_2 is valid) and the metadata are accordingly updated.

The worst-case time complexity of DPASAT is linear in the maximal-size subgraph of \mathcal{G} forming a flawed SCC. Depending on the SCC structure of the CSP (TL -graph \mathcal{G}), DPASAT’s complexity is better than the complexity of van Beek’s algorithm for the relaxed CSP (\mathcal{G}'), which is linear in the full \mathcal{G}' , i.e., $O(c + n)$ for c constraints and n variables in the CSP.

Theorem 3. *The time complexity of DPASAT is $O(c_f + n_f)$, where c_f/n_f is the number of constraints/variables represented by the largest induced graph of a flawed SCC in \mathcal{G} ; the space complexity of DPASAT is $O(c + n)$.*

Proof. (Sketch) For DPASAT the most expensive relaxations to process in the sequence of a D-PSAT instance are those requiring to run **FlawedSCCs**, whose complexity can be shown to be $O(e_f + n_f)$ by using Tarjan’s algorithm [29] for computing the SCCs of the the input graph \mathcal{F} , with e_f edges and n_f vertices, and computing the $E_<$ and E_{\neq} sets as described in Fig. 3. It is easy to see that DPASAT’s data structured require $O(c + n)$ space.

Moreover, DPASAT handles in constant time all relaxations $xR'y$ where (1) both the involved variables are *not* in the same flawed SCC, (2) R is ‘ \neq ’ and R' is ‘ \top ’, and (3) R is ‘ $<$ ’ and R' is ‘ \leq ’. Let k_0 the number of such relaxations in a D-PSAT instance. The complexity of using DPASAT for D-PSAT is as follows:

Theorem 4. *The time complexity of DPASAT for a sequence of k constraint relaxations is $O(k_0 + (k - k_0) \cdot (c_f + n_f))$, where c_f/n_f is the number of constraints/variables represented by the largest induced graph of a flawed SCC in \mathcal{G} .*

3.2 An Algorithm for D-OHSAT: DOHSAT

Our algorithm for D-OHSAT, DOHSAT, is an extension of a static method for ISAT over ORD-Horn called **ORDHORN-SAT** [17], which has $O(c \cdot n)$ time complexity for c constraints and n variables in the CSP. For lack of space the revised algorithm is described without detailed pseudocode.³

³ More details and pseudocode are available in a technical report.

Algorithm 2: ORDHORN-SAT**Input:** A set Ω of constraints over ORD-Horn;**Output:** true if Ω is satisfiable; false otherwise.

```

1:  $\Sigma_1 \leftarrow \pi_1(\Omega)$ ;  $\Sigma_2 \leftarrow \pi_2(\Omega)$ ;  $C \leftarrow \emptyset$ 
2: repeat
3:   if  $\Sigma_1$  is unsatisfiable then return false
4:    $C' \leftarrow$  SCCs of the  $TL$ -graph of  $\Sigma_1$ 
5:   if  $C' = C$  then return true
6:   foreach  $\delta$  in  $\Sigma_2$  and each disjunct  $p \neq q$  of  $\delta$  do
7:     if  $p$  and  $q$  are both in a component of  $C'$  then
8:       |   remove  $p \neq q$  from  $\delta$  in  $\Sigma_2$ 
9:     if  $\delta$  becomes empty clause then return false
10:   $D \leftarrow$  set of reduced disjunctions derived by steps 6–8
11:  if  $D$  is not empty then
12:    |    $\Sigma_1 \leftarrow \Sigma_1 \cup D$ ;  $\Sigma_2 \leftarrow \Sigma_2 - D$ ;  $C \leftarrow C'$ 
13: until  $D$  is empty
14: return true

```

Fig. 5. Pseudocode of ORDHORN-SAT.

Given a CSP Ω over ORD-Horn, ORDHORN-SAT processes the clause translation of Ω as described in Fig. 5. After checking the satisfiability of $\pi_1(\Omega)$, for each disjunct $p \neq q$ of each clause $\delta \in \pi_2(\Omega)$, if p and q both belong to a SCC of the TL -graph of Σ_1 (initially $\pi_1(\Omega)$), then δ is reduced by removing $p \neq q$ from the clause. If there is a clause that is reduced to the empty clause, then Ω is inconsistent. Otherwise Σ_1 is extended with the unary clauses (PA constraints) that have been generated, and the clauses that have been reduced are eliminated from Σ_2 (initialized with $\pi_2(\Omega)$). This reduction process (lines 3–12), that we call *reduction epoch*, is repeated until either an empty clause is generated, or no new unary clause is generated.

Each generated SCC is associated with an id (an integer). Each reduction epoch ρ can be represented by a pair $\langle D, \Delta \rangle$, where D is the set of the unary clauses (disjuncts) generated by ρ , and Δ is the set of ids of SCCs used to make the disjunction reductions. We call *reduction chain* the sequence of epochs generated by ORDHORN-SAT, and we denote it with $\mathcal{C}(\Omega)$. For any CSP Ω over ORD-Horn, it can be proved that the length of $\mathcal{C}(\Omega)$ is $O(n)$ [17].

The relaxation of a ORD-Horn relation into a relation in the same class corresponds to relaxing one or more PA constraints in $\pi_1(\Omega)$, either adding or removing one or more disjunctions in $\pi_2(\Omega)$, or performing a mix of these changes.⁴

Let Ω' be the CSP obtained by relaxing xRy into $xR'y$ in Ω . The two changes that we make to ORDHORN-SAT to use it for efficiently solving D-OHSAT are

⁴ We remind that every equality $x = y$ in Σ_1 is replaced by $x \leq y$ and $y \leq x$, and so relaxing $x = y$ to, e.g., $x \leq y$ corresponds to relax $y \leq x$ to $y \top x$. Moreover, if $x = y$ appears in a disjunction d of Σ_2 , d is replaced by two disjunctions where instead of $x = y$ we have $x \leq y$ and $y \leq x$, respectively.

- (1) using algorithm DPASAT to deal with the relaxations to perform in $\pi_1(\Omega)$, and
- (2) accelerating the computation of $\mathcal{C}(\Omega')$ by reusing $\mathcal{C}(\Omega)$.

In the following we focus on (2), assuming that $\pi_1(\Omega')$ is consistent (otherwise $\mathcal{C}(\Omega')$ is not computed). Let $\mu = \rho_1, \dots, \rho_{|\mathcal{C}(\Omega)|}$ the sequence of epochs forming $\mathcal{C}(\Omega)$, where $\rho_j = \langle D_j, \Delta_j \rangle$ ($j = 1, \dots, |\mathcal{C}(\Omega)|$). The main idea is to quickly identify a subchain $\mu' = \rho_1, \dots, \rho_h$ of μ that would be generated also for $\mathcal{C}(\Omega')$. If μ' exists, Σ_1 is extended with $\bigcup_{j=1}^h D_j$ and the corresponding disjunctions in $\pi_2(\Omega)$ are removed from Σ_2 . All new disjunctions generated by the relaxation, if any, are then added in Σ_2 . Then the algorithm continues from line 2 of ORDHORN-SAT building the reduction chain for Ω' .

There are only two kinds of relaxation $xR'y$ that can make an epoch ρ *invalid* for reuse in $\pi(\Omega')$, i.e., there is no guarantee that for $\pi(\Omega')$ the same SCCs used in ρ will be generated:

- (a) $\pi(xR'y)$ revises a constraint $p \leq q$ of $\pi(xRy)$ into $p \top q$;
- (b) $\pi(xR'y)$ removes a disjunction δ of $\pi(xRy)$ that is reduced in $\mathcal{C}(\Omega)$ ($\delta \notin \pi_2(\Omega')$).

The first case may invalidate ρ when it uses a SCC containing p and q ; the second case may invalidate ρ if the reduction of δ added a ' \leq ' constraint to Σ_1 generating in the TL -graph of Σ_1 a SCC used to make additional reductions in $\mathcal{C}(\Omega)$.

Let ρ_{inv} be the *earliest invalid epoch* of $\mathcal{C}(\Omega)$ among the invalid epochs determined by each relaxation in $\pi(xR'y)$ w.r.t. $\pi(xRy)$. We define μ' as the initial subchain of $\mathcal{C}(\Omega)$ up to ρ_{inv} (excluded). In order to efficiently represent $\mathcal{C}(\Omega)$ and quickly identify ρ_{inv} , we use and maintain at each relaxation a $O(n^2)$ space data structure called **SCCFusionMatrix** (abbreviated **SFM**). **SFM** represents the set of SCCs (C' in ORDHORN-SAT(Ω)) at each chain epoch. Its rows are indexed by the epochs of $\mathcal{C}(\Omega)$ and its columns by the graph vertices. $\mathbf{SFM}[i, j]$ is the id of the SCC to which vertex v_j belongs at epoch $i - 1$ (considering the TL -graph of $\pi_1(\Omega)$ for $i = 1$). Figure 6 shows an example of **SFM**.

For case (a), if $v \leq w$ is relaxed to $v \top w$ in $\pi_1(\Omega)$, the invalidated epoch is the first epoch ρ (if any) s.t. $\mathbf{SFM}[\rho, v] = \mathbf{SFM}[\rho, w]$ (v and w are in the same SCC) and SCC $\mathbf{SFM}[\rho, v]$ is used in ρ ($\mathbf{SFM}[\rho, v] \in \Delta_\rho$).⁵ For case (b), if the disjunction that the current relaxation in Ω removes was reduced at an epoch ρ' , generating a disjunct $v' \leq w'$ added to Σ_1 , then we consider invalid the first epoch after ρ' (if any) where v' and w' are in the same SCC σ , and σ is used in such an epoch.

DOHSAT's complexity for D-OHSAT can be stated taking account of the structure of the input CSP and of the type of relaxations. Let p_0 be the number of relaxations not modifying $\pi_1(\Omega)$ and p_1 the number of relaxations modifying it, if $\pi_1(\Omega)$ is inconsistent, and $p_0, p_1 = 0$ otherwise. The complexity is as follows:

Theorem 5. DOHSAT solves D-OHSAT in $O(p_0 + p_1 \cdot (c_f \cdot n_f) + (k - p_0 - p_1) \cdot (c \cdot n))$ time and $O(n^2)$ space, where c is the number of constraints, n the number of variables, k the number of constraint relaxations.

⁵ The second condition can be decided in constant time by storing for each epoch a binary hash table indexed by the SCC ids.

$10 \in \Delta_2$		v_1	v_2	v_3	v_4	v_5	v_6	...	$v_1 \leq v_6 \in \pi_1(xRy)$
	ρ_1	1	1	2	3	4	5	...	$v_2 \leq v_3 \in \pi_1(xRy)$
	ρ_2	10	10	10	11	12	13	...	$v_1 \top v_6 \in \pi_1(xR'y)$
	$v_2 \top v_3 \in \pi_1(xR'y)$
	$\rho_{ C(\Omega) }$	36	36	36	36	36	37	...	

Fig. 6. Example of SFM. The revision $v_1 \top v_6$ invalidates no epoch; $v_2 \top v_3$ invalidates ρ_2 since ρ_2 is the first epoch where v_2 and v_3 belong to the same SCC and Δ_2 contains the id of such SCC.

4 Experimental Results

We experimentally evaluated the performance of our algorithms using the existing static algorithms for PA and ORD-Horn as a baseline. All algorithms are implemented in C, and they share the same data structures as much as possible. All experiments were conducted on a 2.00GHz Core Intel(R) Xeon(R) CPU E5-2620. Each instance of D-PSAT and D-OHSAT in the used data sets (except those derived from plans) consists of a randomly generated inconsistent CSP and a sequence of random relaxations leaving every relaxed CSP, possibly except the last one, inconsistent. We considered different data sets according to the constraint density of the initial CSP and the policy defining the relaxations. Each *dense* CSP has a constraint for every pair of variables; each *sparse* CSP has $\lfloor n \cdot \log_2 n \rfloor$ constraints in total, where n is the number of variables. We considered two relaxation policies (the second only for D-PSAT):

RR: each relaxation involves randomly chosen variables;

RFSCC: each relaxation involves variables belonging to the same flawed SCC of the TL -graph representing the CSP.

The former is DPASAT/DOHSAT agnostic, the latter produces more challenging relaxations for DPASAT since the policy has no relaxation that is trivial to process for DPASAT. In the initial CSPs of the generated D-PSAT (D-OHSAT) instances, each constraint consists of a random PA (ORD-Horn) relation.

The compared algorithms are evaluated in terms of total CPU time over the full relaxation sequence and cumulative CPU time after each relaxation in the instance sequence. The data used to plot the results on the curves (in logarithmic scale with E-notation for Figures 7 and 8, in linear scale for Figs. 9 and 10) are always average values over 30 instances of D-PSAT or D-OHSAT.

As shown in Fig. 7, for every kind of considered benchmark, DPASAT clearly outperforms van Beek’s static algorithm (VB). While VB needs to compute the SCCs at every relaxation, DPASAT generates the SCCs from scratch only once, and then repairs the flawed ones when needed, which makes DPASAT about one order of magnitude faster than VB. DPASAT requires additional time (included in our results) for the first relaxation to initialize its metadata. However, such time is then regained when processing the remainder of the relaxation sequence.

Figures 8a.1 and a.2 compare DOHSAT, ORDHORN-SAT, and an efficient path consistency algorithm (PC) for solving D-OHSAT.⁶ DOHSAT is always signifi-

⁶ We tested Vilain et al.’s algorithm [31] optimized by using the full (precomputed) composition table of every IA relations.

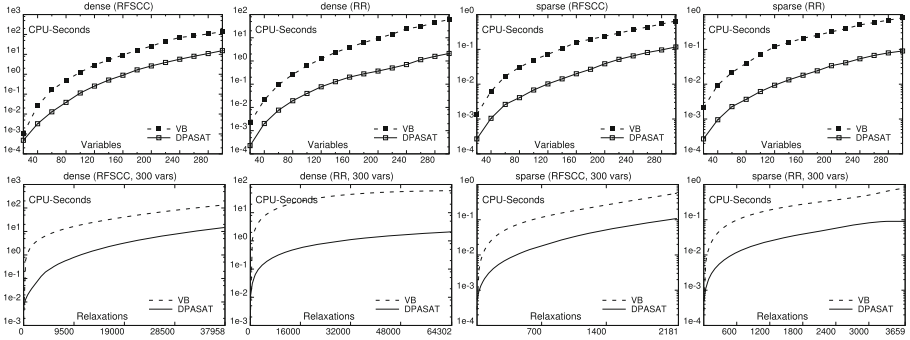


Fig. 7. Performance of DPASAT and van Beek’s algorithm (VB) for D-PSAT instances with sparse and dense CSPs, and relaxation policies RR and RFSCC. Top plots: average total time for processing instances with increasing number of variables; bottom plots: average cumulative time after processing each relaxation for instances with 300 variables.

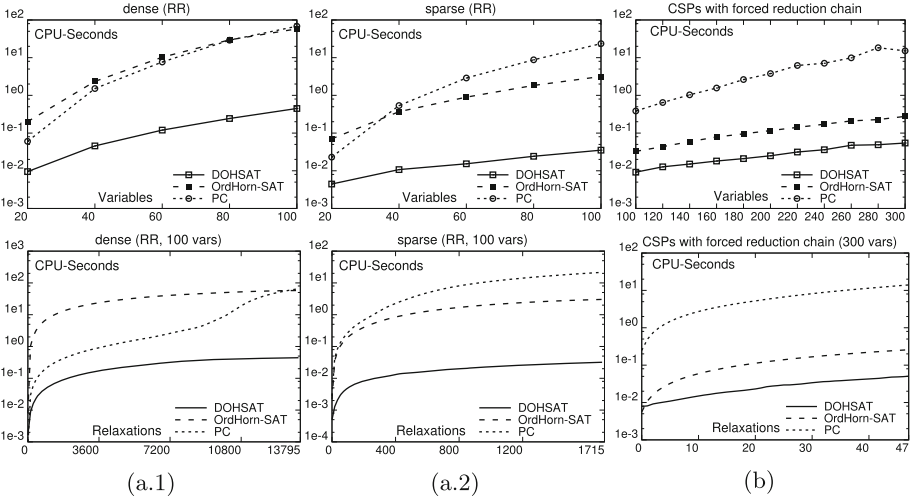


Fig. 8. Performance of DOHSAT and static algorithms for D-OHSAT instances. Columns (a.1)–(a.2) show results with dense and sparse CSPs, and relaxation policy RR; Column b) show results for the benchmark with forced non-empty reduction chains. Top plots: average total time for processing instances with increasing number of variables; bottom plots: average cumulative time after each relaxation for instances with 100 variables for sparse and dense CSPs, with 300 variables for non-empty reduction chains.

cantly faster than the other algorithms and, except for small CSPs, it performs between one and two orders of magnitude better. PC performs generally better than ORDHORN-SAT for dense CSPs, while ORDHORN-SAT performs better than PC for sparse CSPs (except for small ones).

By analyzing DOHSAT’s behaviour in the used benchmarks, we noticed that often either the inconsistency of a relaxed CSP Ω' is detected in $\pi_1(\Omega')$ by DPASAT, or $\pi_1(\Omega')$ is consistent but the generated reduction chain is very short. To better evaluate the usefulness of the reduction chains in DOHSAT and use D-OHSAT instances that are more challenging for our algorithm, we built an ad-hoc set of D-OHSAT instances in which: the initial CSPs are sparse, $\pi_1(\Omega')$ is always consistent, inconsistency of Ω' is always detected by DOHSAT at some epoch of $\mathcal{C}(\Omega')$, and every relaxation invalidates the reduction chain $\mathcal{C}(\Omega)$. The results for this benchmark are in Fig. 8b. The good performance of DOHSAT was confirmed also for this data set, obtaining improvements of almost one and three orders of magnitude relative to ORDHORN-SAT and PC, respectively.

When the CSP relaxations are performed “online” by users, it is likely that the relaxation sequence is relatively be short. Thus, we have tested our algorithms also using datasets with short relaxation sequences. In these benchmarks, for each sparse CSP at most 0.5% of the constraints are relaxed, while for each dense CSP the relaxations are at most 0.1% (less relaxations for dense CSPs because they have more constraints). As shown in the top plots of Fig. 9 and in Fig. 10, DPASAT and DOHSAT still perform very well with respect to the compared static algorithms.

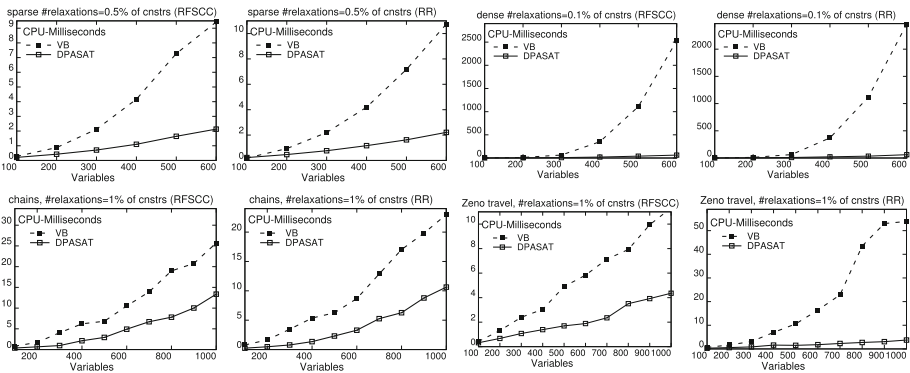


Fig. 9. Performance of DPASAT and van Beek’s algorithm (VB) for D-PSAT instances with short relaxation sequences. Top plots: random sparse and dense CSPs with relaxation policies RFSCC and RR. Bottom-left plots: chain-structured CSPs. Bottom-right plots: CSPs from the temporal structure of plans in domain Zenotravel. The plots show average total time (over 30 instances) for instances with increasing number of variables.

Finally, since real-world CSPs may have a structure that is missing in completely random CSPs, we considered two known benchmarks for structured (consistent) CSPs over PA: random CSPs with chain structure, used in [9], and CSPs

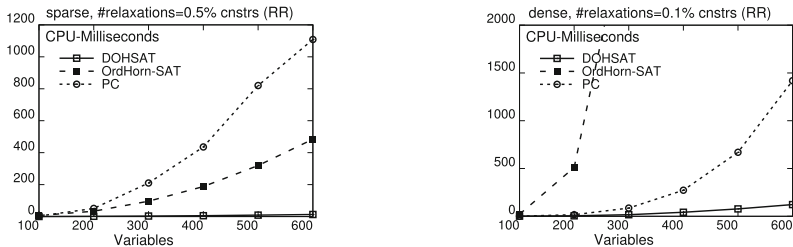


Fig. 10. Performance of DOHSAT and static algorithms for D-OHSAT instances (sparse and dense CSPs) with short relaxations sequences using relaxation policy RR. All plots show average total time (over 30 instances) for instances with increasing number of variables. The ORDHORN-SAT curve is pruned due to its very poor performances.

obtained from the action ordering constraints (of type ‘<’ and ‘≤’) of plans for the ZenoTravel domain [24,27], used in [18].⁷ Each CSP of these benchmarks has been made inconsistent by adding a number of < constraints equal to 2% of the original CSP constraints generating flawed SCCs in the TL -graph of the extended CSP. These CSPs are very sparse, and so we used relaxation sequences of size at most 1% of the constraints, corresponding to at most 15 relaxations for the largest chain-structured CSPs (1000 variables), and at most 12 relaxations for the largest ZenoTravel CSPs.

As shown by the plots at the bottom-right of Fig. 9, DPASAT performs significantly better than the compared static algorithm also for the considered structured CSPs with short relaxation sequences.

5 Conclusions and Future Work

The ability of handling inconsistent (over-constrained) CSPs by making constraint relaxations is an important task in applications using constraint-based reasoning. After showing that finding a minimal set of constraints to relax for making a temporal CSP consistent is NP-hard even when the CSP contains only <-constraints, we have introduced and addressed the decremental consistency checking problem for inconsistent temporal CSPs, proposing two new algorithms specialized for the well-known Point Algebra and ORD-Horn class. An experimental analysis shows that, for the considered benchmarks, our algorithms perform generally very well compared to the use of standard static algorithms.

Current and future work concern additional experiments and the study of decremental consistency for other classes of constraints in qualitative temporal reasoning, as well as in the context of qualitative spatial reasoning [5], where PA can be used for modelling constraints on the relative region sizes [15].

⁷ As in [9], Each chain-structured CSP corresponds to a random graph with 5 chains, each with $n/5$ vertices, $n/5$ transitive ‘<’ edges for the whole graph, $n/5$ cross-chain edges, and 10% of the input constraints are \neq constraints. The ZenoTravel plans were computed using planner SHOP2 [25].

References

1. Allen, J.: Maintaining knowledge about temporal intervals. *Commun. ACM* **26**(1), 832–843 (1983)
2. Bakker, R.R., Dikker, F., Tempelman, F., Wognum, P.M.: Diagnosing and solving over-determined constraint satisfaction problems. *IJCAI* **93**, 276–281 (1993)
3. van Beek, P.: Reasoning about qualitative temporal information. In: *Proceedings of the Eighth National Conference of the American Association for Artificial Intelligence (AAAI-1990)*, Boston, MA, pp. 728–734 (1990)
4. van Beek, P.: Reasoning about qualitative temporal information. *Artif. Intell.* **58**(1–3), 297–321 (1992)
5. Cohn, A.G., Renz, J.: Qualitative spatial representation and reasoning. In: *Handbook of Knowledge Representation*, pp. 551–596 (2008)
6. Condotta, J., Kaci, S., Schwind, N.: A framework for merging qualitative constraints networks. In: *Proceedings of the Twenty-First International Florida Artificial Intelligence*, pp. 586–591 (2008)
7. Cormen, T., Leiserson, C., Rivest, R.: *Introduction to Algorithms*. The MIT Press, Cambridge (1990)
8. Delgrande, J., Gupta, A.: Updating \leq , \prec -chains. *Inf. Process. Lett.* **83**(5), 261–268 (2002)
9. Delgrande, J., Gupta, A., Allen, T.: A comparison of point-based approaches to qualitative temporal reasoning. *Artif. Intell.* **131**(1–2), 135–170 (2001)
10. Drakengren, T., Jonsson, P.: Twenty-one large tractable subclasses of Allen’s algebra. *Artif. Intell.* **93**(1–2), 297–319 (1997)
11. Eppstein, D., Galil, Z., Italiano, G.: Dynamic graph algorithms. In: Atallah, M. (ed.) *Algorithms and Theory of Computation Handbook*. CRC Press, Boca Raton (1999)
12. Freuder, E.C., Wallace, R.J.: Partial constraint satisfaction. *Artif. Intell.* **58**(1–3), 21–70 (1992)
13. Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Dynamic algorithms for classes of constraint satisfaction problems. *Theor. Comput. Sci.* **259**(1–2), 287–305 (2001)
14. Georgiadis, L., Hansen, T.D., Italiano, G.F., Krinninger, S., Parotsidis, N.: Incremental data structures for connectivity and dominators in directed graphs. In: *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017*, pp. 42:1–42:15 (2017)
15. Gerevini, A., Renz, J.: Combining topological and size information for spatial reasoning. *Artif. Intell.* **137**, 1–42 (2002)
16. Gerevini, A., Schubert, L.: Efficient algorithms for qualitative reasoning about time. *Artif. Intell.* **74**, 207–248 (1995)
17. Gerevini, A.E.: Incremental qualitative temporal reasoning: algorithms for the point algebra and the ORD-Horn class. *Artif. Intell.* **166**(1–2), 37–80 (2005)
18. Gerevini, A.E., Saetti, A.: Computing the minimal relations in point-based qualitative temporal reasoning through metagraph closure. *Artif. Intell.* **175**(2), 556–585 (2011)
19. Golumbic, C., Shamir, R.: Complexity and algorithms for reasoning about time: a graph-theoretic approach. *J. Assoc. Comput. Mach. (ACM)* **40**(5), 1108–1133 (1993)
20. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) *Complexity of Computer Computations*, pp. 85–103. Springer, Boston (1972). <https://doi.org/10.1007/978-1-4684-2001-2.9>. Plenum

21. Krokhin, A., Jeavons, P., Jonsson, P.: The tractable subalgebras of Allen's interval algebra. *J. Assoc. Comput. Mach. (ACM)* **50**(5), 591–640 (2003)
22. Ladkin, P., Maddux, R.: On binary constraint problems. *J. Assoc. Comput. Mach. (ACM)* **41**(3), 435–469 (1994)
23. Łački, J.: Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Algorithms* **9**(3), 27:1–27:15 (2013)
24. Long, D., Fox, M.: The 3rd international planning competition: results and analysis. *J. Artif. Intell. Res.* **20**, 1–59 (2003)
25. Nau, D.S., Au, T.C., Ilghami, O., Kuter, U., Murdock, J.W., Wu, D., Yaman, F.: Shop2: an htn planning system. *J. Artif. Intell. Res.* **20**, 379–404 (2003)
26. Nebel, B., Bürckert, H.J.: Reasoning about temporal relations: a maximal tractable subclass of Allen's interval algebra. *J. Assoc. Comput. Mach. (ACM)* **42**(1), 43–66 (1995)
27. Penberthy, J.S.: Planning with continuous change (1993), Technical report UW-CSE-93-12-01
28. Sharir, M.: A strong-connectivity algorithm and its applications in data flow analysis. *Comput. Math. Appl.* **7**(1), 67–72 (1981)
29. Tarjan, R.: Depth first search and linear graph algorithms. *SIAM J. Comput.* **1**(2), 215–225 (1972)
30. Vilain, M., Kautz, H.: Constraint propagation algorithms for temporal reasoning. In: *Proceedings of the Fifth National Conference of the American Association for Artificial Intelligence (AAAI-1986)*, pp. 377–382. Morgan Kaufmann (1986)
31. Vilain, M., Kautz, H., van Beek, P.: Constraint propagation algorithms for temporal reasoning: a revised report. In: *Readings in Qualitative Reasoning about Physical Systems*, pp. 373–381. Morgan Kaufman, San Mateo (1990)
32. Yang, Q.: *Intelligent Planning*. Springer, Heidelberg (1997). <https://doi.org/10.1007/978-3-642-60618-2>



Domain Reduction for Valued Constraints by Generalising Methods from CSP

Martin C. Cooper¹(✉), Wafa Jguirim^{1,2}, and David A. Cohen³

¹ IRIT, University of Toulouse III, Toulouse, France
{cooper,Wafa.Jguirim}@irit.fr

² National School of Computer Science, University of Manouba, Manouba, Tunisia

³ Department of Computer Science, Royal Holloway, University of London, Egham, UK

d.cohen@rhul.ac.uk

Abstract. For classical CSPs, the absence of broken triangles on a pair of values allows the merging of these values without changing the satisfiability of the instance, giving experimentally verified reduction in search time. We generalise the notion of broken triangles to VCSPs to obtain a tractable value-merging rule which preserves the cost of a solution.

We then strengthen this value merging rule by using soft arc consistency to remove soft broken triangles and we show that the combined rule generalises known notions of domain value substitutability and interchangeability. Unfortunately the combined rules are no longer tractable to apply, but can still have applications as heuristics for reducing the search space.

Finally we consider the generalisation of another value-elimination rule for CSPs to binary VCSPs. This new rule properly generalises neighbourhood substitutability and so we expect it will also have practical applications.

Keywords: Valued Constraint Satisfaction Problem · Value merging
Value elimination · Tractability

1 Introduction, Notation and Definitions

Constraint Satisfaction (CSP) has had a significant impact on our ability to solve large and practical declarative problems, for example crew scheduling [14], and online workflow allocation [20]. However, in such complex problem domains, it has been restricted in its applicability by not being able to express degrees of satisfaction. The natural extension, valued constraint satisfaction (VCSP), allows us to express preferences amongst assignments and so forms a useful paradigm, extending the CSP to optimisation, whilst maintaining the declarative feel, allowing us to reason directly in problem domains.

Partially supported by ANR-11-LABX-0040-CIMI within the French *Agence Nationale de la Recherche* program ANR-11-IDEX-0002-02.

Since these problems are NP-hard much effort has been applied to find algorithms and techniques to reduce the search space. Such techniques are most effective if they do not change the set of possible solutions, but simply avoid exploring search avenues that we know cannot be productive. Key amongst these techniques are propagation and symmetry reduction, which can both be applied either before or during search. Search-space reduction has been approached in two ways: by the application of group theory [1, 3, 11, 19, 21] to identify equivalent branches of the search tree, or by using local structure to prune values or variables from the search [2, 6, 7, 9, 16, 18]. It is the latter approach that we continue to develop in this paper, extending domain reduction results from constraint satisfaction to valued constraint satisfaction [5, 17]. Indeed we will be combining domain reduction methods from consistency approaches [4, 8] which have been essential in making global constraints effective [15] with local pattern based value merging to make our techniques more widely applicable.

We observe that there are often several ways to extend reduction techniques from CSP to VCSP. This is most notably the case for arc consistency, which has been generalised to distinct techniques, FDAC, EDAC and VAC, which all coincide with CSP arc consistency when applied directly to the classical CSP, but correspond to distinct levels of soft consistency on the VCSP [8].

Similarly, different generalisations of neighbourhood substitution have been proposed for VCSPs [5, 13, 17], including a stronger condition for substitutability taking into account the current lower and upper bounds [12]. In this paper we consider generalisations from CSPs to VCSPs of value-elimination rules based on the merging of values [7, 18].

1.1 Definitions

A VCSP instance is a collection of cost functions applied to sets of variables [8].

For simplicity of presentation, in this paper we assume that costs are taken from the non-negative rationals together with the special cost infinity (∞). Of course, such instances are precisely equivalent to classical (crisp) CSP instances when the costs happen to all be either 0 or ∞ . For simplicity of notation we always assume that the set of variables of the instance I is $V(I) = \{X_1^I, \dots, X_n^I\}$, allowing us to use indexes to refer to variables. The domain of possible values for the variable X_i^I is denoted by D_i^I .

A subset of the variables is called a scope. An assignment to scope σ maps each $X_i^I \in \sigma$ to an element of its domain, D_i^I . The instance I includes a set of cost functions $C(I) = \{\phi_\sigma^I \mid \sigma \in S(I)\}$ where $S(I)$ is a set of scopes and each $\phi_\sigma^I \in C(I)$ maps assignments (to the variables σ) to costs. When we refer to a cost function ϕ_σ^I for which $\sigma \notin S(I)$ we always mean the cost function that is identically zero.

The cost of an assignment s to a set of variables $Y \subseteq V(I)$ is given by

$$cost_Y^I(s) = \sum_{\sigma \subseteq Y} \phi_\sigma^I(s|_\sigma).$$

A solution is an assignment to $V(I)$ that minimizes cost.

Where it improves readability we omit the name of the instance when referring to domains and costs. We simplify notation by using ϕ_i, ϕ_{ij} to denote $\phi_{\{X_i\}}$ and $\phi_{\{X_i, X_j\}}$ respectively. Furthermore, for single variables X_i we use the assignment on $\{X_i\}$ mapping X_i to a interchangeably with the value a , since the meaning is always clear from the context.

Analogously to consistency propagation in the CSP, any soft consistency operation on a VCSP instance [8] alters the cost functions while preserving all solutions. A soft arc consistency (SAC) operation at variable X_i replaces the unary cost ϕ_i with ϕ'_i different only at domain value $d \in D_i$ where $\phi'_i(d) = \phi_i(d) + \alpha$, (α may be negative). To compensate, it replaces one other cost function ϕ_σ for which $X_i \in \sigma$ with ϕ'_σ where $\phi'_\sigma(s) = \phi_\sigma(s)$ except when $s(X_i) = d$ in which case $\phi'_\sigma(s) = \phi_\sigma(s) - \alpha$. In order to be well defined the result of this SAC operation must leave all costs non-negative: we are not allowed to subtract a larger cost from any existing cost. To make this definition subsume (crisp) arc-consistency we extend the definition of subtraction so that $\infty - \infty = \infty$.

This is illustrated in Fig. 1, where as usual, variables are (grey) ellipses containing domain value: cost pairs and non-zero (binary) costs are shown with labelled arcs. A solution to this instance assigns c to X_1 , e to X_2 and f to X_3 and has cost 4.

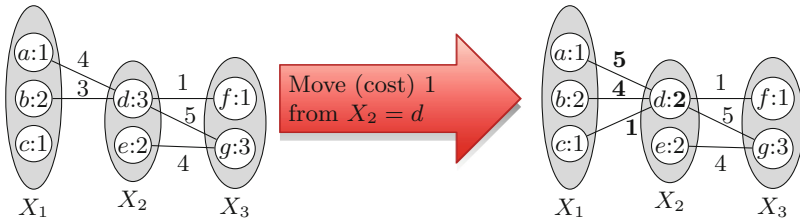


Fig. 1. SAC: moving cost 1 from $X_2 = d$ to $\phi_{1,2}$. Changed costs are highlighted.

Definition 1. In a VCSP instance, a GASBT (general-arity soft broken triangle) on values a, b for X_i is an assignment s to the union of distinct non-empty scopes σ and ρ , where $X_i \notin \sigma \cup \rho$, such that

$$\begin{aligned} \phi_{\sigma \cup \{X_i\}}(s|_\sigma \cup a) &< \phi_{\rho \cup \{X_i\}}(s|_\rho \cup b) \\ \phi_{\rho \cup \{X_i\}}(s|_\rho \cup a) &> \phi_{\sigma \cup \{X_i\}}(s|_\sigma \cup b) \\ \text{cost}_{\sigma \cup \rho}(s) &< \infty \end{aligned}$$

For binary CSP instances (where the costs lie in $\{0, \infty\}$), the notion of a general arity soft broken triangle coincides precisely with the classical CSP notion of broken triangle [7]. This correspondence is shown graphically in Fig. 2. In this figure the crisp CSP instance has dashed arcs to indicate disallowed tuples (cost = ∞) and solid arcs to indicate permitted tuples (cost = 0). Hence when σ and ρ each contain just one variable we say that a GASBT is a soft broken triangle (SBT).

Although the definition of a GASBT is independent of the unary costs $\phi_i(a), \phi_i(b)$, these costs will be critical in the definition of a value-merging rule.

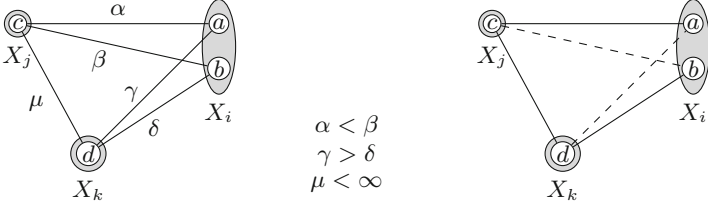


Fig. 2. A soft broken triangle and the corresponding crisp BTP pattern.

Definition 2. *The VCSP instance J obtained from I by merging a value pair $a, b \in D_i$ to produce a new value c has the same variables and domains as I except that $D_i^J = (D_i^I - \{a, b\}) \cup \{c\}$.*

The cost functions in J are defined as follows:

$$\phi_\sigma^J(t) = \begin{cases} \phi_\sigma^I(t) & \text{If } X_i \notin \sigma, \text{ or } t(X_i) \neq c \\ \min\{\phi_\sigma^I(t \cup a), \phi_\sigma^I(t \cup b)\} & \text{Otherwise.} \end{cases}$$

2 Value-Merging Rules

We say that $a, b \in D_i$ are *mergeable* in a VCSP instance if the cost of a solution to the new instance (after the merging of a and b) is identical to the cost of a solution to the original instance. We have the following rule.

Proposition 1. *Whenever $\phi_i(a) = \phi_i(b)$ and there is no general arity soft broken triangle on a, b , then $a, b \in D_i$ are mergeable. Furthermore, given a solution to the instance resulting from the merging of two values, we can find a solution to the original instance in linear time.*

Proof. Suppose that there is no GASBT on $a, b \in D_i$ in I , and let J be identical to I except that a, b have been merged to produce the value c . If a solution to J does not have value c at X_i^J then it is also a solution to I and we have nothing to prove. So, let s be a solution to J which assigns c to X_i^J . Denote by s_a, s_b the assignments in I which are identical to s except that X_i^I is assigned a or b (respectively).

It is clear from the definition of the cost functions in the merged instance that the cost of s in J is at most the minimum of the costs of s_a and s_b in I . If the cost of s is infinite, we have nothing to prove, so we assume it is finite.

Since there is no GASBT on a, b , and this is true for every pair of scopes σ and ρ , we must have either

$$\forall \sigma, \rho \text{ where } X_i^I \notin \sigma \cup \rho, \phi_{\sigma \cup \{X_i^I\}}^I(s|\sigma \cup a) \leq \phi_{\rho \cup \{X_i^I\}}^I(s|\rho \cup b)$$

or

$$\forall \sigma, \rho \text{ where } X_i^I \notin \sigma \cup \rho, \phi_{\rho \cup \{X_i^I\}}^I(s|\rho \cup a) \geq \phi_{\sigma \cup \{X_i^I\}}^I(s|\sigma \cup b)$$

Without loss of generality, suppose it is the former. Then

$$\forall \sigma, \text{ where } X_i^J \notin \sigma, \phi_{\sigma \cup \{X_i^J\}}^J(s|\sigma \cup c) = \phi_{\sigma \cup \{X_i^I\}}^I(s|\sigma \cup a)$$

Now, since $\phi_i^I(a) = \phi_i^I(b)$ we can replace c by a in s to obtain a solution to the original instance with the same global cost as s .

So, reconstructing a solution to I simply requires checking which of s_a or s_b is a solution to I . This can be achieved in time which is linear in the size of I .

We use the term SBT-merging (Soft Broken Triangle merging) for the merging of two values in a binary VCSP instance, allowed by the premise of Proposition 1.

2.1 Applying GASBTP Value Merging

For any $k \geq 1$, applying k -consistency operations until convergence to a CSP instance produces a unique closure [4]. Similarly, applying neighbourhood substitution operations [9] until convergence to a CSP instance produces a unique closure modulo isomorphism [5]. For VCSPs, finding the closure by soft arc consistency operation is not unique, but the problem of finding the best closure can be solved in polynomial time by linear programming [8]. It is therefore natural to ask the question of the uniqueness of and the complexity of finding the best closure of a VCSP instance under SBT merging operations. It turns out that the answer depends on whether the VCSP instance has infinite costs or not.

Theorem 1. *For a finite-valued VCSP (i.e. with no infinite costs), closure under GASBT-merging is unique up to value renaming. This closure can be found in polynomial time if the scopes are of bounded size. For general-valued VCSPs, maximizing the number of SBT-merges is NP-hard.*

Proof. For finite-valued VCSPs, it suffices to notice that GASBT-merging is equivalent to eliminating values $a \in D_i$ if $\exists b \in D_i$ such that $\forall \sigma, X_i \notin \sigma$ and for all assignments t to $\sigma \cup \{X_i\}$ we have that $\phi_{\sigma \cup \{X_i\}}(t \cup a) \geq \phi_{\sigma \cup \{X_i\}}(t \cup b)$. Clearly, elimination of such a value a cannot prevent eliminations at the same or other variables by the same rule. Of course we are free to name new domain values in any way that we choose, so we cannot guarantee that value merging ends up with precisely the same VCSP instance. However, it does not matter in which order we apply GASBT value merges to a finite-valued VCSP instance: we always converge to isomorphic instances.

Testing whether there is a GASBT on a pair of values at X_i for a given pair of scopes σ and ρ requires testing that a VCSP on $\sigma \cup \rho$ has finite value. In a finite valued VCSP this test is trivial. If either $\sigma \cup \{X_i\}$ or $\rho \cup \{X_i\}$ contains more variables than the bound on the size of a scope then the associated cost function is identically zero, so cannot satisfy the conditions required for a GASBT.

So, the existence of a GASBT value merge can be tested in polynomial time if we bound the arity of cost functions. Hence, the closure is unique modulo isomorphism and can be found in polynomial time by a greedy algorithm.

To show NP-hardness of optimal value merging when infinite costs are allowed observe that crisp binary CSP instances are precisely those binary VCSP instances with costs restricted to $\{0, \infty\}$. Since an SBT in such a binary VCSP is precisely a crisp broken triangle, SBT value merging is simply BT-merging in the corresponding CSP instance. It is known that finding the maximum number of BT-merges in a CSP instance is NP-hard (even for domains of size 3) [7]. It follows that finding the maximum number of SBT-merges in VCSPs is NP-hard, even if all costs belong to $\{0, \infty\}$, domains are of size 3, and scopes are binary.

3 Combining Soft Arc Consistency and SBT-Merging

GASBT value merging can only be performed when we have two domain values with identical unary costs. However, SAC operations allow us to move costs away from domain values. It is therefore possible that we can merge values, but only after performing the correct sequence of SAC operations. We will show two practical examples (with low complexity) where this does indeed occur and then show that, in general it is NP-hard to find such a sequence of SAC operations. In fact we will show, in the first result, that SAC followed by value merging subsumes a (natural but weak) form of valued neighbourhood substitution [5, 17].

Definition 3. *We say that a is weak neighbourhood substitutable for b at variable X_i if every cost associated with a tuple assigning value b to X_i is not made worse by substituting value a . That is,*

For all σ , $X_i \notin \sigma$ for all assignments t to σ we have

$$\phi_{\sigma \cup \{X_i\}}(t \cup a) \leq \phi_{\sigma \cup \{X_i\}}(t \cup b).$$

Example 1. Consider again the binary VCSP instance in Fig. 1. Before the SAC operation the two values for variable X_2 cannot be GASBT value merged as they have different unary costs. On the other hand, even before the SAC operation, e is weak neighbourhood substitutable for d .

Since there are no infinite costs in this VCSP, testing for value merging here amounts to checking that the costs associated with value d are always at least as high as those associated with value e .

Now consider the binary VCSP instance shown in Fig. 3. Since the value $\phi_{12}(a, c) < \phi_{12}(b, c)$ but $\phi_{12}(a, e) > \phi_{12}(b, e)$ neither a nor b can be weak neighbourhood substituted for the other. On the other hand a and b can be GASBT value merged since $\phi_{23}(c, e) = \infty$.

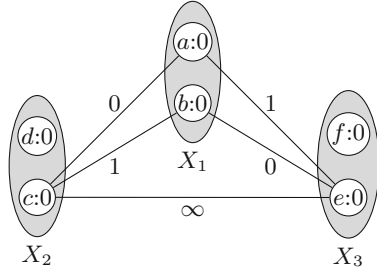


Fig. 3. In this binary VCSP instance a and b can be GASBT value merged, but they are not weak neighbourhood substitutable.

Notice that after the SAC operation the two values e and d in Fig. 1 can be GASBT merged. We now prove that this holds in general: weak substitutable values can always be GASBT merged after precisely one SAC operation.

Theorem 2. *If a is weak neighbourhood substitutable for b then, in polynomial time, we can find a SAC operation after which a and b can be GASBT value merged.*

Proof. Suppose that a is weak neighbourhood substitutable for b and that $\phi_i(a) < \phi_i(b)$. Choose any scope ρ and use a SAC operation to move the cost $\alpha = \phi_i(b) - \phi_i(a)$ from b to ϕ_ρ , where $X_i \in \rho$, replacing ϕ_i with ϕ'_i and ϕ_ρ with ϕ'_ρ and leaving all other cost functions unchanged.

Before the SAC operation we have, for all σ , $X_i \notin \sigma$ and for all assignments t to σ :

$$\phi_{\sigma \cup \{X_i\}}(t \cup a) \leq \phi_{\sigma \cup \{X_i\}}(t \cup b).$$

So, after the SAC operation, for all assignments t to $\rho - \{X_i\}$:

$$\begin{aligned} \phi'_\rho(t \cup b) &= \phi_\rho(t \cup b) + \alpha && \text{the SAC operation} \\ &> \phi_\sigma(t \cup b) && \text{since } \phi_i(a) < \phi_i(b) \\ &\geq \phi_\sigma(t \cup a) && \text{since } a \text{ is substitutable for } b \\ &= \phi'_\sigma(t \cup a). \end{aligned}$$

So the SAC operation preserves the weak neighbourhood substitutability, but now both a and b have equal unary cost at X_i .

We complete the proof by showing that if a is weak neighbourhood substitutable for b then there cannot be any GASBT on a and b . In fact this is trivially true since it can never occur for any scope ρ and assignment s that:

$$\phi_{\rho \cup \{X_i\}}(s|_\rho \cup a) > \phi_{\rho \cup \{X_i\}}(s|_\rho \cup b)$$

Since the SAC operation was entirely determined by the unary costs of a and b and σ was arbitrary this is polynomial time.

In fact we can identify another simple case in which we can immediately find a soft arc consistency operation which leads to GASBT-merging.

Definition 4. We say that $a, b \in D_i$ are almost interchangeable if there is a scope σ with $X_i \notin \sigma$ such that for all scopes $\rho \neq \sigma$ with $X_i \notin \rho$ we have, for every assignment t to ρ , $\phi_\rho(t \cup a) = \phi_\rho(t \cup b)$.

This definition of almost interchangeability is independent of the unary cost functions but still allows GASBT value merging after an appropriate SAC operation.

Proposition 2. If $a, b \in D_i$ are almost interchangeable, then, in polynomial time, we can find a SAC operation after which a and b can be GASBT merged.

Proof. We can simply make $\phi_i(a) = \phi_i(b)$ using a SAC operation which sends the difference in their cost to the cost function ϕ_σ . This leaves a and b almost interchangeable.

Since there is only one scope for which the cost functions can differ when a is replaced by b at X_i there can be no GASBT on a and b and they can now be merged.

When the costs lie in $\{0, \infty\}$, the notion of almost interchangeability coincides with the notion of virtual interchangeability [18].

Corollary 1. In a binary VCSP, suppose there is a variable X_j ($j \neq i$) such that $\forall k \notin \{i, j\}, \forall c \in D_k, \phi_{ik}(a, c) = \phi_{ik}(b, c)$ then a and b are almost interchangeable and can be merged.

Having shown the usefulness of applying SAC operations to remove any occurrences of GASBT we conclude the section with a proof that it is in general NP-hard to determine whether such SAC operations can be found.

Suppose that we have a binary VCSP instance I with a unary cost function ϕ_i for which $\phi_i(a) > \phi_i(b)$. Our problem is to find a set of costs $\{q_j \mid j \neq i\}$, one cost for each (other) variable X_j , such that by sending each cost q_j from $\phi_i(a)$ to the cost function ϕ_{ij} , we obtain an instance with $\phi_i(a) = \phi_i(b)$ that has no soft broken triangle on a and b at X_i .

We now prove that this problem is NP-hard.

Theorem 3. Given a VCSP instance I and variable X with domain values a and b , it is NP-hard to determine whether there are soft arc consistency operations on a and b which make the unary costs of a and b equal whilst also eliminating all SBT occurrences on a and b .

Proof. We provide a polynomial reduction from the problem SUBSETSUM which is well known to be NP-complete [1,10]. An instance $\langle S, M \rangle$ of SUBSETSUM consists of a set S of positive integers and an integer M . The corresponding question is whether there exists a subset $T \subseteq S$ whose elements sum to M , i.e.

$$\sum_{s \in T} s = M.$$

Given an instance $R = \langle S, M \rangle$ of SUBSETSUM we will construct a binary VCSP instance I_R such that there exists a set of SAC operations on two values a and b for variable X eliminating all SBT occurrences on a and b at X if and only if R is a yes instance.

Let $S = \{a_1, \dots, a_n\}$. Since we are showing hardness we need only reduce instances for which $n > 1$. The VCSP I_R has $2n + 1$ variables. The domain $D_{2n+1} = \{a, b\}$. All other domains are $\{0, \dots, 4\}$. The only non-trivial unary constraint is ϕ_{2n+1} where $\phi_{2n+1}(a) = M$ and $\phi_{2n+1}(b) = 0$.

The binary cost function between X_{2n+1} and any other variable depends on whether the index of that variable is even or odd. In all cases there is zero cost if X_{2n+1} is assigned value a . The full table of costs follows. For $i = 1, \dots, n$, $u \in \{0, \dots, 4\}$:

$$\begin{array}{ll} \phi_{2i-1,2n+1}(u, a) = 0 & \phi_{2i,2n+1}(u, a) = 0 \\ \phi_{2i-1,2n+1}(0, b) = 0 & \phi_{2i,2n+1}(0, b) = 0 \\ \phi_{2i-1,2n+1}(1, b) = 0 & \phi_{2i,2n+1}(1, b) = a_i/2 \\ \phi_{2i-1,2n+1}(2, b) = a_i/2 & \phi_{2i,2n+1}(2, b) = 0 \\ \phi_{2i-1,2n+1}(3, b) = a_i/2 & \phi_{2i,2n+1}(3, b) = M + 1 \\ \phi_{2i-1,2n+1}(4, b) = M + 1 & \phi_{2i,2n+1}(4, b) = a_i/2 \end{array}$$

For $i = 1, \dots, n$ the cost function $\phi_{2i-1,2i}$ is (crisp) equality. That is:

$$\forall u, v \in \{0, \dots, 4\}, \phi_{2i-1,2i}(u, v) = \begin{cases} 0 & \text{if } u = v \\ \infty & \text{otherwise} \end{cases}$$

All other (binary) cost functions only allow both variables to be zero. That is:

$$\forall u, v \in \{0, \dots, 4\}, \phi(u, v) = \begin{cases} 0 & \text{if } u = v = 0 \\ \infty & \text{otherwise} \end{cases}$$

Having defined the instance I_R we need to determine each cost q_i to move from the unary cost $\phi_{2n+1}(a)$ to the binary cost function between X_{2n+1} and X_i . After these SAC operations we require that there be no SBT on a and b at X_{2n+1} . We also require that the resultant unary costs of a and b at X_{2n+1} are equal. Hence $\sum_{i=1}^{2n} q_i = M$.

This latter condition induces a constraint on the allowed values of q_i and q_j for every non-infinite allowed pair of values on variables X_i and X_j .

Consider first the crisp constraints which force both X_j and X_k to have value 0. With this X_j and X_k a soft broken triangle can only occur for values $X_j = X_k = 0$. In this case we consider the four costs between values 0 at X_j and X_k and values a and b for X_{2n+1} . A broken triangle does not appear involving X_j and X_k precisely when:

$$(q_j \geq 0 \wedge q_k \geq 0) \vee (q_j \leq 0 \wedge q_k \leq 0) \tag{1}$$

Since $n > 2$ these constraints connect every variable except X_{2n+1} . It follows from Eq. (1) and the fact that $\sum_{i=1}^{2n} q_i = M > 0$, that each $q_i, i = 1 \dots, q_{2n}$ is non-negative and strictly less than $M + 1$.

Every other constraint that might be involved in a soft broken triangle is a strict equality between a pair of variables X_{2i-1} and X_{2i} . The five non-zero allowed pairs of values in each such constraint give the five following disjunctions.

$$(q_{2i-1} \geq 0 \wedge q_{2i} \geq 0) \vee (q_{2i-1} \leq 0 \wedge q_{2i} \leq 0) \quad (2)$$

$$(q_{2i-1} \geq 0 \wedge q_{2i} \geq a_i/2) \vee (q_{2i-1} \leq 0 \wedge q_{2i} \leq a_i/2) \quad (3)$$

$$(q_{2i-1} \geq a_i/2 \wedge q_{2i} \geq 0) \vee (q_{2i-1} \leq a_i/2 \wedge q_{2i} \leq 0) \quad (4)$$

$$(q_{2i-1} \geq a_i/2 \wedge q_{2i} \geq M + 1) \vee (q_{2i-1} \leq a_i/2 \wedge q_{2i} \leq M + 1) \quad (5)$$

$$(q_{2i-1} \geq M + 1 \wedge q_{2i} \geq a_i/2) \vee (q_{2i-1} \leq M + 1 \wedge q_{2i} \leq a_i/2) \quad (6)$$

Equation 2 is redundant. Equations 3–6 are equivalent to

$$\text{For } i = 1, \dots, n, \quad (q_{2i-1} = q_{2i} = 0) \vee (q_{2i-1} = q_{2i} = a_i/2) \quad (7)$$

Setting $a_i = q_{2i-1} + q_{2i}$, we can see that there exist q_1, \dots, q_{2n} satisfying the above equations if and only if there is a solution a_1, \dots, a_n to the SUBSETSUM instance R .

This reduction is clearly polynomial. Since SUBSETSUM is NP-complete, we can deduce that testing the existence of a set of soft arc consistency operations on a and b which makes their unary costs equal and eliminates all soft broken triangles, allowing us to apply Proposition 1 and merge a and b , is itself NP-hard.

4 Effect on Search-Tree Size of Merging

It has been shown [7] that in CSPs, BT-merging can increase the number of nodes in the search tree, when arc consistency is maintained during search.

Example 2. Consider an instance I with four boolean variables X_1, X_2, X_3, X_4 and the following constraints $X_1 \Rightarrow X_2, \overline{X_1} \Rightarrow X_3, X_2 \neq X_3, X_2 \neq X_4, X_3 \neq X_4$. The two domain values for X_1 can be merged.

A search which assigns $X_1 = 0$ and maintains arc consistency will detect inconsistency:

$$X_1 = 0 \rightarrow X_3 = 1 \rightarrow X_2 = 0 \rightarrow X_4 = 1$$

which wipes out the domain for X_3 .

On the other hand, after value merging there are no constraints involving X_1 and inconsistency will only be detected after another variable is instantiated.

However, it has been demonstrated experimentally that BT-merging applied during preprocessing reduces the average number of search-tree nodes by 27% [7].

We can make the following theoretical observation concerning naive (chronological) backtracking.

Proposition 3. *If a search algorithm is used which only prunes nodes based on the cost of the corresponding partial solution and instantiates variables in a fixed order, then merging values due to absence of GASBTs cannot increase the number of nodes visited.*

Proof. Suppose that there is no GASBT on $a, b \in D_i$ and that a, b have been merged to produce a new instance I' in which c is the result of the merge of a and b . Let Y be the variables assigned at some given node of the search tree, where $X_i \in Y$. Consider any assignment s_c to variables $Y \subseteq X$ in I' which assigns c to X_i . Denote by s_a, s_b the assignments which are identical to s_c except that X_i is now assigned a or b (respectively). There were no GASBTs on values a, b in the sub-problem on variables Y . So, from the proof of Proposition 1, we know that the cost of s_c on variables Y is $\min\{cost_Y(s_a), cost_Y(s_b)\}$. Hence, if pruning only depends on this cost, then s_c will only survive (i.e. the corresponding node will not be pruned) in I' if s_a or s_b survives in I . Thus, the total number of nodes cannot increase.

In the special case of binary CSPs, GASBTs are simply broken triangles, which gives us the following corollary.

Corollary 2. *If BT-merging is applied to a CSP then the number of nodes in the search tree cannot increase in a naive backtracking search.*

Of course we can expect that value merging due to the absence of GASBTs will often significantly reduce the number of search nodes visited as it is analogous to BT merging in the CSP, and hence the earlier experiments apply directly.

5 Soft Snakes

The broken triangle is just one example of a forbidden pattern which allows domain reduction. The general notion of forbidden pattern has led to the discovery of several novel value-elimination rules in binary CSPs [2, 6].

In this section we generalise the CSP pattern $\exists 2\text{snake}$ [2] to VCSPs. This is a further step towards identifying and classifying the generalisation of CSP value-elimination patterns to VCSPs. For simplicity, we concentrate on binary VCSPs, since no general-arity version of $\exists 2\text{snake}$ has yet been proposed for CSPs.

However, we are very pleased to be able to generalise the snake pattern since it is significantly stronger than GASBT value merging: snake-substitution is a generalisation of a strong form of soft neighbourhood substitution (and hence of weak neighbourhood substitution). Given its relatively low complexity (See Proposition 5) we expect that value merging due the absence of the valued snake pattern will be even more effective than that obtained by the absence of (soft) broken triangles. We first give the definition of soft neighbourhood substitution [5, 13, 17].

Definition 5. In a binary VCSP instance I , value $a \in D_i$ is soft neighbourhood substitutable for $b \in D_i$ if

$$\phi_i(b) - \phi_i(a) + \sum_{\substack{j \neq i \\ c \in D_j}} \min(\phi_{ij}(b, c) - \phi_{ij}(a, c)) \geq 0.$$

Soft neighbourhood substitutability of $a \in D_i$ for $b \in D_i$ implies that b can be replaced in any solution by a . It depends on the costs $\phi_{ij}(a, c)$ and $\phi_{ij}(b, c)$ for all variables X_j . Snake-substitutability extends this by looking at a third variable X_k while allowing a substitution of value $c \in D_j$ by a value $d \in D_k$.

Definition 6. In a binary VCSP instance I , value $a \in D_i$ is snake-substitutable for $b \in D_i$ if

$$\phi_i(b) - \phi_i(a) + \sum_{j \neq i} \min_{c \in D_j} (\max_{d \in D_j} f_{ij}(a, b, c, d)) \geq 0$$

where

$$\begin{aligned} f_{ij}(a, b, c, d) = & \phi_{ij}(b, c) - \phi_{ij}(a, d) + \phi_j(c) - \phi_j(d) \\ & + \sum_{k \neq i, j} \min_{e \in D_k} (\phi_{jk}(c, e) - \phi_{jk}(d, e)). \end{aligned}$$

The sum in the definition of $f_{ij}(a, b, c, d)$ is the minimum reduction in cost we obtain by replacing c by d (as assignments to variable X_j) in the sub-instance on variables $X_1, \dots, X_{i-1}, X_{i+1}, \dots, X_n$. We can see that snake-substitutability subsumes soft neighbourhood substitutability by setting $c = d$ in Definition 6.

Example 3. Consider a 2-variable instance of MAX-CSP over boolean domains $\{0, 1\}$ with a single constraint $X_1 \neq X_2$. In this case, the sum in the definition of $f_{12}(a, b, c, d)$ is zero since there is no third variable X_k such that $k \neq 1, 2$. No values are soft neighbourhood substitutable in this instance, but 1 is snake-substitutable for 0 in the domain of X_1 since for any assignment $(0, c)$ to (X_1, X_2) there is an assignment $(1, d)$ of no greater cost.

Proposition 4. Snake-substitutability is a valid value-elimination rule in binary VCSPs.

Proof. Consider a binary VCSP instance I in which $a \neq b \in D_n$ and a is snake-substitutable for b . We can assume that I has a solution s_b with $s_b[X_n] = b$ and $\text{cost}(s_b) \neq \infty$ otherwise there is nothing to prove.

For each $j \neq n$ let

$$d_j = \arg \max_d (f_{nj}(a, b, s_b[X_j], d)).$$

Thus, d_j is, in some sense, the best replacement for $s_b[X_j]$ when we replace b with a at X_n .

Now substitute b for a at X_n in the solution s_b , while also changing assignments to each variable X_j to d_j ($j \neq n$), to obtain the assignment s_a . We only need to show that $cost(s_a) \leq cost(s_b)$, since then s_a is a solution.

Since a is snake substitutable for b at X_n we can use the positivity and finiteness of the expression in Definition 6 to establish that some terms are finite. Finiteness will allow us to use simple subtraction later in the proof.

In Definition 6 we could choose c and e to be the values assigned by s_b to variables X_j and X_k . Finiteness of $cost(s_b)$ then implies that all terms occurring positively in Definition 6 are finite. This in turn implies that all terms occurring negatively in Definition 6 are also finite. So we can see that, for each $j \neq n$ and for each $k \neq n, j$, the following are all finite:

$$\phi_n(a), \phi_{nj}(a, d_j), \phi_j(d_j), \phi_{jk}(d_j, s_b[X_k])$$

We now define n intermediate solutions s_r between s_b and s_a . For $r = 0, \dots, n - 1$,

$$\begin{aligned} s_r[X_j] &= d_j \quad (j = 1, \dots, r) \\ s_r[X_j] &= s_b[X_j] \quad (j = r + 1, \dots, n - 1) \\ s_r[X_n] &= a \end{aligned}$$

Thus, in particular, $s_{n-1} = s_a$ and s_0 is identical to s_b except that variable X_n is assigned the value a .

Let $cost_{n-1}(s)$ denote the cost of assignment s on variables X_1, \dots, X_{n-1} :

$$cost_{n-1}(s) = \sum_{j=1}^{n-1} \phi_j(s[X_j]) + \sum_{j=1}^{n-1} \sum_{k=j+1}^{n-1} \phi_{jk}(s[X_j], s[X_k]).$$

From the definition of $cost_{n-1}$, we have:

$$cost(s_b) = cost_{n-1}(s_0) + \phi_n(b) + \sum_{r=1}^{n-1} \phi_{rn}(s_b[X_r], b) \quad (8)$$

Then, from the definition of $cost_{n-1}$ and s_r ($r = 0, \dots, n - 1$), and using the finiteness of $\phi_{rk}(d_r, s_r[X_k])$ for $k \neq r, n$ and $\phi_r(d_r)$, we have: for $r = 1, \dots, n - 1$:

$$\begin{aligned} cost_{n-1}(s_{r-1}) &= cost_{n-1}(s_r) + \sum_{k \neq n, r} (\phi_{rk}(s_b[X_r], s_r[X_k]) - \phi_{rk}(d_r, s_r[X_k])) \\ &\quad + \phi_r(s_b[X_r]) - \phi_r(d_r) \end{aligned} \quad (9)$$

We also have

$$cost(s_a) = cost(s_{n-1}) = cost_{n-1}(s_{n-1}) + \phi_n(a) + \sum_{r=1}^{n-1} \phi_{rn}(d_r, a). \quad (10)$$

We also know that $\phi_n(a) + \sum_{r=1}^{n-1} \phi_{rn}(d_r, a)$ is finite. This allows us to rewrite Eq. (10) as

$$\text{cost}_{n-1}(s_{n-1}) = \text{cost}(s_a) - \phi_n(a) - \sum_{r=1}^{n-1} \phi_{rn}(d_r, a) \quad (11)$$

By successive substitutions of Eq. (9) ($r = 1, \dots, n-1$) and then Eq. (11) in Eq. (8), we obtain

$$\begin{aligned} \text{cost}(s_b) &= \text{cost}(s_a) + \phi_n(b) - \phi_n(a) + \sum_{r=1}^{n-1} \{\phi_{rn}(s_b[X_r], b) - \phi_{rn}(d_r, a) \\ &\quad + \phi_r(s_b[X_r]) - \phi_r(d_r) + \sum_{k \neq n, r} (\phi_{rk}(s_b[X_r], s_r[X_k]) - \phi_{rk}(d_r, s_r[X_k]))\} \\ &\geq \text{cost}(s_a) + \phi_n(b) - \phi_n(a) + \sum_{r=1}^{n-1} \{\phi_{rn}(s_b[X_r], b) - \phi_{rn}(d_r, a) \\ &\quad + \phi_r(s_b[X_r]) - \phi_r(d_r) + \sum_{k \neq n, r} \min_{e \in D_k} (\phi_{rk}(s_b[X_r], e) - \phi_{rk}(d_r, e))\} \\ &= \text{cost}(s_a) + \phi_n(b) - \phi_n(a) + \sum_{r=1}^{n-1} f_{nr}(a, b, s_b[X_r], d_r) \\ &= \text{cost}(s_a) + \phi_n(b) - \phi_n(a) + \sum_{r=1}^{n-1} \max_{d \in D_r} f_{nr}(a, b, s_b[X_r], d) \\ &\geq \text{cost}(s_a) + \phi_n(b) - \phi_n(a) + \sum_{r=1}^{n-1} \min_{c \in D_r} \max_{d \in D_r} f_{nr}(a, b, c, d) \\ &\geq \text{cost}(s_a) \end{aligned}$$

by definition of snake substitutability. Hence, given any solution which assigns b to X_n , we can find a solution which assigns a to X_n .

Example 4. Consider a binary VCSP instance I in which there is a crisp equality constraint $X_h = X_i$ between variables X_h and X_i which have identical domains and $|D_i| > 1$. Clearly, we could merge these two variables to form a single variable Y . Suppose that a is (weak) neighbourhood substitutable for b at Y in this new variable-merged instance \bar{I} . However, because of the crisp equality constraint, a is not (weak) neighbourhood substitutable for b in I . Nonetheless, a is snake-substitutable for b in I . To see this, for all $j \neq i, h$, set $d = c$ and for $j = h$, set $d = a$ in Definition 6. Then the snake-substitutability of a for b in I follows from the (weak) neighbourhood substitutability of a for b in \bar{I} .

We now analyse the computational complexity of checking that no values are snake substitutable (according to Definition 6). Direct application of the definition leads to a time complexity of $O(n^3 d^3 + n^2 d^4)$ and a space complexity of $O(n^2 d^2)$. However, we can improve this complexity in the case of finite costs.

In this case, we can rewrite the definition of $f_{ij}(a, b, c, d)$ as follows:

$$\begin{aligned}
f_{ij}(a, b, c, d) &= \phi_{ij}(b, c) - \phi_{ij}(a, d) + \phi_r(c) - \phi_r(d) \\
&\quad + \sum_{\substack{e \in D_k \\ k \neq i, j}} \min(\phi_{jk}(c, e) - \phi_{jk}(d, e)) \\
&= \phi_{ij}(b, c) - \phi_{ij}(a, d) + \phi_r(c) - \phi_r(d) \\
&\quad - \min_{e \in D_i}(\phi_{ji}(c, e) - \phi_{ji}(d, e)) + \sum_{\substack{e \in D_k \\ k \neq j}} \min(\phi_{jk}(c, e) - \phi_{jk}(d, e))
\end{aligned}$$

which allows us to check that no values are snake substitutable in time complexity $O(n^2 d^4)$.

Proposition 5. *In finite-valued VCSPs, snake substitutable values can be found in time complexity $O(n^2 d^4)$.*

It is worth pointing out that this is only a factor of d greater than the complexity of checking that no soft neighbourhood substitutions are possible [5].

6 Conclusion

We have extended the notion of broken triangle from CSPs to VCSPs. This has allowed us to define a valid domain-reduction operation based on value-merging. We have extended the usefulness of this pattern by considering SAC operations that might enable us to perform extra reductions. In each case we have briefly considered the complexity of the reduction. We have shown that it is NP-hard to determine whether there exists some set of SAC operations that can allow us to perform extra GASBT reductions. However this is a reduction from SUBSETSUM and it is well known that there are pseudo-polynomial algorithms that solve this problem. It is an open question to determine whether there is a pseudo-polynomial algorithm for finding a set of SAC operations that allow us to merge values. It seems unlikely as this problem is in fact a quadratic optimisation problem.

In the final section we considered another domain-reduction operation called snake-substitutability which is a strong generalisation of neighbourhood substitutability in the case of binary VCSPs. From a theoretical point of view, SBT-merging and snake-substitutability are incomparable, since there are instances where one can be applied but not the other, even if on binary finite-valued instances, snake-substitutability subsumes SBT-merging. It is an ongoing research program to discover a general rule or classification of all domain reduction patterns for the VCSP.

Acknowledgements. We would like to thank Wady Naanaa for useful discussion concerning the generalisation to the general-arity case.






References

1. Alfonsín, J.L.R.: On variations of the subset sum problem. *Discrete Appl. Math.* **81**(1–3), 1–7 (1998)
2. Cohen, D.A., Cooper, M.C., Escamocher, G., Zivny, S.: Variable and value elimination in binary constraint satisfaction via forbidden patterns. *J. Comput. Syst. Sci.* **81**(7), 1127–1143 (2015)
3. Cohen, D.A., Jeavons, P., Jefferson, C., Petrie, K.E., Smith, B.M.: Symmetry definitions for constraint satisfaction problems. *Constraints* **11**(2–3), 115–137 (2006)
4. Cooper, M.C.: An optimal k-consistency algorithm. *Artif. Intell.* **41**(1), 89–95 (1989)
5. Cooper, M.C.: Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets Syst.* **134**(3), 311–342 (2003)
6. Cooper, M.C.: Beyond consistency and substitutability. In: O’Sullivan, B. (ed.) *CP 2014. LNCS*, vol. 8656, pp. 256–271. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_20
7. Cooper, M.C., Duchain, A., Mouelhi, A.E., Escamocher, G., Terrioux, C., Zanuttini, B.: Broken triangles: from value merging to a tractable class of general-arity constraint satisfaction problems. *Artif. Intell.* **234**, 196–218 (2016)
8. Cooper, M., de Givry, S., Sanchez, M., Schiex, T., Zytnicki, M., Werner, T.: Soft arc consistency revisited. *Artif. Intell.* **174**(7), 449–478 (2010)
9. Freuder, E.C.: Eliminating interchangeable values in constraint satisfaction problems. In: Dean, T.L., McKeown, K. (eds.) *Proceedings of the 9th National Conference on Artificial Intelligence*, Anaheim, CA, USA, 14–19 July 1991, vol. 1, pp. 227–233. AAAI Press/The MIT Press (1991)
10. Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York (1990)
11. Gent, I.P., Petrie, K.E., Puget, J.F.: Symmetry in constraint programming. In: Rossi, F., van Beek, P., Walsh, T. (eds.) *Handbook of Constraint Programming, Foundations of Artificial Intelligence*, vol. 2, pp. 329–376. Elsevier (2006)
12. de Givry, S., Prestwich, S.D., O’Sullivan, B.: Dead-end elimination for weighted CSP. In: Schulte, C. (ed.) *CP 2013. LNCS*, vol. 8124, pp. 263–272. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_22
13. Goldstein, R.: Efficient rotamer elimination applied to protein side-chains and related spin glasses. *Biophys. J.* **66**(5), 1335–1340 (1994)
14. Guerink, N., Van Caneghem, M.: Solving crew scheduling problems by constraint programming. In: Montanari, U., Rossi, F. (eds.) *CP 1995. LNCS*, vol. 976, pp. 481–498. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60299-2_29
15. van Hoeve, W.J., Katriel, I.: Global constraints. In: Rossi, F., van Beek, P., Walsh, T. (eds.) *Handbook of Constraint Programming, Foundations of Artificial Intelligence*, vol. 2, pp. 169–208. Elsevier (2006)
16. Jeavons, P., Cohen, D., Cooper, M.: A substitution operation for constraints. In: Borning, A. (ed.) *PPCP 1994. LNCS*, vol. 874, pp. 1–9. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58601-6_85
17. Lecoutre, C., Roussel, O., Dehane, D.E.: WCSP integration of soft neighborhood substitutability. In: Milano, M. (ed.) *CP 2012. LNCS*, pp. 406–421. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33558-7_31
18. Likitvivanavong, C., Yap, R.H.C.: Eliminating redundancy in CSPs through merging and subsumption of domain values. *SIGAPP Appl. Comput. Rev.* **13**(2), 20–29 (2013)

19. Roney-Dougal, C.M., Gent, I.P., Kelsey, T., Linton, S.: Tractable symmetry breaking using restricted search trees. In: Proceedings of the 16th European Conference on Artificial Intelligence, ECAI 2004, pp. 211–215 (2004)
20. Senkul, P., Toroslu, I.H.: An architecture for workflow scheduling under resource allocation constraints. *Inf. Syst.* **30**(5), 399–422 (2005)
21. Smith, B.M., Bistarelli, S., O’Sullivan, B.: Constraint symmetry for the soft CSP. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 872–879. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_66



Solver-Independent Large Neighbourhood Search

Jip J. Dekker^{1,2}(✉) , Maria Garcia de la Banda¹ , Andreas Schutt² ,
Peter J. Stuckey^{1,2} , and Guido Tack^{1,2} 

¹ Monash University, Melbourne, Australia
{jip.dekker,maria.garciadelabanda,maria.garciadelabanda,
peter.stuckey,guido.tack}@monash.edu

² Data61, CSIRO, Melbourne, Australia
andreas.schutt@data61.csiro.au

Abstract. The combination of large neighbourhood search (LNS) methods with complete search methods has proved to be very effective. By restricting the search to (small) areas around an existing solution, the complete method is often able to quickly improve its solutions. However, developing such a combined method can be time-consuming: While the model of a problem can be expressed in a high-level solver-independent language, the LNS search strategies typically need to be implemented in the search language of the target constraint solvers. In this paper we show how we can simplify this process by (a) extending constraint modelling languages to support solver-independent LNS search definitions, and (b) defining small solver extensions that allow solvers to implement these solver-independent LNS searches. Modellers can then implement an LNS search to be executed in any extended solver, by simply using the modelling language constructs. Experiments show that the resulting LNS searches only introduce a small overhead compared to direct implementations in the search language of the underlying solvers.

1 Introduction

Large neighbourhood search (LNS [20]) is a meta-search that has proved to be very successful for scaling complete search methods, such as Constraint Programming (CP), to large optimisation problem sizes. LNS iteratively applies a particular search method to a neighbourhood surrounding a given solution. The neighbourhoods are chosen to be as large as possible, while being small enough for the search to quickly find a higher quality solution. The LNS meta-search then selects a new neighbourhood around this new solution and repeats the process.

The combination of LNS and CP has proven to be crucial for improving solving performance in a range of hard optimisation problems. While simple, unstructured neighbourhoods often work surprisingly well, the performance of many problems can be further improved if the neighbourhoods exploit the problem structure [4]. For example, in Vehicle Routing Problems, Shaw [20] proposes

to remove *related* customer visits (such as those assigned to the same vehicle) from the tours of a given solution, and then re-insert them using a CP solver. Pacino and Van Hentenryck [12] solve Job Shop Scheduling problems by repeatedly re-scheduling all activities on a single machine, or in a particular time window. See [15] for a good overview of LNS and its applications.

All these combinations of LNS and CP have been either implemented from scratch, or within a particular CP system, in order to be able to program the interaction between the meta-search, the CP search and the constraint model. This close coupling makes it difficult to experiment with different solvers as backends for LNS, and it either precludes the use of high-level solver-independent modelling languages, or it introduces a gap between the problem model (expressed in a high-level language) and the definition of the neighbourhoods (expressed at the solver level).

The aim of this paper is to **lift LNS from the solver level to the modelling level**. Our main contribution is to show that many problem-specific neighbourhoods can be (a) expressed in a very natural way in a high-level, solver-independent constraint modelling language; and (b) compiled into efficient solver-level specifications that only require a small extension of existing solvers. The new approach has been implemented for the MiniZinc [10] modelling language and solvers Gecode [7] and Chuffed [2]. Our experiments show that the approach is expressive, efficient and effective.

2 Background

Constraint Optimisation Problems. A *Constraint Optimisation Problem* (COP) is defined as a tuple $P = (C, X, D, f)$, with X a set of variables, C a set of constraints over subsets of X , D a domain such that for each $x \in X$, $D(x)$ is the set of values x may assume, and f an objective function. A *solution* of P is an *assignment* a such that $a(x) \in D(x)$ for all $x \in X$, and a satisfies all $c \in C$. An *optimal solution* is a solution a such that $f(a)$ is minimal. Usually, we are not just interested in solving one particular COP, but rather a whole *parameterised family* of COPs. We usually call these *models*, and an individual COP with fixed parameters, an *instance* of a model.

A CP solver starts from an instance (C, X, D, f) and performs *constraint propagation* of all constraints, pruning inconsistent values from D until it reaches a fixpoint D' . If at this point propagation has emptied the domain D' for any variable, the problem is *failed*. It is a *solution* if there is exactly one value left in $D(x)$ for each $x \in X$. Otherwise, the solver splits the instance into smaller sub-instances $(C \cup c_1, X, D', f) \dots (C \cup c_k, X, D', f)$ by adding new constraints $c_1 \dots c_k$, and recursively solves each of them. When the solver finds a solution, it evaluates its quality using f and adds a constraint to the remainder of the search to only allow solutions better than the current one.

Constraint Modelling Languages. Most CP solvers take as input a flat list of variables and constraints. While the facilities of the host programming language (e.g., loops and overloading) can be used to make modelling more comfortable,

Algorithm 1. Large Neighbourhood Search

```

1: procedure LNS( $P = (C, X, D, f)$ )
2:    $a \leftarrow \text{findsolution}(P)$ 
3:   while  $\neg \text{timeout}()$  do
4:      $P' = (C \cup \text{nbh}(a) \cup \text{obj}(a), X, D, f)$ 
5:      $a' \leftarrow \text{findsolution}(P')$ 
6:     if  $a'$  is a solution then  $a \leftarrow a'$ 
7:   return  $a$ 

```

dedicated Constraint Modelling Languages have become popular (e.g., OPL [24], AMPL [5], Essence [6], and MiniZinc [10]), as they support problem specification at a higher level of abstraction, and in a solver-independent way. We use MiniZinc due to its widespread use and support for over 20 different solvers. MiniZinc supports high-level features, such as different variable types, complex Boolean and arithmetic expressions, user-defined functions and predicates, and a comprehensive library of predefined global constraints. A (solver-independent) MiniZinc model is translated by the MiniZinc compiler into (solver-specific) FlatZinc, which is then interpreted by the target solver. The compiler uses a library of predicate and function definitions, written in MiniZinc specifically for the target solver, to generate FlatZinc code that only contains constraints and variable types supported by the target solver. This cleanly de-couples the MiniZinc compiler from the solver.

Large Neighbourhood Search. LNS is usually described as a meta-search that starts from a solution to a COP, *relaxes* part of the solution, and then *re-optimises* that part. This process is iterated using a meta-heuristic (e.g., hill-climbing or simulated annealing) to improve the solution, until some stopping criterion is met, such as a time limit or a solution quality. The relaxation step is also described as *freeing up*, *destroying* or *thawing* certain variables in the solution to their original domain. Formally, consider a problem $P = (C, X, D, f)$, a solution a of P , and a domain D' built by selecting a subset of the variables $Y \subset X$ such that $D'(x)$ is set to $a(x)$ for $x \in Y$, and to $D(x)$ otherwise. A *neighbourhood* of a is defined as (C, X, D', f) . Depending on the size of Y , this problem is significantly smaller than P , and can thus be solved efficiently using e.g. a CP solver. A more flexible and expressive definition of neighbourhood is $(C \cup \text{nbh}(a), X, D, f)$, where given solution a , $\text{nbh}(a)$ returns a set of constraints to be added to P for the next iteration of the LNS. This definition subsumes the one above, and allows neighbourhoods that instead of fixing variables, constrain them to take values close to the last solution.

Algorithm 1 shows a simple version of LNS, where $\text{findsolution}(P)$ invokes a complete solver to compute a new solution of problem P , and function $\text{obj}(a)$ adds constraints to ensure that only solutions better than a (according to f) are returned. Thus, the algorithm implements a simple hill-climbing that is terminated after a timeout. Real implementations would also terminate the complete solver in line 5 after a timeout (in order to balance intensification

and exploration). Note that if P' does not lead to an improved solution, the algorithm computes a new neighbourhood $nbh(a)$ for the previous solution. Therefore, function nbh is typically non-deterministic (via random number generators) and impure (in programming language terms). More sophisticated versions of this algorithm implement other meta-heuristics, such as simulated annealing, or automatically switch between neighbourhood definitions (as in Adaptive LNS [19]).

3 Modelling of Neighbourhoods and Meta-Heuristics

This section introduces a MiniZinc extension that enables modellers to define neighbourhoods using the $nbh(a)$ approach described above. This extension is based on the constructs introduced in MiniSearch [18], as summarised below.

```

1 predicate uniformNeighbourhood(array[int] of var int: x, float: destrRate) =
2   forall(i in index_set(x))
3     (if uniform(0.0,1.0) > destrRate then x[i] = sol(x[i]) else true endif);

```

Listing 1: A simple random LNS predicate implemented in MiniSearch

```

1 function ann: lns(var int: obj, array[int] of var int: vars,
2                 int: iterations, float: destrRate, int: exploreTime) =
3   repeat (i in 1..iterations) ( scope(
4     if has_sol() then post(uniformNeighbourhood(vars,destrRate))
5     else true endif /\
6     time_limit(exploreTime, minimize_bab(obj)) /\
7     commit() /\ print()
8   ) /\ post(obj < sol(obj)) );

```

Listing 2: A simple LNS metaheuristic implemented in MiniSearch

3.1 LNS in MiniSearch

MiniSearch introduced a MiniZinc extension that enables modellers to express meta-searches inside a MiniZinc model. A meta-search in MiniSearch typically solves a given MiniZinc model, performs some calculations on the solution, adds new constraints and then solves again. An LNS definition in MiniSearch consists of two parts. The first part is a declarative definition of a neighbourhood as a MiniZinc predicate that posts the constraints that should be added with respect to a previous solution. This makes use of the MiniSearch function: `function int: sol(var int: x)`, which returns the value that variable x was assigned to in the previous solution (similar functions are defined for Boolean, float and set variables). In addition, a neighbourhood predicate will typically make use of the random number generators available in the MiniZinc standard library. Listing 1 shows a simple random neighbourhood. For each decision variable $x[i]$, it draws a random number from a uniform distribution and, if it exceeds threshold `destrRate`, posts constraints forcing $x[i]$ to take the same value as in the previous solution. For example, `uniformNeighbourhood(x,0.2)` would result in each variable in the array x having a 20% chance of being

unconstrained, and an 80% chance of being assigned to the value it had in the previous solution.

The second part of a MiniSearch LNS is the meta-search itself. The most basic example is that of function `lns` in Listing 2. It performs a fixed number of iterations, each invoking the neighbourhood predicate `uniformNeighbourhood` in a fresh scope (so that the constraints only affect the current loop iteration). It then searches for a solution (`minimize_bab`) with a given timeout, and if the search does return a new solution, it commits to that solution (so that it becomes available to the `sol` function in subsequent iterations). The `lns` function also posts the constraint `obj < sol(obj)`, ensuring the objective value in the next iteration is strictly better than that of the current solution.

Limitations of the MiniSearch Approach. Although MiniSearch enables the modeller to express *neighbourhoods* in a declarative way, the definition of the *meta-search* is rather unintuitive and difficult to debug, leading to unwieldy code for defining simple restarting strategies. Furthermore, the MiniSearch implementation requires either a close integration of the backend solver into the MiniSearch system, or it drives the solver through the regular text-file based FlatZinc interface, leading to a significant communication overhead. To address these two issues for LNS, we propose to keep modelling neighbourhoods as predicates, but define a small number of additional MiniZinc built-in annotations and functions that (a) allow us to express important aspects of the meta-search in a more convenient way, and (b) enable a simple compilation scheme that requires no additional communication with and only small, simple extensions of the backend solver.

3.2 Restart Annotations

Instead of the complex MiniSearch definitions, we propose to add support for simple meta-searches that are purely based on the notion of *restarts*. A restart happens when a solver abandons its current search efforts, returns to the root node of the search tree, and begins a new exploration. Many CP solvers already provide support for controlling their restarting behaviour, e.g. they can periodically restart after a certain number of nodes, or restart for every solution. Typically, solvers also support posting additional constraints upon restarting (e.g. Comet [9]) that are only valid for the particular restart (i.e., they are “retracted” for the next restart). In its simplest form, we can therefore implement LNS by specifying a neighbourhood predicate, and annotating the `solve` item to indicate the predicate should be invoked upon each restart:

```
solve ::on_restart(myNeighbourhood) minimize cost;
```

Note that MiniZinc currently does not support passing functions or predicates as arguments. Calling the predicate, as in `::on_restart(myNeighbourhood())`, would not have the correct semantics, since the predicate needs to be called for *each* restart. As a workaround, we currently pass the name of the predicate to be called for each restart as a string (see the definition of the new `on_restart` annotation in Listing 3).

The second component of our LNS definition is the *restarting strategy*, defining how much effort the solver should put into each neighbourhood (i.e., restart), and when to stop the overall search. We propose adding new search annotations to MiniZinc to control this behaviour (see Listing 3). The `restart_on_solution` annotation tells the solver to restart immediately for each solution, rather than looking for the best one in each restart, while `restart_without_objective` tells it not to add branch-and-bound constraints on the objective. The other `restart_X` annotations define different strategies for restarting the search when no solution is found. The `timeout` annotation gives an overall time limit for the search, whereas `restart_limit` stops the search after a fixed number of restarts.

3.3 Neighbourhood Selection

It is often beneficial to use several neighbourhood definitions for a problem. Different neighbourhoods may be able to improve different aspects of a solution, at different phases of the search. Adaptive LNS [14, 19], which keeps track of the neighbourhoods that led to improvements and favours them for future iterations, is the prime example for this approach. A simpler scheme may apply several neighbourhoods in a round-robin fashion.

In MiniSearch, adaptive or round-robin approaches can be implemented using *state variables*, which support destructive update (overwriting the value they store). In this way, the MiniSearch strategy can store values to be used in later iterations. We use the *solver state* instead, i.e., normal decision variables, and define two simple built-in functions to access the solver state of the *previous restart*. This approach is sufficient for expressing neighbourhood selection strategies, and its implementation is much simpler.

```

1 % post predicate "pred" whenever the solver restarts
2 annotation on_restart(string: pred);
3 % restart after fixed number of nodes
4 annotation restart_constant(int: nodes);
5 % restart with scaled Luby sequence
6 annotation restart_luby(int: scale);
7 % restart with scaled geometric sequence (scale*base^n in the n-th iteration)
8 annotation restart_geometric(float: base, int: scale);
9 % restart with linear sequence (scale*n in the n-th iteration)
10 annotation restart_linear(int: scale);
11 % restart on each solution
12 annotation restart_on_solution;
13 % restart without branch-and-bound constraints on the objective
14 annotation restart_without_objective;
15 % overall time limit for search
16 annotation timeout(int: seconds);
17 % overall limit on number of restarts
18 annotation restart_limit(int: n_restarts);

```

Listing 3: New annotations to control the restarting behaviour

```

1 % Report the status of the solver (before restarting).
2 enum STATUS = {START, UNKNOWN, UNSAT, SAT, OPT}
3 function STATUS: status();
4 % Provide access to the last assigned value of variable x.
5 function int: lastval(var int: x);
    
```

Listing 4: Functions for accessing previous solver states

State Access and Initialisation. The state access functions are defined in Listing 4. Function `status` returns the status of the previous restart, namely: `START` (there has been no restart yet); `UNSAT` (the restart failed); `SAT` (the restart found a solution); `OPT` (the restart found and proved an optimal solution); and `UNKNOWN` (the restart did not fail or find a solution). Function `lastval` (which, like `sol`, has versions for all basic variable types) allows modellers to access the last value assigned to a variable (the value is undefined if `status()`=`START`). In order to be able to initialise the variables used for state access, we reinterpret `on_restart` so that the predicate is also called for the initial search (i.e., before the first “real” restart) with the same semantics, that is, any constraint posted by the predicate will be retracted for the next restart.

Parametric Neighbourhood Selection Predicates. We define standard neighbourhood selection strategies as predicates that are parametric over the neighbourhoods they should apply. For example, since `on_restart` now also includes the initial search, we can define a strategy `basic_lns` that applies a neighbourhood only if the current status is not `START`:

```

1 array[1..n] of var 1..n: x; % decision variables
2 var int: cost; % objective function
3 % ... some constraints defining the problem
4 % The user-defined LNS strategy
5 predicate my_lns() = basic_lns(uniformNeighbourhood(x,0.2));
6 % Solve using my_lns, restart every 500 nodes, overall timeout 120 seconds
7 solve ::on_restart("my_lns") ::restart_constant(500) ::timeout(120)
8 minimize cost;
    
```

Listing 5: Complete LNS example

```

1 predicate round_robin(array[int] of var bool: nbhs) =
2     let { int: N = length(nbhs);
3         var -1..N-1: select; % Neighbourhood selection
4     } in if status()==START then select= -1
5         else select= (lastval(select) + 1) mod N
6         endif /\
7         forall(i in 1..N) (select=i-1 -> nbhs[i]);
    
```

Listing 6: A predicate providing the round robin meta-heuristic

```

predicate basic_lns(var bool: nbh) = (status()!=START -> nbh);
    
```

In order to use this predicate with the `on_restart` annotation, we cannot simply pass `basic_lns(uniformNeighbourhood(x,0.2))`. First of all, calling

`uniformNeighbourhood` like that would result in a *single* evaluation of the predicate, since MiniZinc employs a call-by-value evaluation strategy. Furthermore, the `on_restart` annotation only accepts the name of a nullary predicate. Therefore, users have to define their overall strategy in a new predicate. Listing 5 shows a complete example of a basic LNS model.

We can also define round-robin and adaptive strategies using these primitives. Listing 6 defines a round-robin LNS meta-heuristic, which cycles through a list of `N` neighbourhoods `nbhs`. To do this, it uses the decision variable `select`. In the initialisation phase (`status()=START`), `select` is set to `-1`, which means none of the neighbourhoods is activated. In any following restart, `select` is incremented modulo `N`, by accessing the last value assigned in a previous restart (`lastval(select)`). This will activate a different neighbourhood for each restart (line 7). For adaptive LNS, a simple strategy is to change the size of the neighbourhood depending on whether the previous size was successful or not. Listing 7 shows an adaptive version of the `uniformNeighbourhood` that increases the number of free variables when the previous restart failed, and decreases it when it succeeded, within the bounds `[0.6, 0.95]`.

3.4 Meta-Heuristics

The LNS strategies we have seen so far rely on the default behaviour of MiniZinc solvers to use branch-and-bound for optimisation: when a new solution is found, the solver adds a constraint to the remainder of the search to only accept better solutions, as defined by the objective function in the `minimize` or `maximize` clause of the `solve` item. When combined with restarts and LNS, this is equivalent to a simple hill-climbing meta-heuristic.

```

1 predicate adaptiveUniform(array[int] of var int: x, float: initialDestrRate) =
2   let { var float: rate; } in
3   if      status() = START then rate = initialDestrRate
4   elseif status() = UNSAT then rate = min(lastval(rate)-0.02,0.6)
5   else                                     rate = max(lastval(rate)+0.02,0.95)
6   endif /\
7   forall(i in index_set(x))
8     (if uniform(0.0,1.0) > rate then x[i] = sol(x[i]) else true endif);

```

Listing 7: A simple adaptive neighbourhood

We can use the constructs introduced above to implement alternative meta-heuristics such as simulated annealing. In particular, we use `restart_without_objective` to tell the solver not to add the branch-and-bound constraint on restart. It will still use the declared objective to decide whether a new solution is the globally best one seen so far, and only output those (to maintain the convention of MiniZinc solvers that the last solution printed at any point in time is the currently best known one). With `restart_without_objective`, the restart predicate is now responsible for constraining the objective function. Note

that a simple hill-climbing (for minimisation) can still be defined easily in this context as:

```

1 predicate hill_climbing() =
2   if status()=START then true
3   else _objective < sol(_objective) endif;

```

It takes advantage of the fact that the declared objective function is available through the built-in variable `_objective`. A simulated annealing strategy is also easy to express:

```

1 predicate simulated_annealing(float: initTemp, float: coolingRate) =
2   let { var float: temp; } in
3   if status()=START then temp = initTemp
4   else
5     temp = lastval(temp)*(1-coolingRate) /\ % cool down
6     _objective < sol(_objective) - ceil(log(uniform(0.0,1.0))) * temp
7   endif;

```

4 Compilation of Neighbourhoods

The neighbourhoods defined in the previous section can be executed with MiniSearch by adding support for the `status` and `lastval` built-in functions, and by defining the main restart loop. The MiniSearch evaluator will then call a solver to produce a solution, and evaluate the neighbourhood predicate, incrementally producing new FlatZinc to be added to the next round of solving. While this is a viable approach, our goal is to keep the compiler and solver separate, by embedding the entire LNS specification into the FlatZinc that is passed to the solver. This section introduces such a compilation approach. It only requires simple modifications of the MiniZinc compiler, and the compiled FlatZinc can be executed by standard CP solvers with a small set of simple extensions.

4.1 Compilation Overview

The neighbourhood definitions from the previous section have an important property that makes them easy to compile to standard FlatZinc: they are defined in terms of standard MiniZinc expressions, with the exception of a few new built-in functions. When the neighbourhood predicates are evaluated in the MiniSearch way, the MiniSearch runtime implements those built-in functions, computing the correct value whenever a predicate is evaluated. Instead, the compilation scheme presented below uses a limited form of *partial evaluation*: parameters known at compile time will be fully evaluated; those only known during the solving, such as the result of a call to any of the new functions (`sol`, `status` etc.), are replaced by decision variables. This essentially **turns the new built-in functions into constraints** that have to be supported by the target solver. The neighbourhood predicate can then be added as a constraint to the

model. The evaluation is performed by hijacking the solver’s own capabilities: It will automatically perform the evaluation of the new functions by propagating the new constraints.

To compile an LNS specification to standard FlatZinc, the MiniZinc compiler performs four simple steps:

1. Replace the annotation `::on_restart("X")` with a call to predicate `X`.
2. Inside predicate `X` and any other predicate called recursively from `X`: treat any call to built-in functions `sol`, `status`, and `lastval` as returning a `var` instead of a `par` value; and rename calls to random functions, e.g., `uniform` to `uniform_nbh`, in order to distinguish them from their standard library versions.
3. Convert any expression containing a call from step 2 to `var` to ensure the functions are compiled as constraints, rather than statically evaluated by the MiniZinc compiler.
4. Compile the resulting model using an extension of the MiniZinc standard library that provides declarations for these built-in functions, as defined below.

These transformations will not change the code of many neighbourhood definitions, since the built-in functions are often used in positions that accept both parameters and variables. For example, the `uniformNeighbourhood` predicate from Listing 1 uses `uniform(0.0,1.0)` in an `if` expression, and `sol(x[i])` in an equality constraint. Both expressions can be translated to FlatZinc when the functions return a `var`.

4.2 Compiling the New Built-Ins

We can compile models that contain the new built-ins by extending the MiniZinc standard library as follows.

status. Listing 8 shows the definition of the `status` function. It simply replaces the functional form by a predicate `status` (declared in line 1), which constrains its local variable argument `stat` to take the status value.

sol and lastval. Since `sol` is overloaded for different variable types and FlatZinc does not support overloading, we produce type-specific built-ins for every type of solver variable (`int_sol(x, xi)`, `bool_sol(x, xi)`, etc.). The resolving of the `sol` function into these specific built-ins is done using an overloaded definition like the one shown in Listing 9 for integer variables. If the value of the variable in question becomes known at compile time, we use that value instead. Otherwise, we replace the function call with a type specific `int_sol` predicate, which is the constraint that will be executed by the solver. To improve the compilation of the model further, we use the declared bounds of the argument (`lb(x)..ub(x)`) to constrain the variable returned by `sol`. This bounds information is important for the compiler to be able to generate the most efficient FlatZinc code for expressions involving `sol`. The compilation of `lastval` is similar to that for `sol`.

```

1 predicate status(var int: stat);
2 function var STATUS: status() =
3     let { var STATUS: stat;
4         constraint status(stat);
5     } in stat;

```

Listing 8: MiniZinc definition of the status function

```

1 predicate int_sol(var int: x, var int: xi);
2 function int: sol(var int: x) = if is_fixed(x) then fix(x)
3     else let { var lb(x)..ub(x): xi;
4         constraint int_sol(x,xi);
5     } in xi;
6 endif;

```

Listing 9: MiniZinc definition of the sol function for integer variables

Random Number Functions. Calls to the random number functions have been renamed by appending `_nbh`, so that the compiler does not simply evaluate them statically. The definition of these new functions follows the same pattern as for `sol`, `status`, and `lastval`. The MiniZinc definition of the `uniform_nbh` function is shown in Listing 10.¹ Note that the function accepts variable arguments `l` and `u`, so that it can be used in combination with other functions, such as `sol`.

```

1 predicate float_uniform(var float:l, var float: u, var float: r);
2 function var float: uniform_nbh(var float: l, var float: u) :: impure =
3     let { var lb(l)..ub(u): rnd;
4         constraint float_uniform(l,u,rnd);
5     } in rnd;

```

Listing 10: MiniZinc definition of the `uniform_nbh` function for floats

4.3 Solver Support for LNS FlatZinc

We will now show the minimal extensions required from a solver to interpret the new FlatZinc constraints and, consequently, to execute LNS definitions expressed in MiniZinc. First, the solver needs to parse and support the restart annotations of Listing 3. Many solvers already support all this functionality. Second, the solver needs to be able to parse the new constraints `status`, and all versions of `sol`, `lastval`, and random number functions like `float_uniform`. In addition, for the new constraints the solver needs to:

- `status(s)`: record the status of the previous restart, and fix `s` to the recorded status.
- `sol(x, sx)` (variants): constrain `sx` to be equal to the value of `x` in the incumbent solution. If there is no incumbent solution, it has no effect.

¹ Random number functions need to be marked as `::impure` for the compiler not to apply Common Subexpression Elimination (CSE) [23] if they are called multiple times with the same arguments.

- `lastval(x, lx)` (variants): constrain `lx` to take the last value assigned to `x` during search. If no value was ever assigned, it has no effect. Note that many solvers (in particular SAT and LCG solvers) already track `lastval` for their variables for use in search. To support LNS a solver must at least track the *lastval* of each of the variables involved in such a constraint. This is straightforward by using the `lastval` propagator itself. It wakes up whenever the first argument is fixed, and updates the last value (a non-backtrackable value).
- random number functions: fix their variable argument to a random number in the appropriate probability distribution.

Importantly, these constraints need to be propagated in a way that their effects can be undone for the next restart. Typically, this means the solver must not propagate these constraints in the root node of the search.

Modifying a solver to support this functionality is straightforward if it already has a mechanism for posting constraints during restarts. We have implemented these extensions for both Gecode (110 new lines of code) and Chuffed (126 new lines of code).

Example 1. Consider the model from Listing 5 again. Listing 11 shows a part of the FlatZinc that arises from compiling `basic_lns(uniformNeighbourhood(x, 0.2))`, assuming that `index.set(x) = 1..n`. Lines 1–4 define a Boolean variable `b1` that is true iff the status is not `START`. The second block of code (lines 6–15) represents the decomposition of the expression `(status() != START /\ uniform(0.0, 1.0) > 0.2) -> x[1] = sol(x[1])`, which is the result of merging the implication from the `basic_lns` predicate with the `if` expression from `uniformNeighbourhood`. The code first introduces and constrains a variable for the random number, then adds two Boolean variables: `b2` is constrained to be true iff the random number is greater than 0.2; while `b3` is constrained to be the conjunction `status() != START /\ uniform(0.0, 1.0) > 0.2`. Line 13 constrains `x1` to be the value of `x[1]` in the previous solution. Finally, the half-reified constraint in line 15 implements `b3 -> x[1] = sol(x[1])`. We have omitted the similar code generated for `x[2]` to `x[n]`. Note that the FlatZinc shown here has been simplified for presentation.

The first time the solver is invoked, it sets `s` to 1 (`START`). Propagation will fix `b1` to `false` and `b3` to `false`. Therefore, the implication in line 15 is not activated, leaving `x[1]` unconstrained. The neighbourhood constraints are effectively switched off.

When the solver restarts, all of the special propagators are re-executed. Now `s` is not 1, and `b1` will be set to `true`. The `float_random` propagator assigns `rnd1` a new random value and, depending on whether it is greater than 0.2, the Boolean variables `b2`, and consequently `b3` will be assigned. If it is `true`, the constraint in line 15 will become active and assign `x[1]` to its value in the previous solution. □

```

1 var 1..5: s;
2 constraint status(s);
3 var bool b1;
4 constraint int_ne_reif(s,1,b1); % b1 <-> status() != START
5
6 var 0.0..1.0: rnd1;
7 constraint float_uniform(0.0,1.0,rnd1);
8 var bool: b2;
9 constraint float_gt_reif(rnd1,0.2,b2);
10 var bool: b3;
11 constraint bool_and(b1,b2,b3);
12 var 1..3: x1;
13 constraint int_sol(x[1],x1);
14 % (status() != START ^ uniform(0.0,1.0) > 0.2) -> x[1] = sol(x[1])
15 constraint int_eq_imp(x[1],x1,b3);
16 ...
    
```

Listing 11: FlatZinc that results from compiling `basic_lns(UniformNeighbourhood(x,0.2))`.

5 Experiments

We will now show that a solver that evaluates the compiled FlatZinc LNS specifications can (a) be effective and (b) incur only a small overhead compared to a dedicated implementation of the neighbourhoods.

To measure the overhead, we implemented our new approach in Gecode [7]. The resulting solver (`gecode-fzn` in the tables below) has been instrumented to also output the domains of all model variables after propagating the new special constraints. We implemented another extension to Gecode (`gecode-replay`) that simply reads the stream of variable domains for each restart, essentially replaying the LNS of `gecode-fzn` without incurring any overhead for evaluating the neighbourhoods or handling the additional variables and constraints. Note that this is a conservative estimate of the overhead: `gecode-replay` has to perform *less* work than any real LNS implementation.

In addition, we also present benchmark results for the standard release of Gecode 6.0 without LNS (`gecode`); as well as `chuffed`, the development version of Chuffed; and `chuffed-fzn`, Chuffed performing LNS with FlatZinc neighbourhoods. These experiments illustrate that the LNS implementations indeed perform well compared to the standard solvers.² All experiments were run on a single core of an Intel Core i5 CPU @ 3.4 GHz with 4 cores and 16 GB RAM running MacOS High Sierra. LNS benchmarks are repeated with 10 different random seeds and the average is shown. The overall timeout for each run is 120 s.

We ran experiments for three models from the MiniZinc challenge [21, 22] (`gbac`, `steelmillslab`, and `rcpsp-wet`). The best objective found during the MiniZinc Challenge is shown for every instance (*best known*). For each solving method we measured the average integral of the model objective after finding the initial solution (f), the average best objective found (min), and the standard

² Our implementations are available at <https://github.com/Dekker1/{libminizinc,gecode,chuffed}> on branches containing the keyword `on_restart`.

Table 1. gbac benchmarks

	best known	gecode		gecode-fzn		gecode-replay		chuffed		chuffed-fzn	
Instance	min	f	min	f	min	f	min	f	min	f	min
UD2-gbac	146	1502k	12515	93k	376 ¹⁶	92k	362 ¹⁵	1494k	12344	207k	598 ⁵⁴
UD4-gbac	396	1517k	12645	121k	932 ²⁴	120k	932 ²⁴	1151k	9267	160k	1142 ⁵
UD5-gbac	222	2765k	23028	283k	2007 ³⁹	281k	2007 ³⁹	2569k	21233	483k	2572 ²²
UD8-gbac	40	1195k	9611	21k	53 ²⁶	20k	53 ²⁶	1173k	9559	114k	76 ²⁶
reduced_UD4	949	629k	4917	114k	950 ⁰	114k	950 ⁰	715k	5491	117k	950 ⁰

deviation of the best objective found in percentage (%), which is shown as the superscript on min when running LNS. The underlying search strategy used is the fixed search strategy defined in the model. For each model we use a round robin evaluation (Listing 6) of two neighbourhoods: a neighbourhood that destroys 20% of the main decision variables (Listing 1) and a structured neighbourhood for the model (described below). The restart strategy is `::restart_constant(250) ::restart_on_solution`.

gbac. The Generalised Balanced Academic Curriculum problem comprises courses having a specified number of credits and lasting a certain number of periods, load limits of courses for each period, prerequisites for courses, and preferences of teaching periods for professors. A detailed description of the problem is given in [1]. The main decisions are to assign courses to periods, which is done via the variables `period_of` in the model. Listing 12 shows the neighbourhood chosen, which randomly picks one period and frees all courses that are assigned to it.

The results for **gbac** in Table 1 show that the overhead introduced by `gecode-fzn` w.r.t. `gecode-replay` is quite low, and both their results are much better than the baseline `gecode`. Since learning is not very effective for **gbac**, the performance of `chuffed` is inferior to `Gecode`. However, LNS again significantly improves over standard `Chuffed`.

steelmillslab. The Steel Mill Slab design problem consists of cutting slabs into smaller ones, so that all orders are fulfilled while minimising the wastage. The steel mill only produces slabs of certain sizes, and orders have both a size and a colour. We have to assign orders to slabs, with at most two different colours on each slab. The model uses the variables `assign` for deciding which order is assigned to which slab. Listing 13 shows a structured neighbourhood that randomly selects a slab and frees the orders assigned to it in the incumbent solution. These orders can then be freely reassigned to any other slab.

```

1 let { int: period = uniform(periods) } in
2 forall(i in courses where sol(period_of[i]) != period)
3   (period_of[i] = sol(period_of[i]));

```

Listing 12: gbac: neighbourhood freeing all courses in a period.

```

1 predicate free_slab() =
2   let { int: slab = uniform(1, nbSlabs) } in
3   forall(i in 1..nbSlabs where slab != sol(assign[i]))
4     (assign[i] = sol(assign[i]));

```

Listing 13: steelmillslab: Neighbourhood that frees all orders assigned to a selected slab.

Table 2. steelmillslab benchmarks

Instance	best known	gecode		gecode-fzn		gecode-replay		chuffed		chuffed-fzn	
	min	f	min	f	min	f	min	f	min	f	min
bench_13_0	0	3247	27	20	0⁰	19	0⁰	1315	9	50	0⁰
bench_14_1	0	1248	0	32	0⁰	31	0⁰	72	0	79	0⁰
bench_15_11	0	4458	30	27	0⁰	26	0⁰	143	0	65	0⁰
bench_16_10	0	2446	0	19	0⁰	19	0⁰	122	0	51	0⁰
bench_19_5	0	3380	28	12	0⁰	11	0⁰	3040	19	31	0⁰

For this problem a solution with zero wastage is always optimal. The use of LNS makes these instances easy, as all the LNS approaches find optimal solutions. As Table 2 shows, `gecode-fzn` is again slightly slower than `gecode-replay` (the integral is slightly larger). While `chuffed` significantly outperforms `gecode` on this problem, once we use LNS, the learning in `chuffed-fzn` is not advantageous compared to `gecode-fzn` or `gecode-replay`. Still, `chuffed-fzn` outperforms `chuffed` by always finding an optimal solution.

rcsp-wet. The Resource-Constrained Project Scheduling problem with Weighted Earliness and Tardiness cost, is a classic scheduling problem in which tasks need to be scheduled subject to precedence constraints and cumulative resource restrictions. The objective is to find an optimal schedule that minimises the weighted cost of the earliness and tardiness for tasks that are not completed by their proposed deadline. The decision variables in array `s` represent the start times of each task in the model. Listing 14 shows our structured neighbourhood for this model. It randomly selects a time interval of one-tenth the length of the planning horizon and frees all tasks starting in that time interval, which allows a reshuffling of these tasks.

Table 3 shows that `gecode-replay` and `gecode-fzn` perform almost identically, and substantially better than baseline `gecode` for these instances. The baseline learning solver `chuffed` is best overall on the easy examples, but LNS makes it much more robust. The poor performance of `chuffed-fzn` on the last instance is due to the fixed search, which limits the usefulness of nogood learning.

Table 3. rcpsp-wet benchmarks

Instance	best known		gecode		gecode-fzn		gecode-replay		chuffed		chuffed-fzn	
	min	f	min	f	min	f	min	f	min	f	min	f
j30_1.3-wet	93	20k	161	11k	93 ⁰	11k	93 ⁰	3k	93	13k	93 ⁰	
j30_43.10-wet	121	19k	158	15k	121 ⁰	14k	121 ⁰	10k	121	15k	121 ⁰	
j60_19.6-wet	227	54k	441	29k	235 ³	29k	235 ³	63k	487	29k	227 ⁰	
j60_28.3-wet	266	94k	770	33k	273 ⁰	33k	273 ⁰	79k	604	35k	272 ¹	
j90_48.4-wet	513	199k	1653	72k	535 ²	71k	535 ²	201k	1638	109k	587 ²	

```

1 predicate free_timeslot() =
2   let { int: slot = max(Times) div 10;
3         int: time = uniform(min(Times), max(Times) - slot); } in
4   forall(t in Tasks)
5     ((sol(s[t]) < time ∨ time+slot > sol(s[t])) -> s[t] = sol(s[t]));

```

Listing 14: rcpsp-wet: Neighbourhood freeing all tasks starting in the drawn interval.

Summary. The results show that LNS outperforms the baseline solvers, except for benchmarks where we can quickly find and prove optimality. However, the main result from these experiments is that the overhead introduced by our Flat-Zinc interface, when compared to an optimal LNS implementation, is relatively small. We have additionally calculated the rate of search nodes explored per second and, across all experiments, `gecode-fzn` achieves around 3% fewer nodes per second than `gecode-replay`. This overhead is caused by propagating the additional constraints in `gecode-fzn`. Overall, the experiments demonstrate that the compilation approach is an effective and efficient way of adding LNS to a modelling language with minimal changes to the solver.

6 Related Work and Conclusion

Large neighbourhood search is straightforward to implement using a scripting language and a separate modelling language. Scripting languages like MiniSearch [18], OPL Script [25] and AMPL Script can be used transparently with their underlying modelling language. However, this form of LNS requires either the solver to be restarted from scratch for every solve, or the scripting language to be tightly tied to a particular solver.

Many CP systems such as Choco [16], Comet [9], Objective CP [26], OSCAR [11], or-tools [8] have an `onRestart` or `onSolution` event (or similar APIs) to which arbitrary code can be attached. This makes LNS easy to implement, although it typically mixes declarative and procedural aspects. It is also much more expressive than MiniSearch but relies on a tight relationship with the underlying solvers. Support for our compiled LNS specifications will be easy to implement for these solvers.

There are a number of approaches to automatically defining neighbourhoods, such as random [3], propagation guided [13], and explanation based [17] neighbourhoods. If the solver supports these then they can be used to build LNS solutions using this strategy straightforwardly. They are orthogonal to user defined neighbourhoods.

In this paper we have shown how we can take a high level model and LNS definition and communicate that to a CP solver, which then completes the LNS search. The additions to the CP solver are minor, by hijacking the use of propagators to do expression evaluation, and adding a few simple (pseudo-) propagators. The result is a solver independent approach to LNS that does not rely on repeated calls to the solver. While we have concentrated on LNS, it seems that more of MiniSearch [18] can be compiled into FlatZinc in the same way. This opens up interesting possibilities for further research.

Acknowledgements. This research was partly sponsored by the Australian Research Council grant DP180100151.

References

1. Chiarandini, M., Gaspero, L.D., Gualandi, S., Schaerf, A.: The balanced academic curriculum problem revisited. *J. Heuristics* **18**(1), 119–148 (2012)
2. Chu, G.: Improving Combinatorial Optimization. Department of Computing and Information Systems, University of Melbourne (2011)
3. Cipriano, R., Di Gaspero, L., Dovier, A.: Gelato: a multi-paradigm tool for Large Neighborhood Search. In: Talbi, E.-G. (ed.) *Hybrid Metaheuristics*, pp. 389–414. Springer, Heidelberg (2013)
4. Danna, E., Perron, L.: Structured vs. Unstructured Large Neighborhood Search: A Case Study on Job-Shop Scheduling Problems with Earliness and Tardiness Costs. In: Rossi, F. (ed.) *CP 2003. LNCS*, vol. 2833, pp. 817–821. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45193-8_59
5. Fourer, R., Gay, D., Kernighan, B.: AMPL: A Mathematical Programming Language. *Manage. Sci.* **36**, 519–554 (1990)
6. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: a constraint language for specifying combinatorial problems. *Constraints* **13**(3), 268–306 (2008)
7. Gecode Team: Gecode: A Generic Constraint Development Environment (2016). <http://www.gecode.org>
8. Google: or-tools (2017). <https://developers.google.com/optimization/>
9. Michel, L., Van Hentenryck, P.: The Comet Programming Language and System. In: van Beek, P. (ed.) *CP 2005. LNCS*, vol. 3709, pp. 881–881. Springer, Heidelberg (2005). https://doi.org/10.1007/11564751_119
10. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) *CP 2007. LNCS*, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_38
11. Oscar Team: Oscar: Scala in OR (2012). <https://bitbucket.org/oscarlib/oscar>

12. Pacino, D., Van Hentenryck, P.: Large neighborhood search and adaptive randomized decompositions for flexible jobshop scheduling. In: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Three. IJCAI 11, pp. 1997–2002. AAAI Press, Barcelona (2011)
13. Perron, L., Shaw, P., Furnon, V.: Propagation guided large neighborhood search. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 468–481. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30201-8_35
14. Pisinger, D., and Ropke, S.: A general heuristic for vehicle routing problems. *Comput. Oper. Res.* **34**(8), 2403–2435 (2007)
15. Pisinger, D., and Ropke, S.: Large neighborhood search. In: Gendreau, M., Potvin, J.-Y. (eds.) *Handbook of Metaheuristics*, pp. 399–419. Springer, Boston (2010). ISBN: 978-1-4419-1665-5. https://doi.org/10.1007/978-1-4419-1665-5_13
16. Prud’homme, C., Fages, J.-G., Lorca, X.: Choco documentation. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. (2017). <http://www.choco-solver.org>
17. Prud’homme, C., Lorca, X., Jussien, N.: Explanation-Based Large Neighborhood Search. *Constraints* **19**(4), 339–379 (2014)
18. Rendl, A., Guns, T., Stuckey, P.J., Tack, G.: MiniSearch: a solver-independent meta-search language for MiniZinc. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 376–392. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_27
19. Ropke, S., Pisinger, D.: An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transp. Sci.* **40**(4), 455–472 (2006)
20. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M., Puget, J.-F. (eds.) CP 1998. LNCS, vol. 1520, pp. 417–431. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49481-2_30
21. Stuckey, P.J., Becket, R., Fischer, J.: Philosophy of the MiniZinc challenge. *Constraints* **15**(3), 307–316 (2010)
22. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The MiniZinc challenge 2008–2013. *AI Mag.* **35**(2), 55–60 (2014)
23. Stuckey, P.J., Tack, G.: MiniZinc with functions. In: Gomes, C., Sellmann, M. (eds.) CPAIOR 2013. LNCS, vol. 7874, pp. 268–283. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38171-3_18
24. Van Hentenryck, P.: *The OPL Optimization Programming Language*. MIT Press, Cambridge (1999)
25. Van Hentenryck, P., Michel, L.: OPL script: composing and controlling models. In: Apt, K.R., Monfroy, E., Kakas, A.C., Rossi, F. (eds.) WC 1999. LNCS (LNAI), vol. 1865, pp. 75–90. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44654-0_4
26. Van Hentenryck, P., Michel, L.: The Objective-CP optimization system. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 8–29. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_5



Solution-Based Phase Saving for CP: A Value-Selection Heuristic to Simulate Local Search Behavior in Complete Solvers

Emir Demirović^(✉), Geoffrey Chu, and Peter J. Stuckey

School of Computing and Information Systems, University of Melbourne,
Melbourne, Australia

{emir.demirovic,pstuckey}@unimelb.edu.au

Abstract. Large neighbourhood search, a meta-heuristic, has proven to be successful on a wide range of optimisation problems. The algorithm repeatedly generates and searches through a neighbourhood around the current best solution. Thus, it finds increasingly better solutions by solving a series of simplified problems, all of which are related to the current best solution. In this paper, we show that significant benefits can be obtained by simulating local-search behaviour in constraint programming by using phase saving based on the best solution found so far during the search, activity-based search (VSIDS), and nogood learning. The approach is highly effective despite its simplicity, improving the highest scoring solver, Chuffed, in the free category of the MiniZinc Challenge 2017, and can be easily integrated into modern constraint programming solvers. We validated the results on a wide range of benchmarks from the competition library, comparing against seventeen state-of-the-art solvers.

1 Introduction

Large neighbourhood search (LNS) [14] is a widely used metaheuristic for constrained optimisation. A neighbourhood of a given solution is the set of solutions that can be obtained by performing perturbations on a target solution. The size of the neighbourhood is determined by the used perturbations. Conventional local search considers small neighbourhoods due to efficiency. However, since the scope of the search is narrow, such methods are prone to be trapped in local optima. In contrast, large neighbourhood search uses significantly larger neighbourhoods. Thus, the optimisation algorithm has more options to escape local optima, while retaining the advantages of local search. Different techniques may be used to explore the neighbourhoods, including systematic search methods such as constraint programming or problem-specific heuristics. The use of constraint programming for neighbourhood exploration is particularly suitable for highly constrained optimisation problems where propagation and systematic search are advantageous compared to heuristic algorithms.

Phase saving is an approach, originally from SAT solvers [11], where the last value assigned for a variable in the search is given priority the next time the variable is branched on. The advantage is that phase saving interacts well with restarting since it was presumably nontrivial to find the value before the restart.

Solution-based phase saving is different, and not so commonly applied (see e.g. [1]). It gives priority to the value the variable is assigned to in the last *solution* found. For satisfaction problems, this is meaningless, as the search terminates once a solution is found. In optimisation problems, however, this concentrates the search around the current best solution, just as in LNS.

We use solution-based phase saving with activity-based search and nogood learning as a means of focusing the search around the best solution, mimicking typical local search behaviour. An advantage of this approach is that standard CP machinery can be used; hence it can be easily incorporated in most CP solvers. The method bears similarities with large neighbourhood search, as activity-based search and restarts can be seen as defining a “neighbourhood”.

Our experimental results on a wide range of benchmarks from the MiniZinc Challenge 2017 demonstrate that by using the described approach, we can achieve significant improvements over Chuffed [4]. Furthermore, our approach can be easily integrated into modern constraint programming solvers, and it does not introduce additional parameters. To summarise:

- We introduce solution-based phase saving in constraint programming as a means to capture some of the benefits of LNS in a complete solving approach.
- We combine solution-based phase saving, activity-based search, Luby restarts, and nogood learning to obtain a complete CP algorithm that focuses its search around the best solution found so far. Our experiments show that activity-based search, perhaps unsurprisingly, is better suited for this task than random variable selection.
- We evaluate the proposed approach using benchmarks from the MiniZinc Challenge 2017 competition and compare with state-of-the-art solvers used in the competition. Overall, the results demonstrate that the approach is highly effective, improving the results for the highest scoring solver Chuffed.
- We discuss the relationship of our approach, as a contribution to search and black-box solvers, to large neighbourhood search.

2 Preliminaries

Constraint Programming. A *constraint satisfaction problem* (CSP) is a tuple $P \equiv (V, D, C)$, where V is a set of variables, D is a mapping from variable $v \in V$ to a set of values $D(v)$, and C is a set of constraints. An assignment θ assigns each $v \in V$ to an element $\theta(v) \in D(v)$. A *solution* is an assignment that satisfies all constraints in C . A *constraint optimization problem* (COP) is CSP augmented by an objective function f that maps each assignment to a value. The aim is to compute the *optimal solution* θ^* such that $\forall \theta' : f(\theta^*) \leq f(\theta')$.

Nogood Learning. Upon reaching a conflict, nogood learning solvers analyse conflicts to determine the assignments responsible for its cause. The reason for

failure is recorded in the form of a clause and added to the database. The mechanism allows *nonchronological backtracking*, where backjumps can take place several decision levels above the current level.

Restarts. To avoid searching extensively around local optima, solvers perform restarts after reaching a certain number of conflicts. The key is to strike a balance between diversification (frequent restarts) and intensification (infrequent restarts). Luby restarts [7] are widely used in SAT/CP solvers and aim to introduce a variety of restart frequencies, with smaller restarts being significantly more common, i.e. a partial Luby sequence is as follows: 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8...

Dynamic Search. The search algorithm repeatedly decides on the branching variables. A common approach is to select variable based on their recent activity in conflicts (VSIDS scheme [8]). After a conflict is detected, the *activity* of involved variables is increased and periodically a decay on all activities is applied. Thus, variables that we the cause of recent conflicts have priority. Other methods include *first fail* [5], *dom/wdeg* [2], and *impact-based search* [13].

Phase Saving [11]. As explained previously, when branching solvers must decide on a value assignment. A wide-spread approach for value selection, originally used in SAT solvers, is to choose the value used most recently for the variable. Therefore, after backtracking, the solver aims to return to its previous state as closely as possible. This behaviour is particularly well suited for restarts, as the solver can continue using the previous assignments instead of searching again. In addition, the information learned about its previous region of the search space through clause learning will still be relevant.

Large Neighbourhood Search [14]. The assignments of a subset of variables is fixed with respect to a given solution and a search algorithm, either local search or an exhaustive technique, is used to determine the best assignment for the remaining variables. The number of fixed variables is typically chosen adaptively: initially the algorithm selects a large number of variables to intensify the search and gradually diversifies by decreases the number of selected variables over the course of the algorithm.

3 Experiments

Solvers. We implemented solution-based phase saving for the value-selection strategy in the CP solver Chuffed [4]. In the experimentation, we compare with seventeen state-of-the-art solvers and their variants which were submitted to the MiniZinc Challenge 2017. For the sake of brevity we do not reference each solver individually but refer the interested reader to the competition.

Benchmarks. We used the 20 benchmarks with five instances of the competition. They encompass a wide range of problems.

Hardware, Time Limits, and Experimental Setup. We run the experiments using the same hardware and setting as in the *free search* category from

Table 1. Comparison of our approaches with Chuffed as the baseline solver.

Solver	score	iscore	area
chuffed-free	94.25	98.58	25887171
chuffed-random-free	76.63	77.17	40247361
chuffed-sbps-free	126.12	121.25	23332231

the MiniZinc Challenge 2017, where no restrictions are imposed on the solvers regarding search strategies. By doing so, we can position the approach directly to all solvers that participated in the competition.

Evaluation Metrics. We used the evaluation metrics of the Challenge. In short, *score* treats each benchmark instance as a vote between two solvers. It awards one point to the solver that found the better result and zero points for the other. If they perform equally well, the point is split in inverse proportion to their run times. The *score* for a solver is defined as the sum over all benchmark instances and considered solvers. The variation for incomplete solvers, *iscore*, ignores proof of optimality when comparing the performance of solvers.

The *area* score gives a measure of anytime performance of the solver. It computes the *area* under the curve defined by the function: 5000 if no solution found, $2500 \times (s - best)(worst - best) + 1250$ for a solution of value s where *worst* and *best* are the worst and best solutions found by any solver, and zero for proving optimality. This function effectively combines 25% of points for finding a solution, 50% for finding good solutions and 25% for proving optimality. It represents the area under the score curve over the twenty minutes run time, averaged across all benchmarks.

3.1 Comparison

A detailed breakdown of the results, in the style of the MiniZinc challenge results (minizinc.org/challenge2017/results2017.html), is available online: emirdemirovic.com/misc/cp18-sbps-comparison/.

Comparison with Chuffed: The first experiment compares variants of chuffed: *chuffed-free* is the competition version with Luby restarts and alternating free (activity-based) and fixed search; *chuffed-sbps-free* simply add solution-based phase saving to this; *chuffed-random-free* adds solution-based phase saving while using random variable selection in the free search; thus effectively mimicking random neighbourhoods (see Sect. 4).

We show the results in Table 1. Clearly, the use of solution-based phase saving significantly improves the performance over the baseline. Random neighbourhoods are not beneficial, as the advantages of solution-based phase saving are defeated by the random variable selection.

Looking more closely at the individual benchmarks: *chuffed-sbps-free* improves on all 20 benchmarks except *opt-cryptanalysis* where they are identical, and *cargo*, *crosswords*, *hrc* and *rc-graph-coloring*. The reason why

it underperforms is that the baseline approach can prove optimality of one or more instances where solution-based phase saving cannot. Clearly, solution-based phase saving is not as effective in proving optimality.

Interestingly, random neighbourhoods are preferable to activity-based ones on benchmarks `mario`, `opd`, and `rcpsp-wet`, showing its effectiveness for these particular situations. On other benchmarks the performance can be poor.

Comparison with All Solvers. `chuffed-free` was the highest points scorer in the free category of the Challenge. Comparing our variants, we find `chuffed-sbps-free` is clearly better than all other solvers and it reduces the area under the curve with respect to the baseline by 10%. Surprisingly, the use of random neighbourhoods is still ahead of all solvers except three, showing that solution-based phase saving is still powerful, even with a poor neighbourhood strategy.

Comparison with Local Search Solvers. Since LNS and local search perform well on the same problems, we compare the use of solution-based-phase-saving against the local search solvers in the competition, in particular on the problems where local search provided good results.

The results shown in Table 2 restrict the comparison to eight benchmarks where a local search solver ranked in the top three. Solution-based phase saving provides a substantial difference, pushing `chuffed` from below the performance of all local search solvers, to better than all of them. Interestingly, the low area suggests that `chuffed-free` finds good solutions early, but then gets stuck, where the local search solvers continue to improve. Solution-based phase saving is much better at continuing to improve solutions.

If we restrict our attention to the two problems where a local search solver was the best solver (`oscar-free` in both cases), we see that for these benchmarks even random neighbourhoods can improve on the baseline performance. Solution based phase-saving is not able to compete with the best local search in this case, but certainly markedly enhances the performance over the baseline (Table 3).

Table 2. Comparison of our approaches and local search solvers on benchmarks where local search solvers scored in the top three ranks.

Solver	score	iscore	area
<code>chuffed-free</code>	69.53	71.00	11043751
<code>chuffed-random-free</code>	49.42	51.00	17323701
<code>chuffed-sbps-free</code>	99.70	93.00	8198639
<code>izplus-par</code>	81.67	84.50	11368249
<code>oscar-free</code>	74.83	74.50	14734476
<code>yuck-free</code>	71.85	73.00	13638341

Table 3. Comparison of our approaches and local search solvers on the two benchmarks sets, `road-cons` and `opd`, where local search was the most effective method.

Solver	score	iscore	area
<code>chuffed-free</code>	19.88	23.00	3684082
<code>chuffed-random-free</code>	22.81	24.50	2831003
<code>chuffed-sbps-free</code>	26.45	27.00	2710511
<code>izplus-par</code>	8.80	11.50	6040284
<code>oscar-free</code>	45.40	40.00	795009
<code>yuck-free</code>	23.67	21.00	5853594

4 Relationship with Large Neighbourhood Search

We now contrast a modern CP solver using restarts, dynamic variable selection, and solution-based phase saving versus large neighbourhood search. We shall see that there are striking similarities between the two.

4.1 Restarts Versus Neighbourhood Size

Most uses of CP incorporate restarts to avoid being trapped in large useless parts of the search space. Restarts are managed by limiting some resource, such as time or number of conflicts. Once the limit is reached, the search is restarted. Limits usually increase over time to maintain completeness, e.g. using either geometric [15] or Luby [7] sequences. Restarting requires either nogoods [9] or randomisation in search to avoid repeating previous work.

LNS usually defines neighbourhoods by fixing a set of variables to their value in the best solution. The search of the neighbourhood continues until it finds a better solution or it thoroughly explores the neighbourhood. The size of the neighbourhood gives an implicit limit on the computation of this subsolve. In addition, LNS often explicitly restricts resource usage for the subsolve, to avoid cases where the neighbourhood defined is too large to explore exhaustively. Similarly, as for restarts, the limits are typically increased with time and randomisation in used to avoid searching through the same neighbourhood.

Therefore, both restarts and LNS tackle a series of subproblems while using randomisation and imposing limits on the computation for each subproblem.

Luby Restarts for Neighbourhood Size. Using small neighbourhoods can provide quick improvements but cannot escape local optima effectively, while large neighbourhoods provide the reverse effect. Thus, a balance between the two is often sought for, and many LNS algorithms adopt adaptive strategies to determine the size of the neighbourhoods. In CP solvers, the Luby [7] sequence can be used to determine the restart limits, and it achieves the desired behaviour: a balance between frequent and extended restarts. It simulates the adaptive strategies often seen in LNS.

4.2 Dynamic Search and Phase Saving Versus Neighbourhoods

In LNS, a significant subset of variables is selected, and the variables are assigned values according to an incumbent solution while leaving the remaining variables to the search strategy to explore. In CP solvers with dynamic variable ordering and solution-based phase saving, the search selects variables according to the variable-selection strategy and sets them to their value in the current best solution. Hence, the search will not fail until it fixes most variables since only the requirement for finding a better solution can invalidate the current best solution.

Thus, a similarity can be seen between the two approaches. The CP approach will fix almost all variables to their current best solution value, and afterwards, explore around this selection. Given the computation limits, it will backtrack only a subset of these decisions. Hence, it *implicitly* defines a neighbourhood given by the set of variables that are never reached during backtracking.

The first effective difference is that LNS may exhaust the neighbourhood before reaching its search limits, after which it terminates the search. In contrast, the CP approach will in effect *expand the neighbourhood* it explores, until hitting the restart limit. The second difference is that solution-based phase saving will always set a variable to its value in the best solution if possible, whereas in LNS this is not necessarily the case.

VSIDS as an Implicit Neighbourhood Selection Strategy. In LNS, it is crucial to select strongly related variables to avoid defining highly restrictive neighbourhoods with few solutions. In CP, activity-based search (VSIDS) tends to select connected variables as well, and when coupled with solution-based phase saving, we claim it implicitly builds “neighbourhoods”.

We make the following observation to show the type of neighbourhoods generated by VSIDS. When conflicts occur during the search, activities of involved variables will be increased. As VSIDS prioritises variables with high activities for branching, this will create a positive feedback loop as more conflicts among related variables will be generated, hence further increasing their activity. Moreover, the exponential decay rate in VSIDS ensures that a variable’s activity is largely determined by its most recent involvement in failure. Therefore, variables with similar activity values have been active at similar times. Conversely, related variables are likely to be active at the same time. Thus, as VSIDS branches on variables based on their activity, when coupled with solution-based phase saving, the resulting neighbourhood will consist of strongly connected variables with their values assigned as in the best solution found so far. This can be seen as *emergent LNS* behavior.

We note that solution-based phase saving with VSIDS has built-in diversification. Upon restart, variables that were previously selected first will have low activity values since they are unlikely to take part in many conflicts directly. Hence, they will not be selected early again, while the most active variables from the previous restart will now be placed at the top of the search tree. Thus, the variable-selection strategy will cycle through variables.

4.3 Further Differences

Large neighbourhood search has many variations, and not all of these are easily captured by solution-based phase saving, dynamic variable ordering, and restarts. We discuss this in the following text.

Local Objectives. While using the global objective within a neighbourhood is often appropriate, in some cases each LNS subsolve uses a different objective. This variation is particularly important when the global objective is likely to be fixed by variables outside the neighbourhood. The CP approach uses a single global objective and note that it cannot be “fixed” by the variable-selection procedure since that will cause failure, but this effectively means that some variables high in the dynamic ordering will be fixed to a different value than in the best solution.

Adaptive LNS. Often a set of different neighbourhoods are defined in LNS, which are used adaptively, biasing choices to those that lead to improvements during the search. However, while it would be possible to generate a dynamic variable ordering strategy that acts similarly, it is not standard.

Acceptance Heuristics. Variants of LNS will accept equally good solutions, so-called *side moves*, or slightly worse solutions (e.g. using a simulated annealing approach) to give more diversification to the search. Merely changing the variable ordering cannot achieve this.

5 Related Work and Conclusion

Solution-based phase saving can be used to mimic a local search strategy in CP solvers. While it is merely a value-selection heuristic, when combined with activity-based search and restarts, it is very similar to LNS. There are a number of other approaches to automatic neighbourhood generation.

In [10] neighbourhoods are created based on the propagation between variables. Neighbourhoods are built in two ways: by reduction or expansion. The first *reduction* approach iteratively selects a variable from a list of fixed size if possible and random otherwise. It is assigned its value in the best solution and variables whose domain was reduced by propagation of the assignment are added to the list. The next variable is chosen as that in the list with the most significant domain reduction. The process continues until the remaining problem is deemed small enough. The *expansion* approach works in the reverse direction.

The approach of [6] selects neighbourhood variables randomly with a bias towards those with high *impact* on the objective function. The rationale is that these variables are responsible for the current value of the solution. By changing their assignments, it can presumably obtain better solutions. The effectiveness of the approach is further improved by considering a combination of impact and *proximity*, where *proximity* follows a similar idea as closeness in [10].

The intuition is that impactful variables should be accompanied by related variables as otherwise the neighbourhood might be too restrictive.

In [12] neighbourhoods based on explanations arising from conflicts are investigated. The reasoning is that variables involved in conflicts are related, and hence form a suitable neighbourhood. This method is similar to VSIDS-based neighbourhoods, but here explanations are restricted to decisions, and they choose neighbourhoods based on the variables that lead to most conflicts, which is in some sense the opposite of the VSIDS approach.

A methodology for devising large neighbourhood search algorithms was presented in [3]. Unlike the previously discussed methods, it is not fully automated but instead offers guidelines for the design of large neighbourhood search algorithms. The authors suggest the following three principles: neighbourhood design should focus around the part of the problem that contributes the cost to the objective, several different adaptive neighbourhoods should be considered to ensure completeness of the approach, and learning techniques should be employed to determine the most effective combination of neighbourhoods and their resource limitations. The approach can be applied to a wide range of problems, and the authors demonstrate its effectiveness on job-shop scheduling.

Solution-based phase saving is a straightforward addition to a CP solver. It offers a substantial improvement on a wide range of benchmarks, significantly improving the best performing solver in the free category of the MiniZinc Challenge 2017, Chuffed. We expect other solvers to adopt solution-based phase saving as a powerful yet simple value selection strategy.

Acknowledgements. We would like to thank Andreas Schutt for his exceptional assistance with comparing solvers and Graeme Gange for his insight on the implementation.

References

1. Roig, I.A.: Solving hard industrial combinatorial problems with SAT. Ph.D. thesis, Technical University of Catalonia (UPC) (2013)
2. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: Proceedings of ECAI 2004, pp. 146–150 (2004)
3. Carchrae, T., Beck, J.C.: Principles for the design of large neighborhood search. *J. Math. Model. Algorithms* **8**(3), 245–270 (2009)
4. Chu, G.: Improving combinatorial optimization. Ph.D. thesis, The University of Melbourne (2011)
5. Haralick, R., Elliott, G.: Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.* **14**, 263–313 (1980)
6. Lombardi, M., Schaus, P.: Cost impact guided LNS. In: Simonis, H. (ed.) CPAIOR 2014. LNCS, vol. 8451, pp. 293–300. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07046-9_21
7. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. *Inf. Proc. Let.* **47**(4), 173–180 (1993)
8. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of DAC 2001, pp. 530–535 (2001)

9. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation via lazy clause generation. *Constraints* **14**(3), 357–391 (2009)
10. Perron, L., Shaw, P., Furnon, V.: Propagation guided large neighborhood search. In: Wallace, M. (ed.) *CP 2004*. LNCS, vol. 3258, pp. 468–481. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30201-8_35
11. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) *SAT 2007*. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72788-0_28
12. Prud'homme, C., Lorca, X., Jussien, N.: Explanation-based large neighborhood search. *Constraints* **19**(4), 339–379 (2014)
13. Refalo, P.: Impact-based search strategies for constraint programming. In: Wallace, M. (ed.) *CP 2004*. LNCS, vol. 3258, pp. 557–571. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30201-8_41
14. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M., Puget, J.-F. (eds.) *CP 1998*. LNCS, vol. 1520, pp. 417–431. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49481-2_30
15. Walsh, T.: Search in a small world. In: *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 1999*, pp. 1172–1177 (1999)



An SMT Approach to Fractional Hypertree Width

Johannes K. Fichte^{1(✉)}, Markus Hecher^{2(✉)}, Neha Lodha^{3(✉)},
and Stefan Szeider^{3(✉)}

¹ International Center of Computational Logic, TU Dresden, Dresden, Germany
fichte@tu-dresden.de

² Database and Artificial Intelligence Group, TU Wien, Vienna, Austria
hecher@dbai.tuwien.ac.at

³ Algorithms and Complexity Group, TU Wien, Vienna, Austria
{neha,sz}@ac.tuwien.ac.at

Abstract. Bounded fractional hypertree width (*fhtw*) is the most general known structural property that guarantees polynomial-time solvability of the constraint satisfaction problem. Bounded *fhtw* generalizes other structural properties like bounded induced width and bounded hypertree width.

We propose, implement and test the first practical algorithm for computing the *fhtw* and its associated structural decomposition. We provide an extensive empirical evaluation of our method on a large class of benchmark instances which also provides a comparison with known exact decomposition methods for hypertree width. Our approach is based on an efficient encoding of the decomposition problem to SMT (SAT modulo Theory) with Linear Arithmetic as implemented in the SMT solver Z3. The encoding is further strengthened by preprocessing and symmetry breaking methods. Our experiments show (i) that *fhtw* can indeed be computed exactly for a wide range of benchmark instances, and (ii) that state-of-the-art SMT techniques can be successfully applied for structural decomposition.

1 Introduction

A prominent research question is the identification of structural restrictions that make the constraint satisfaction problem (CSP) tractable [10]. Structural restrictions are concerned only in the way how constraints and variables interact, in contrast to language restrictions that are only concerned with the relations that appear in the constraints. Hybrid restrictions are concerned with both aspects.

In his seminal work, Freuder [21] showed that the CSP is tractable under structural restrictions imposed in terms of bounded treewidth of the constraint

The work has been supported by the Austrian Science Fund (FWF), Grants Y698 and P26696, and the German Science Fund (DFG), Grant HO 1294/11-1. Fichte and Hecher are also affiliated with the University of Potsdam, Germany.

graph. The following decades brought a phalanx of results that identified more and more general structural restrictions that still guarantee polynomial-time tractability of the CSP, some prominent notions are spread-cut width [11] and hypertree width [24]. This line of research found its culmination point in the work of Grohe and Marx [27, 28], who introduced the notion of *fractional hypertree width*, which generalizes all known structural restrictions that guarantee polynomial-time tractability of the CSP.

So far, fractional hypertree width was mostly of theoretical interest, because of the lack of practical algorithms for actually computing the associated decompositions. One can approximate the fractional hypertree width in polynomial time with a cubic error factor [37]. This is prohibitive for practical applications, since CSP algorithms that exploit (fractional) hypertree decompositions are exponential in time and space in the width of the decomposition [11, 24, 27, 28]. It is unlikely that one could compute the exact fractional hypertree width in polynomial time as checking whether a hypergraph has fractional hypertree width $\leq w$ is already NP-hard for $w = 2$ [20].

Contributions. In this paper we propose, implement and test the first practical approach to compute the fractional hypertree width. Our approach is based on an efficient SMT-encoding of the problem, and utilizes preprocessing and symmetry breaking methods. We establish an *ordering-based* characterization of fractional hypertree-width which is similar to the well-known elimination order characterization of treewidth (see, e.g., [8, 13]), which traces back to the work of Rose [39]. Ordering-based characterizations of treewidth have been shown to be well-suited for SAT encodings of treewidth and related width measures [4, 7, 36, 42], hence it was promising to establish such a characterization also for fractional hypertree width. This indeed turned out to be both feasible as well as effective. In fact, to encode the linear ordering as well as the hyperedges induced by the ordering, we could utilize the very same Boolean variables and constraints that have been used for treewidth encodings. However, for treewidth one needs to bound the cardinalities of certain sets of vertices, which in the existing encodings was accomplished by SAT-based cardinality constraints or Max-SAT formulations. For fractional hypertree width, however, we need to find certain *real-valued weights* of hyperedges and enforce lower and upper bounds on the sums of weights of certain sets of hyperedges. We found that these constraints can be handled well by the SAT modulo Theory (SMT) framework, in particular by SMT with Linear Arithmetic as implemented in the state-of-the-art SMT solver Z3 [38]. On top of the SMT encoding we also developed various *preprocessing* and *symmetry breaking* methods.

We would like to point out that for CSP instances of bounded fractional hypertree width, one can not only decide satisfiability, but also count the number of satisfying assignments in polynomial time, as observed by Duran and Mengel [15]. Hence also from a complexity theoretic point of view it is justified to use an SMT solver which operates in the class NP to facilitate the solution of a harder #P-complete counting problem.

We implemented our methods creating the prototype tool *FraSMT* and performed extensive experiments on benchmark instances which contain real-world instances from various application domains. To the best of our knowledge, there have not been any practical algorithms for fractional hypertree width reported in the literature. Thus we took as a reference point the algorithm *det-k-decomp* of Gottlob and Samer [26] for the related (but less general) parameter hypertree width, which in turn was shown to outperform the algorithm *opt-k-decomp* proposed earlier by Gottlob et al. [25].

Our results show that on an extensive collection of benchmark instances the new SMT approach clearly outperforms the known algorithm *det-k-decomp*, even without preprocessing or symmetry breaking. Adding these techniques gives again a significant performance boost.

In summary, our findings are significant as they show (i) that fractional hypertree width can indeed be computed for a wide range of benchmark instances, and (ii) that SMT techniques can be successfully applied for structural decomposition and outperform known methods.

2 Preliminaries

A *hypergraph* is a pair $H = (V(H), E(H))$, consisting of a set $V(H)$ of *vertices* and a set $E(H)$ of *hyperedges*, each hyperedge being a subset of $V(H)$.

For a hypergraph $H = (V, E)$ and a vertex $v \in V$, we write $E_H(v) = \{e \in E \mid v \in e\}$ and $N_H(v) = (\cup E_H(v)) \setminus \{v\}$; the latter set is the *neighborhood* of v . If $u \in N_H(v)$ we say that u and v are *adjacent*.

The *hypergraph* $H - v$ is defined by $H - v = (V \setminus \{v\}, \{e \setminus \{v\} \mid e \in E\})$.

The *primal graph* (or *2-section*) of a hypergraph $H = (V, E)$ is the graph $P(H) = (V, E_{P(H)})$ with $E_{P(H)} = \{\{u, v\} \mid u \neq v, \text{ there is some } e \in E \text{ such that } \{u, v\} \subseteq e\}$.

Consider a hypergraph $H = (V, E)$ and a set $S \subseteq V$. An *edge cover* of S is a set $F \subseteq E$ such that for every $v \in S$ there is some $e \in F$ with $v \in e$. A *fractional edge cover* of S (with respect to H) is a mapping $\gamma : E \rightarrow [0, 1]$ such that for every $v \in S$ we have $\sum_{e \in E, v \in e} \gamma(e) \geq 1$. The *weight* of γ is defined as $\sum_{e \in E} \gamma(e)$. The *fractional edge cover number* of S with respect to a hypergraph H , denoted $fn_H(S)$, is the minimum weight over all its fractional edge covers with respect to H .

A *tree decomposition* of a hypergraph $H = (V, E)$ is a pair $\mathcal{T} = (T, \chi)$ where $T = (V(T), E(T))$ is a tree and χ is a mapping that assigns each $t \in V(T)$ a set $\chi(t) \subseteq V$ (called the *bag* at t) such that the following properties hold:

- for each $v \in V$ there is some $t \in V(T)$ with $v \in \chi(t)$ (“ v is covered by t ”),
- for each $e \in E$ there is some $t \in V(T)$ with $e \subseteq \chi(t)$ (“ e is covered by t ”),
- for any three $t, t', t'' \in V(T)$ where t' lies on the path between t and t'' , we have $\chi(t') \subseteq \chi(t) \cap \chi(t'')$ (“bags containing the same vertex are connected”).

The width of a tree decomposition \mathcal{T} of H is the size of a largest bag of \mathcal{T} minus 1. The treewidth $tw(H)$ of H is the smallest width over all its tree decompositions.

We will frequently use the following well-known fact (see, e.g. [9]).

Fact 1. *Let (T, χ) be a tree decomposition of a graph G and K a clique in G , then there exists a node $t \in V(T)$ with $V(K) \subseteq \chi(t)$.*

Using this fact it is easy to see that $tw(H) = tw(P(H))$ holds for every hypergraph H .

A *generalized hypertree decomposition* of H is a triple $\mathcal{G} = (T, \chi, \lambda)$ where (T, χ) is a tree decomposition of H and λ is a mapping that assigns each $t \in V(T)$ an *edge cover* $\lambda(t)$ of $\chi(t)$. The *width* of \mathcal{G} is the size of a largest edge cover $\lambda(t)$ over all $t \in V(T)$. A *hypertree decomposition* is a generalized hypertree decomposition that satisfies a certain additional property which was added in order to make the computation of the decomposition tractable [24]. The *generalized hypertree width* $ghtw(H)$ of H is the smallest width over all generalized hypertree decompositions of H . The *hypertree width* $htw(H)$ is the smallest width over all hypertree decompositions of H .

A *fractional hypertree decomposition* of H is a triple $\mathcal{F} = (T, \chi, \gamma)$ where (T, χ) is a tree decomposition of H and γ is a mapping that assigns each $t \in V(T)$ a fractional edge cover $\lambda(t)$ of $\chi(t)$ with respect to H . The *width* of \mathcal{F} is the largest weight of the fractional edge covers $\lambda(t)$ over all $t \in V(T)$. The fractional hypertree width $fhtw(H)$ of H is the smallest width over all fractional hypertree decompositions of H .

To avoid trivial cases, we consider only hypergraphs $H = (V, E)$ where $E_H(v) \neq \emptyset$ for all $v \in V$. Consequently, every considered hypergraph H has a (fractional) edge cover and $fhtw(H)$ is always defined. If $|V| = 1$ then $fhtw(H) = 1$.

Since an edge cover can be seen as the special case of a fractional edge cover, with weights restricted to $\{0, 1\}$, it follows that for every hypergraph H we have $fhtw(H) \leq ghtw(H) \leq htw(H) \leq tw(P(H))$.

3 Ordering-Based Characterization of Fractional Hypertree Width

The first SAT encoding of treewidth was suggested by Samer and Veith [42], it uses an ordering-based characterization of treewidth. Also more recent SAT encodings of treewidth are ordering-based [4, 7]. In view of the success of ordering-based characterizations of treewidth, we developed an ordering-based characterization of fractional hypertree width, and used it for our SMT encoding. The remainder of this section is devoted to the definition of this characterization and a proof of its correctness. Kamis et al. [33] have suggested a similar characterization.

Let $H = (V, E)$ be a hypergraph with $n = |V|$ and $L = (v_1, \dots, v_n)$ a linear ordering of the vertices of H . We define the *hypergraph induced by L* as $H_L^n = (V, E^n)$ where E^n is obtained from E by adding hyperedges successively as follows. We let $E^0 = E$, and for $1 \leq i \leq n$ we let $E^i = E^{i-1} \cup \{e_i\}$ where $e_i = \{v \in \{v_{i+1}, \dots, v_n\} \mid \text{there is some } e \in E^{i-1} \text{ containing } v \text{ and } v_i\}$. We

consider the binary relation $Arc_L = \{(v_i, v_j) \in V \times V \mid i < j \text{ and } v_i \text{ and } v_j \text{ are adjacent in } H_L^n\}$. We write $Arc_L(i) = \{v_i\} \cup \{v_j \mid (v_i, v_j) \in Arc_L\}$, hence $Arc_L(i) = \{v_i\} \cup e_i$.

The *fractional hypertree width of H with respect to a linear ordering L* , denoted $fhtw_L(H)$, is the largest fractional edge cover number with respect to H over all the sets $Arc_L(i)$, i.e.,

$$fhtw_L(H) = \max_{i=1}^n fn_H(Arc_L(i)).$$

We would like to emphasize that in this definition the fractional covers are considered with respect to the original hypergraph H , and not with respect to the induced hypergraph H_L^n .

Figure 1 illustrates these concepts on a small example.

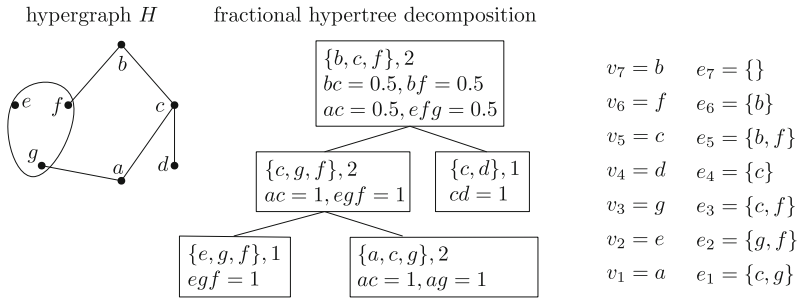


Fig. 1. An example illustrating a fractional hypertree decomposition of width 2 as well as the hyperedges e_i for $1 \leq i \leq 7$.

Theorem 1. *The fractional hypertree width of a hypergraph H equals the smallest fractional width over all its linear orderings, i.e., $fhtw(H) = \min_L fhtw_L(H)$.*

We establish the theorem by means of two lemmas below. Before doing so, we introduce some additional terminology.

Let $H = (V, E)$ be a hypergraph and $E' \subseteq E$ an edge cover of H . An E' -fractional hypertree decomposition of H is a fractional hypertree decomposition $\mathcal{F} = (T, \chi, \gamma)$ of H where each fractional cover $\gamma(t)$ assigns edges $e \in E \setminus E'$ the value 0. Similarly, the E' -fractional hypertree width of H with respect to a linear ordering L , denoted $fhtw_L(E', H)$, is computed by using only fractional edge covers that assign edges $e \in E \setminus E'$ the value 0, i.e.,

$$fhtw_L(E', H) = \max_{i=1}^n fn_{(V, E')} (Arc_L(i)).$$

The proof of the following lemma provides a decoding algorithm that efficiently computes a fractional hypertree decomposition from a given ordering.

Lemma 1. *Let $H = (V, E)$ be a hypergraph, $L = (v_1, \dots, v_n)$ a linear ordering of V , and $E' \subseteq E$ an edge cover of H . Then H has an E' -fractional hypertree decomposition of width $\leq \text{fhtw}_L(E', H)$.*

Proof. We proceed by induction on n . If $n = 1$ the statement is trivially true. Now assume $n > 0$ and that the statement holds for all smaller n . Let $w = \text{fhtw}_L(E', H)$. Let e_1, \dots, e_n and S_1, \dots, S_n as in the definition of a fractional hypertree width of H with respect to the linear ordering L .

We obtain from H the hypergraph H_2 by deleting v_1 and adding the hyper-edge e_1 . Furthermore, we obtain from E' the edge cover $E'_2 \subseteq E(H_2)$ of H_2 by removing v_1 from every edge in E' .

Now $L_2 = (v_2, \dots, v_n)$ is a linear ordering of H_2 , and we observe that its width cannot be larger than the width of L , since the sequence of sets $\text{Arc}_{L_2}(i)$ for $1 \leq i \leq n-1$ is exactly the same as the sequence of sets $\text{Arc}_L(i)$ for $2 \leq i \leq n$. Hence $\text{fhtw}_{L_2}(E'_2, H_2) \leq \text{fhtw}_L(E', H) = w$.

By induction hypothesis, it follows that H_2 has an E'_2 -fractional hypertree decomposition $\mathcal{F}_2 = (T_2, \chi_2, \gamma_2)$ of width $\leq w$. By definition of a tree decomposition, there must be a node $t_2 \in V(T_2)$ such that $e_1 \subseteq \chi_2(t_2)$. We define an E' -fractional hypertree decomposition $\mathcal{F} = (T, \chi, \gamma)$ of H as follows.

1. We obtain T by adding a new node t_1 to T_2 and making it adjacent with t_2 .
2. We set $\chi(t_1) = \{v_1\} \cup e_1 = S_1$ and $\chi(t) = \chi_2(t)$ for all other tree nodes t .
3. We choose for $\gamma(t_1)$ an E' -fractional edge cover of S_1 of smallest weight, which must be $\leq w$ since L was assumed to have weight w , and we set $\gamma(t) = \gamma_2(t)$ for all other tree nodes t .

We observe that (T, χ) satisfies all conditions of a tree decomposition, and conclude that \mathcal{F} is indeed an E' -fractional hypertree decomposition of H of width $\leq w$. □

Lemma 2. *Let $H = (V, E)$ be a hypergraph, $E' \subseteq E$ an edge cover of H and $\mathcal{F} = (T, \chi, \gamma)$ an E' -fractional hypertree decomposition of H of width w . Then there is a linear ordering $L = (v_1, \dots, v_n)$ of V such that $\text{fhtw}_L(E', H) \leq w$.*

Proof. As above we proceed by induction on n . We observe that the statement is trivially true if $n = 1$ or $|V(T)| = 1$. Now assume $n > 0$, $|V(T)| > 1$, and that the statement holds for all smaller n .

W.l.o.g., we may assume that for each leaf t of T there is some $v \in \chi(t)$ that does not belong to $\chi(t')$ for any other node $t' \in V(T) \setminus \{t\}$. Namely, if such a $v \in \chi(t)$ does not exist, then the properties of a tree decomposition imply that $\chi(t) \subseteq \chi(t'')$ for the unique neighbor t'' of t in T , and so all vertices and hyperedges covered at node t are also covered at node t'' , and t can be omitted.

Based on the above assumption, we conclude that there must be some $v_1 \in V$ which belongs to $\chi(t)$ for a leaf t of T , but v_1 does not belong to $\chi(t')$ for any other node $t' \in V(T) \setminus \{t\}$.

Let $e_1 = \{v \in \{v_2, \dots, v_n\} \mid \text{there is some } e \in E \text{ containing } v \text{ and } v_i\}$ and $S_1 = \{v_1\} \cup e_1$ (as in the definition of fractional hypertree width of H with respect to the linear ordering). Since $S_1 \subseteq \chi(t)$, $\gamma(t)$ gives an E' -fractional cover of S_1 with respect to H of weight $\leq w$, hence $\text{fn}_{(V, E')}(S_1) \leq w$.

We obtain the hypergraph $H_2 = (V_2, E_2)$ where $V_2 = V \setminus \{v_1\}$ and $E_2 = \{e \setminus \{v_1\} \mid e \in E\} \cup \{e_1\}$. We also define $E'_2 = \{e \setminus \{v_1\} \mid e \in E'\}$ which is an edge cover of H_2 . It is easy to see that from \mathcal{F} we can obtain an E'_2 -fractional hypertree decomposition $\mathcal{F}_2 = (T, \chi_2, \gamma_2)$ of H_2 of width $\leq w$ as follows.

1. We define $\chi_2(t) = \chi(t) \setminus \{v_1\}$, and $\chi_2(t') = \chi(t')$ for all other tree nodes t .
2. For every a hyperedge $e_2 \in E'_2$ we let $\gamma_2(t)[e_2] = \max\{\gamma(t)[e_1 \cup \{v_1\}] \mid e_1 \cup \{v_1\} \in E'\} \cup \{\gamma(t)[e_1] \mid e_1 \in E'\}$.

The induction hypothesis applies and hence we can conclude that there exists a linear ordering $L_2 = (v_2, \dots, v_n)$ of $V(H_2)$ such that $\text{fhtw}_{L_2}(E'_2, H_2) \leq w$. We now extend L_2 by adding v_1 at the first position and obtain the ordering $L = (v_1, \dots, v_n)$. We have already observed above that $\text{fn}_{(V, E')}(S_1) \leq w$, hence $\text{fhtw}_L(E', H) \leq w$. \square

Theorem 1 now follows by Lemmas 1 and 2 by taking $E' = E$.

4 SMT Encoding

In this section we describe an SMT encoding for the characterization of fractional hypertree decompositions as given in the previous section. The encoding is an adaptation of the Samer-Veith encoding of treewidth [42]. Given a hypergraph $H = (V, E)$ with $V = \{v_1, \dots, v_n\}$, we produce a formula $F(H, w)$ which is satisfiable if and only if the hypergraph V has a linear ordering L of V such that $\text{fhtw}_L(H) \leq w$.

The relation Arc_L can be computed in exactly the same way as Samer and Veith compute the “graph induced by the ordering.” We therefore use the same notation and introduce Boolean *ordering variables* $o_{i,j}$ for $1 \leq i < j \leq n$ and Boolean *arc variables* $a_{i,j}$ for $1 \leq i, j \leq n$.

An ordering variable $o_{i,j}$ is true if and only if $i < j$ and v_i precedes v_j in L . Consequently, to enforce that L is indeed a linear ordering, we must ensure transitivity, which can be accomplished with the following clauses (here $o^*(i, j)$ stands for $o(i, j)$ if $i < j$ and $\neg o(j, i)$ otherwise):

$$[\neg o^*(i, j) \vee \neg o^*(j, k) \vee o^*(i, k)] \quad \text{for } 1 \leq i, j, k \leq n \text{ and } i, j, k \text{ are distinct.}$$

The arc variables are used to represent the relation Arc_L for the ordering L represented by the ordering variables, where $a(i, j)$ is true if and only if $(v_i, v_j) \in \text{Arc}_L$, i.e., if $v_j \in \text{Arc}_L(i)$.

A straightforward encoding of the definitions of Arc_L gives rise to the following clauses:

$$\begin{aligned}
[\neg o(i, j) \vee a(i, j)] \wedge [o(i, j) \vee a(j, i)] & \quad \text{for } \{v_i, v_j\} \in E(P(H)) \text{ and } i < j, \\
[\neg a(i, j) \vee \neg a(i, l) \vee \neg o(j, l) \vee a(j, l)] \wedge [\neg a(i, j) \vee \neg a(i, l) \vee o(j, l) \vee a(l, j)] & \quad \text{for } 1 \leq i, j, l \leq n, i \neq j, i \neq l, \text{ and } j < l, \\
[\neg a(i, j) \vee \neg a(i, l) \vee a(j, l) \vee a(l, j)] & \quad \text{for } 1 \leq i, j, k \leq n, i \neq j, i \neq k \text{ and } j < k, \\
[\neg a(i, i)] & \quad \text{for } 1 \leq i \leq n.
\end{aligned}$$

Now, instead of cardinality constraints as used for treewidth, we use here real-valued weight variables representing the fractional covers. In fact, this makes the overall SMT encoding for fractional hypertree width even simpler and more compact than the SAT encoding for treewidth.

More precisely, we introduce a *weight variable* $w(i, e)$ for each $1 \leq i \leq n$ and $e \in E$, representing the weight of e in a fractional edge cover $\gamma_L(i)$ of the set $\text{Arc}_L(i)$, where L is the ordering represented by the ordering variables.

To ensure that $\gamma_L(i)$ is indeed a fractional edge cover of $\text{Arc}_L(i)$, we add the following two constraints; the first checks that all the vertices in $\text{Arc}_L(i) \setminus \{v_i\}$ are covered by $\gamma_L(i)$, the second checks that v_i is covered by $\gamma_L(i)$:

$$\begin{aligned}
[\neg a(i, j) \vee \sum_{e \in E_H(v_j)} w(i, e) \geq 1] & \quad \text{for } 1 \leq i \neq j \leq n, \\
[\sum_{e \in E_H(v_i)} w(i, e) \geq 1] & \quad \text{for } 1 \leq i \leq n.
\end{aligned}$$

Finally, we ensure that the weights of the fractional covers $\gamma_L(i)$ are at most w , $1 \leq i \leq n$, by means of the following constraints:

$$[\sum_{e \in E} w(i, e) \leq w] \quad \text{for } 1 \leq i \leq n.$$

This completes the construction of the formula $F(H, w)$. The formula $F(H, w)$ has $\mathcal{O}(n(n+m))$ variables where $\mathcal{O}(n^2)$ are Boolean variables and $\mathcal{O}(nm)$ are real variables, and $\mathcal{O}(n^3)$ clauses, where only $\mathcal{O}(n^2)$ are used for restricting the width.

In view of the construction of the formula and by Theorem 1 we obtain the following result.

Theorem 2. *A hypergraph H has fractional hypertree width $\leq w$ if and only if $F(H, w)$ is satisfiable.*

In view of the remark from the end of Sect. 2, we conclude that by replacing the real variables with integer variables yields an encoding for generalized hypertree width.

5 Preprocessing

In this section, we formulate several preprocessing methods. Some of them originate in the context of treewidth [4] and we adapted them for our purposes. It turned out that in some cases the preprocessing techniques decrease

the encoding size significantly. This not only speeds up the solving process but also extends the scope of our method to larger instances.

We exhaustively apply the following preprocessing rules R1–R4 in their order of occurrence.

R1: Contained Hyperedges. A hyperedge that is a subset of another hyperedge can be safely removed.

Proposition 1. *Let $H = (V, E)$ be a hypergraph, $e, f \in E$ be hyperedges such that $e \subsetneq f$, then $\text{fhtw}(H) = \text{fhtw}((V, E \setminus \{e\}))$.*

Proof. Consider a fractional hypertree decomposition $\mathcal{F} = (T, \chi, \lambda)$ of H with $\lambda(e) > 0$. We define a fractional hypertree decomposition $\mathcal{F}' = (T, \chi, \lambda')$ of the same width by setting $\lambda'(e') = \lambda(e')$ for $e' \in E \setminus \{e, f\}$ and $\lambda'(f) = \lambda(f) + \lambda(e)$. \square

R2: Biconnected Components. A hypergraph H is *connected* if for any two vertices $u, v \in V$ there exist vertices $v_1, \dots, v_k \in V$ such that $u = v_1$, $v = v_k$ and v_i and v_{i+1} are adjacent in H for $1 \leq i \leq k - 1$. H is *biconnected* if $H - v$ is connected for every $v \in V$. A *biconnected component* of H is a maximal biconnected hypergraph $H' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. Observe that two biconnected components of H can have at most one vertex in common.

We can split any hypergraph into biconnected components and compute the fractional hypertree width of each component separately.

Proposition 2. *Let H be a hypergraph and H_1, \dots, H_ℓ its biconnected components. Then $\text{fhtw}(H) = \max_{i=1}^k \text{fhtw}(H_i)$.*

We omit the easy proof due to space restrictions.

R3: Deletion of Vertices of Degree 1. A vertex of degree 1 (i.e., a vertex occurring in only one hyperedge) can be safely deleted.

Proposition 3. *Let $H = (V, E)$ be a hypergraph and $v \in V$ be a vertex of degree one, i.e., $|E_H(v)| = 1$, and $\text{fhtw}(H - v) \geq 1$. Then $\text{fhtw}(H) = \text{fhtw}(H - v)$.*

Proof. We know that $\text{fhtw}(H) \geq \text{fhtw}(H - v)$. For showing $\text{fhtw}(H) \leq \text{fhtw}(H - v)$, we take a fractional hypertree decomposition $\mathcal{F} = (T, \chi, \lambda)$ of $H - v$ and modify \mathcal{F} to obtain a fractional hypertree decomposition $\mathcal{F}' = (T', \chi', \lambda')$ of H . In particular, there has to exist node t in T with $\chi(t) = e \setminus \{v\}$, where $e \in E$ such that $v \in e$. Then, we construct \mathcal{F}' by taking \mathcal{F} , adding a fresh node t' as a child node of t to T' , and assigning $\chi(t') = e$ and $\lambda'(t') = 1$. Since $\text{fhtw}(H) \geq 1$, the claim sustains. \square

R4: Simplicial Vertices. Let $H = (V, E)$ be a hypergraph. A vertex $v \in V$ is a *simplicial vertex* of H if the neighborhood of v forms a clique in the primal graph of H .

We can remove a simplicial vertex v as long we maintain $fn_H(N_H[v] \cup \{v\})$ as a lower bound for the fractional hypertree width.

Proposition 4. *Let $H = (V, E)$ be a hypergraph and v a simplicial vertex of H . Then, $fhtw(H) = \max(fhtw(H - v), fn_H(N_H[v] \cup \{v\}))$.*

Proof. We proceed similarly to the proof of Proposition 3, where we modified a fractional hypertree decomposition \mathcal{F} for $H - v$ in order to obtain one for H . Here, however, the fresh decomposition node t' contains $N_H[v] \cup \{v\}$ in its bag. \square

6 Symmetry Breaking and Lower Bounds with Cliques

In this section we present the utilization of cliques in the primal graph for two purposes. First, we can choose any clique (i.e., a complete subgraph) in the primal graph and put the vertices of the clique at the end of the ordering. This can be seen as a symmetry breaking method that decreases the search space. In particular, it helps to speed up the optimality check (i.e., the $F(H, w)$ call when $w = fhtw(H) - 1$), as here the full search space needs to be explored. A similar technique has previously been used for a SAT encoding of treewidth [4].

For a hypergraph $H = (V, E)$ we call a set $S \subseteq V$ a *hyperclique* if S is a complete subgraph of the primal graph $P(H)$.

The next proposition ensures that we can indeed force a hyperclique to be the last in the ordering without effecting the fractional hypertree width.

Proposition 5. *Let $H = (V, E)$ and be a hypergraph and $S = \{v_1, \dots, v_\ell\}$ a hyperclique in H . Then, there is an ordering $L = (\dots, v_1, \dots, v_\ell)$ in which the vertices of S appear at the end, such that $fhtw_L(H) = fhtw(H)$.*

Proof. Let $\mathcal{F} = (T, \chi, \lambda)$ be a fractional hypertree decomposition of H of width $fhtw(H)$. By Fact 1 there is a node t in T with $S \subseteq \chi(t)$. We consider T to be rooted in t and construct a linear ordering L according to the proof of Lemma 2. Since we always pick vertices from bags from leaves of T , we will be left with t as the last tree node, and hence the vertices from $\chi(t)$ will be picked last. As a result, we obtain an ordering L , where vertices V' appear at the end and $fhtw_L(H) = fhtw(H)$. \square

Hypercliques can also be used to obtain a lower bound on the fractional hypertree width. If we want to compute the treewidth of a graph, and we know that the graph contains a clique on k vertices, then by Fact 1 we immediately know that the treewidth of the graph must be at least $k - 1$. However, in the context of hypergraphs and fractional hypertree width, we need to take the fractional edge cover number of the clique into account. Consider for instance a

hypergraph $H = (V, \{V\})$. It is easy to see that $fhtw(H) = 1$, although V forms a hyperclique. However, we still can show the following:

Proposition 6. *Let $H = (V, E)$ be a hypergraph and S a hyperclique of H . Then, $fhtw(H) \geq fn_H(S)$.*

Proof. Assume any fractional hypertree decomposition $\mathcal{F} = (T, \chi, \lambda)$ of H . Since S is a hyperclique, Fact 1 provides us a node t in \mathcal{F} whose bag contains S , i.e., $\chi(t) \supseteq S$. Then, by definition of fractional hypertree decompositions, every vertex of S is covered in t . As a result, the weight $\lambda(t)$ is at least $fn_H(S)$, and $fhtw(H) \geq fn_H(S)$. \square

We performed some experiments that suggest that the symmetry breaking works better with a hyperclique S with large $fn_H(S)$ than with a hyperclique S' where this number is small (e.g., a clique that is contained in a single hyperedge), even when S' is larger than S . Hence a hyperclique S with large $fn_H(S)$ serves two purposes: it facilitates symmetry breaking and provides us with a lower bound on the fractional hypertree width. However, the computation of a hyperclique S where $fn_H(S)$ is maximal is a very hard problem, hence we propose the notion of a k -hyperclique as a compromise.

A hyperclique S of a hypergraph $H = (V, E)$ is a k -hyperclique if no hyperedge of H intersects with S in more than k vertices. Intuitively, small values of k prevent large hyperedges, whereas bigger values provides us with flexibility, resulting in potentially larger cliques.

As already discussed, for symmetry breaking we rely on appropriate cliques. We aim to (i) fix parts of the ordering L of all the vertices of the clique (preferably large), (ii) to influence other bags as much as possible. The chances of L influencing other bags increase when we do not have large hyperedges, i.e., one hyperedge does not cover all the vertices of a large clique. Therefore, there is a tradeoff between finding large cliques, and avoiding large hyperedges.

In order to address this tradeoff we compute maximum cardinality k -hyperclique. We can detect a k -hyperclique of size at least ℓ for given hypergraph $H = (V, E)$ by means of a SAT encoding. Here k is assumed to be a small constant. For each vertex v we introduce a Boolean variable x_v , which is true if v belongs to the k -hyperclique. We then add the following constraints:

$$\begin{aligned} [\neg x_{v_1} \vee \neg x_{v_2}] & \quad \text{for any two vertices } v_1, v_2 \in V \text{ with } v_2 \notin N[v_1]; \\ [\neg x_{v_1} \vee \dots \vee \neg x_{v_k}] & \quad \text{for any } k \text{ vertices } v_1, \dots, v_k \text{ belonging to hyperedge } e \in E; \\ [\sum_{v \in V} x_v \geq \ell] & \quad \text{cardinality constraint for enforcing clique size at least } \ell. \end{aligned}$$

In the next section we will provide more details on how we have implemented the search for k -hypercliques.

7 Experimental Work

We performed a series of experiments on various publicly available benchmark sets, in order to obtain the fractional hypertreewidth of these instances, to evaluate whether our SMT-based approach fits well to obtain exact values on the

width, and to investigate how well our approach scales. The source code of our SMT-based decomposer¹, benchmarks, and detailed results² are publicly available.

7.1 Implementation

We implemented our encoding into our prototypical decomposer *FraSMT*. We used *Python 2.7.14* [40] based on an Anaconda³ distribution, which includes dependency handling for binaries packages. We used the graph library *networkX 2.1* [30], the answer-set programming solver *clingo 5.2.2* (*gringo 5.2.2* and *clasp 3.3.3*) [22], and the SMT solver *Z3 4.6.2* [38]. Our implementation consists of two separate tools: a validator and a decomposer.

Validator. The first part is a reusable validator that validates computed fractional hypertree decompositions or related decompositions such as tree decompositions and hypertree decompositions. The validator takes as input an extended version of the format used for the treewidth track of the Parameterized Algorithms and Computational Experiments Challenge (PACE) [14]. Since the graph library *networkX* does not support hypergraphs, we implemented hypergraph classes and classes that allow for a primal graph view on such a hypergraph. Both classes implement a *networkX*-like hypergraph API. Although it suffices to use rational numbers to compute the fractional hypertree width [27, 28], we still represent the maximum width by a real value since *Z3* does neither support rationals nor reals of arbitrary precision. As we may have a precision loss due to the internal representation of the reals [12], we check for width $w + \varepsilon$ for some small $\varepsilon \geq 0$. By default we set ε to 0.001.

Decomposer and Its Configurations. The second and main part is the decomposer *FraSMT*, which implements the preprocessing techniques, the SMT encoding, invoking the SMT solver, as well as reconstructing a decomposition from the solver assignments and outputting a decomposition (if possible; for details see below). Our decomposer *always* reduces contained hyperedges and splits a hypergraph into biconnected components and computes the width of each component separately. We optionally run finding and deleting degree 1 vertices as well as simplicial vertices. We refer to configurations that include this preprocessing with a string that contains “P”, whereas “p” indicates that this preprocessing technique is disabled (see Table 1). Further, our decomposer computes as a preprocessing step large k -hypercliques for symmetry breaking and for obtaining lower bounds, as discussed above. For this task we employ the answer-set programming (ASP) solver *clingo*, which supports (implicit) incremental solving and unsatisfiable core shrinking [1]. We use this to aim at a k -hyperclique of maximum cardinality. However, we limited the solving time (ten seconds) to

¹ See: github.com/daajoe/frasmt.

² See: [Benchmark repository](#) [16] and [results/raw data](#) [17].

³ See: <https://conda.io/docs/user-guide/install/download.html>.

determine such a clique. Still, at any time during the optimization (as long as at least one k -hyperclique has been computed), the solver is able to provide a large k -hyperclique. The ASP solver supports a natural encoding of cardinality constraints, and allows for incrementally computing maximum cardinality cliques among all k -hypercliques for $3 \leq k \leq \ell$ for some fixed ℓ . We thereby start with $k = \ell$ and then proceed (aiming at better lower bounds) by incrementally decreasing k by adding the necessary constraints to the ASP solver in multiple shots [22].

We then use such a resulting large hyperclique to apply symmetry breaking in our encoding as described in Proposition 5 and we use hypercliques to obtain additional lower bounds for the encoding. Moreover, we take the maximum width over the previously computed components and feed this value into the next computation. In that way we might obtain unsatisfiability and cannot output a decomposition, however, we cut the search space for the SMT solver as the solver does not necessarily need to find an exact solution in order to avoid an easy-hard-easy behavior. In the following configurations we use symmetry breaking as well as lower bounds. Finally, we implemented the encoding via a direct *Python* interface to the solver using additional features of *Z3*.

Other Solvers. In order to obtain results for hypertree width of our instances, we used a backtracking-based implementation *det-k-decomp* by Gottlob and Samer [26]. Since this implementation can only check for hypertree width of size at most k of an instance, we added a simple progression step on top⁴, which for every iteration reduces the result of *det-k-decomp* by 1 to check optimality.

7.2 Benchmark Instances

We considered a selection of 2191 instances, which contain hypergraphs that originate from CSP instances and conjunctive database queries from various sources. The hypergraphs contain up to 2993 vertices and 2958 hyperedges. The first set *DaimlerChrysler* consists of 15 instances, the second set *Grid2D* consists of 12 instances, and the third set *ISCAS'89* consists of 24 instances on circuits [26]. Moreover, the benchmarks contain 35 instances in the set *MaxSAT* [6] and two sets (*csp_application* and *csp_random*) of instances from the well known XCSP benchmarks [3] with less than 100 constraints such that all constraints are extensional. The set *csp_application* contains 1090 instances and the set *csp_random* contains 863 instances. Further, the set *csp_other* contains 82 instances, which have been collected for works on hypertree decompositions⁵. The set *CQ* consists of 156 instances from various conjunctive queries [2, 5, 23, 29, 34, 43]. About a quarter of the instances are graphs. Although *ftw* and *tw* coincide on graphs, these instances are still well-suited as benchmarks as they provide a challenge for the decomposer. All instances have been collected by Fischl et al. [19] (publicly available at [16]). We gratefully acknowledge him for providing this large collection of instances.

⁴ github.com/daajoe/detkdecomp.

⁵ <https://www.dbai.tuwien.ac.at/proj/hypertree/benchmarks.zip>.

7.3 Benchmark Setting

Hardware. Our results were gathered on Ubuntu 16.04 LTS Linux machines kernel 4.13.0-3 on GCC 5.4.1, both post-Spectre and post-Meltdown kernels⁶. We ran the experiments on a cluster of 16 nodes. Each node is equipped with two Intel Xeon E5-2640v4 CPUs consisting of 10 physical cores each at 2.4 GHz clock speed and 160 GB RAM. Hyper threading was disabled.

Setup and Limits. In order to draw conclusions about the efficiency of *FraSMT*, we mainly inspected the wall clock time. We set a timeout of 7200s and limited available RAM to 8 GB per instance. Resource limits were enforced by *runsolver* [41]. Due to hardware resource limitations we conducted only one run per instance and configuration. However, we benchmarked a few instances with multiple runs and observed no significant difference.

7.4 Results

We used a tool to gather data and control the benchmark generation, evaluation, and cluster setting [31]. We publicly provide all experimental data [17], including raw data such as all command line flags used, system sampling (RAM/sysload), standard output and standard error during the run.

Solved Instances/Runtime. Table 1 provides basic statistics on the benchmarks. The table contains the tested configurations of our decomposer and the number of solved instances for which we obtained the (fractional) hypertree width and

Table 1. Overview on the number N of instances for which the respective decomposer configuration outputted the exact (fractional) hypertree width of the instance within the given timeout. Configuration: c/C represents disabled or enabled symmetry breaking and lower bound techniques, respectively. p/P represents disabled or enabled preprocessing techniques. 0 represents that finding cliques was disabled. 4 and 6 represent that we used the ASP solver to search for a k -hyperclique of maximum cardinality with $k \in \{4, 6\}$. However, due to the imposed timeout, the solver might also just use an ℓ -hyperclique where $3 \leq \ell \leq k$. t median (avg, std) represents the median (average, standard deviation) of the runtime in seconds of the decomposer over all instances of our benchmark instances, including the timeouts.

config	N	$t[s]$ median	avg	std
<i>FraSMT</i> (C6P)	1449	1189	3124	3299
<i>FraSMT</i> (C4P)	1434	1187	3192	3326
<i>FraSMT</i> (C4p)	1282	1760	3461	3432
<i>FraSMT</i> (c0p)	1106	7200	4019	3398
<i>det-k-decomp</i>	838	7200	4672	3357

⁶ See: spectreattack.com.

Table 2. Distribution of the fractional hypertree width over the solved instances.

<i>fhtw</i>	1	(1, 2]	(2, 3]	(3, 4]	(4, 5]	(5, 6]	(6, 7]	(7, 8]	(8, 9]
<i>N</i>	145	123	198	255	308	273	65	81	1

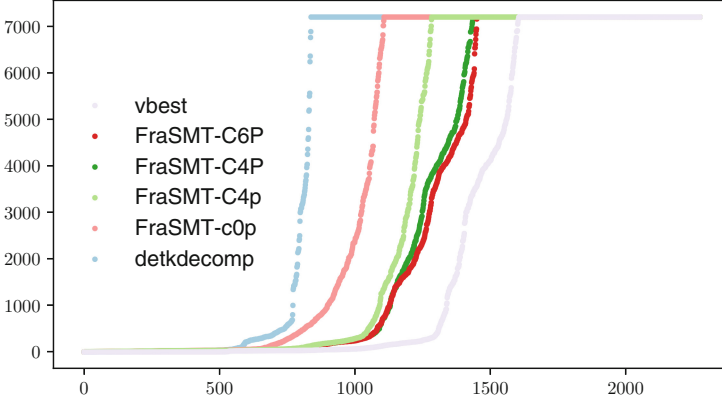


Fig. 2. Runtime in seconds on the considered benchmark instance. *vbest* refers to the virtual best solver. The x-axis labels consecutive integers that identify instances. The instances are ordered by running time, individually for each solver.

present total (average, minimum) runtime of the decomposer. We include time-outs of 7200s into the average and median. Figure 2 illustrates runtime results for the tested decomposer configurations as cactus plot. We solved instances that have up to 1453 vertices, up to 891 hyperedges, and up to hyperedges of size 16. The best configuration, namely *FraSMT(C6P)*, was capable of decomposing 1451 out of the total number of 2191 instances. Using *k*-hypercliques with *k* = 4 instead of *k* = 6 for symmetry breaking solved 1435 instances. Without preprocessing, *FraSMT* was able to solve 1283 instances. Without preprocessing or symmetry breaking, *FraSMT* could obtain 1107 fractional hypertree decompositions of exact width. Solver *det-k-decomp* was able to solve 838 instances, although both underlying methods are exact and *det-k-decomp* computes the less general parameter hypertree width in the same time. We further observe that by analyzing the virtual best solver (*vbest*), there are few instances a single best configuration cannot solve but can be solved by different configurations.

(Fractional) Hypertree Width. We computed the fractional hypertree width for our benchmarks using *FraSMT* and the hypertree width using *det-k-decomp*. The sets contain a few identical instances that occur in multiple sets. Even though we provide here only an overview on all instances, we decided to keep the duplicate instances to analyze the benchmark sets as provided from the original source for easier comparability. We provide detailed statistics online [17]. Using *det-k-decomp* we obtained the hypertree width for 838 instances. Table 2

provides the distribution of the number of instances and their respective fractional hypertree width. Considering all sets, 33% of the instances have fractional hypertree width below 4, 60% of the instances have fractional hypertree width below 6. Overall we were able to obtain the exact width for 66% of the instances.

Fractional Hypertree Width vs. Hypertree Width. When considering the obtained fractional hypertree width and hypertree width for these instances in our benchmark set that have been solved by both methods, the best *FraSMT* configuration and *det-k-decomp*, we observed a difference between the two width measures on 221 instances. The maximum difference was 2, and among these 221 instances the median difference was 0.6. However, since *det-k-decomp* could decompose significantly fewer instances and by construction works better on instances of small width, we expect the difference between *ftw* and *htw* to be significantly higher on the remaining instances.

8 Discussion and Conclusions

Our SMT-based encoding for fractional hypertree width, in combination with preprocessing and symmetry breaking methods, and its implementation, enable us to compute the exact fractional hypertree width for many realistic instances. This provides a significant step for making the theoretical notion of bounded fractional hypertree width, the most general known structural restriction for CSP that guarantees tractability, accessible for a practical use.

Our results show that a large majority of our considered benchmarked instances have low fractional hypertree width (below 10). However, we were unable to compute the exact width for about 33% of the instances. Consequently, we think that upper bound computations either using heuristics for hypertree width or modifying our encoding to obtain only upper bounds are of interest for future investigations.

Interestingly, we obtained the exact fractional hypertree width for more instances using our decomposer *FraSMT* than the exact hypertree width using *det-k-decomp*, although our decomposer determined the more general parameter. An important factor is the extensive preprocessing and symmetry breaking methods, which are not present in *det-k-decomp*, as these methods resulted in 16% more solved instances for our decomposition technique. However, even without preprocessing or symmetry breaking our approach solved more instances than *det-k-decomp*.

Since our results are limited to small and medium-sized hypergraphs up to about 1400 vertices, 900 hyperedges, and hyperedges of small size, heuristics or combinations of heuristics and exact methods might be interesting for practical purposes. Our techniques can be very helpful to evaluate the accuracy of heuristics. Efficient and precise heuristics would enable us to obtain a broad picture about available instances in CSP which might lead to a usage of fractional hypertree decompositions for solving actual CSP instances, in particular, for problems such as model counting in CSP. We believe that our methods can be

extended to compute the “fractional FAQ-width” which, when bounded, renders the Functional Aggregate Query (FAQ) problem tractable [33].

The focus of this paper was the exact computation of fractional hypertree width. However, we would like to point out that with our approach one can also compute good upper bounds on the fractional hypertree width by just skipping the expensive optimality check. We have reasons to believe that this will scale to significantly larger instances, since a similar behaviour has been observed in related work [18, 35]. We are interested to address this potential of our method systematically in future work.

References

1. Alviano, M., Dodaro, C.: Anytime answer set optimization via unsatisfiable core shrinking. *Theory Pract. Log. Program.* **16**(5–6), 533–551 (2016)
2. Arocena, P.C., Glavic, B., Ciucanu, R., Miller, R.J.: The iBench integration metadata generator. In: Li, C., Markl, V. (eds.) *Proceedings of Very Large Data Bases (VLDB) Endowment*, vol. 9:3, pp. 108–119. VLDB Endowment, November 2015. <https://github.com/RJMillerLab/ibench>
3. Audemard, G., Boussemart, F., Lecoutre, C., Piette, C.: XCSP3: An XML-Based Format Designed to Represent Combinatorial Constrained Problems. <http://xcsp.org> (2016)
4. Bannach, M., Berndt, S., Ehlers, T.: Jdrasil: a modular library for computing tree decompositions. In: Iliopoulos, C.S., Pissis, S.P., Puglisi, S.J., Raman, R. (eds.) *16th International Symposium on Experimental Algorithms, SEA 2017, 21–23 June 2017, London, UK, LIPIcs*, vol. 75, pp. 28:1–28:21. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
5. Benedikt, M., Konstantinidis, G., Mecca, G., Motik, B., Papotti, P., Santoro, D., Tsamoura, E.: Benchmarking the chase. In: Geerts, F. (ed.) *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2017)*, pp. 37–52. Association for Computing Machinery, New York, Chicago (2017). <https://github.com/dbunibas/chasebench>
6. Berg, J., Lodha, N., Järvisalo, M., Szeider, S.: MaxSAT benchmarks based on determining generalized hypertree-width. Technical report, MaxSAT Evaluation 2017 (2017)
7. Berg, J., Järvisalo, M.: SAT-based approaches to treewidth computation: an evaluation. In: *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, 10–12 November 2014*, pp. 328–335. IEEE Computer Society (2014)
8. Bodlaender, H.L.: A partial k -arboretum of graphs with bounded treewidth. *Theoret. Comput. Sci.* **209**(1–2), 1–45 (1998)
9. Bodlaender, H.L., Möhring, R.H.: The pathwidth and treewidth of cographs. *SIAM J. Discrete Math.* **6**(2), 181–188 (1993)
10. Carbonnel, C., Cooper, M.C.: Tractability in constraint satisfaction problems: a survey. *Constraints* **21**(2), 115–144 (2016)
11. Cohen, D., Jeavons, P., Gyssens, M.: A unified theory of structural tractability for constraint satisfaction problems. *J. Comput. Syst. Sci.* **74**(5), 721–743 (2008)
12. Committee, M.S.: IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pp. 1–70, August 2008

13. Dechter, R.: Tractable structures for constraint satisfaction problems. In: Rossi, F., van Beek, P., Walsh, T. (eds.) *Handbook of Constraint Programming*, vol. I, chap. 7, pp. 209–244. Elsevier (2006)
14. Dell, H., Komusiewicz, C., Talmon, N., Weller, M.: The PACE 2017 parameterized algorithms and computational experiments challenge: the second iteration. In: Lokshtanov, D., Nishimura, N. (eds.) *Proceedings of the 12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*, pp. 30:1–30:13. LIPIcs (2017)
15. Durand, A., Mengel, S.: Structural tractability of counting of solutions to conjunctive queries. *Theoret. Comput. Sci.* **57**(4), 1202–1249 (2015)
16. Fichte, J.K., Hecher, M., Lodha, N., Szeider, S.: A Benchmark Collection of Hypergraphs, June 2018. <https://doi.org/10.5281/zenodo.1289383>
17. Fichte, J.K., Hecher, M., Lodha, N., Szeider, S.: Analyzed Benchmarks and Raw Data on Experiments for FraSMT, June 2018. <https://doi.org/10.5281/zenodo.1289429>
18. Fichte, J.K., Lodha, N., Szeider, S.: SAT-based local improvement for finding tree decompositions of small width. In: Gaspers, S., Walsh, T. (eds.) *SAT 2017*. LNCS, vol. 10491, pp. 401–411. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_25
19. Fischl, W., Gottlob, G., Longo, D.M., Pichler, R.: HyperBench: A Benchmark of Hypergraphs (2017). <http://hyperbench.dbai.tuwien.ac.at>
20. Fischl, W., Gottlob, G., Pichler, R.: Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems. In: den Bussche, J.V., Arenas, M. (eds.) *Conference SIGMOD/PODS 2018 International Conference on Management of Data*, Houston, TX, USA, 10–15 June 2018, pp. 17–32. ACM (2018)
21. Freuder, E.C.: A sufficient condition for backtrack-bounded search. *J. ACM* **29**(1), 24–32 (1982)
22. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP Solving with Clingo. CoRR abs/1705.09811 (2017). <http://arxiv.org/abs/1705.09811>
23. Geerts, F., Mecca, G., Papotti, P., Santoro, D.: Mapping and cleaning. In: Cruz, I., Ferrari, E., Tao, Y. (eds.) *Proceedings of the IEEE 30th International Conference on Data Engineering (ICDE 2014)*, pp. 232–243, March 2014
24. Gottlob, G., Leone, N., Scarcello, F.: Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.* **64**(3), 579–627 (2002)
25. Gottlob, G., Leone, N., Scarcello, F.: On tractable queries and constraints. In: Bench-Capon, T.J.M., Soda, G., Tjoa, A.M. (eds.) *DEXA 1999*. LNCS, vol. 1677, pp. 1–15. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48309-8_1
26. Gottlob, G., Samer, M.: A backtracking-based algorithm for hypertree decomposition. *J. Exp. Alg.* **13**, 1:1.1–1:1.19 (2009)
27. Grohe, M., Marx, D.: Constraint solving via fractional edge covers. In: *Proceedings of the of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, pp. 289–298. ACM Press (2006)
28. Grohe, M., Marx, D.: Constraint solving via fractional edge covers. *ACM Trans. Alg.* **11**(1) (2014). Article 4, 20
29. Guo, Y., Pan, Z., Heflin, J.: LUBM Benchmark OWL Knowl. Base Syst. *Web semantics: science, services and agents on the world wide web* **3**(2), 158–182 (2005)
30. Hagberg, A.A., Schult, D.A., Swart, P.J.: Exploring network structure, dynamics, and function using networkx. In: Gäel Varoquaux, T.V., Millman, J. (eds.) *Proceedings of the 7th Python in Science Conference (SciPy 2008)*, Pasadena, CA, USA, pp. 11–15, August 2008

31. Kaminski, R., Schneider, M., Rabener, T., et al.: Benchmark-Tool (2017). <https://github.com/potassco/benchmark-tool>
32. Khamis, M.A., Ngo, H.Q., Rudra, A.: FAQ: questions asked frequently. In: Milo, T., Tan, W. (eds.) Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26–July 01 2016. pp. 13–28. Association for Computer Machinery, New York (2016)
33. Khamis, M.A., Ngo, H.Q., Rudra, A.: FAQ: questions asked frequently. CoRR abs/1504.04044 (2017). <http://arxiv.org/abs/1504.04044v6>. Full version of [32]
34. Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., Neumann, T.: How good are query optimizers, really? Proc. Very Large Data Bases (VLDB) Endow. **9**(3), 204–215 (2015)
35. Lodha, N., Ordyniak, S., Szeider, S.: A SAT approach to branchwidth. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 179–195. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_12
36. Lodha, N., Ordyniak, S., Szeider, S.: SAT-encodings for special treewidth and pathwidth. In: Gaspers, S., Walsh, T. (eds.) SAT 2017. LNCS, vol. 10491, pp. 429–445. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_27
37. Marx, D.: Approximating fractional hypertree width. TALG **6**(2) (2010). Article 17, 29
38. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
39. Rose, D.J.: On simple characterizations of k -trees. Discrete Math. **7**, 317–322 (1974)
40. van Rossum, G.: Python tutorial. CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995
41. Roussel, O.: Controlling a solver execution with the runsolver tool. J. Satisfiability Boolean Model. Comput. **7**, 139–144 (2011)
42. Samer, M., Veith, H.: Encoding treewidth into SAT. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 45–50. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_6
43. Transaction Processing Performance Council (TPC): TPC-H decision support benchmark. Technical report, TPC (2014). <http://www.tpc.org/tpch/default.asp>



On the Non-degeneracy of Unsatisfiability Proof Graphs Produced by SAT Solvers

Rohan Fossé^(✉) and Laurent Simon^(✉)

Labri, University of Bordeaux, UMR 5800, 33405 Talence Cedex, France
{rfosse,lsimon}@labri.fr

Abstract. Despite the important effort in developing fast and powerful SAT solvers, many aspects of their behaviors remains largely unexplained. We analyze the properties of learnt clauses derived by a typical Conflict Driven Clause Learning algorithm (CDCL) and study how they are linked to their ancestors, in the dependency graph generated by the resolution steps during conflict analysis and clauses minimizations. We show that all these graphs share a common structure: they are non k-degenerated with surprising large values, which mean they contain a very dense subgraph, the K-Core. We unveil the existence of large K-Cores, even on parallelized SAT solvers with clauses exchanges. We show that the analysis of the K-Core allows a good prediction of which literals will occur in future learnt clauses, until the very end of the computation. Moreover, we show that the analysis of the graph allows to identify a set of learnt clauses that will be necessary for deriving the final contradiction. At last, we demonstrate that the analysis of the dependency graph is possible with a reasonable cost in any CDCL.

1 Introduction

Since the introduction of the Conflict-Driven Clause Learning (CDCL) framework [13, 14], the SAT technology has entered a new era. Solvers are now relying on *lookback* techniques rather than *lookahead* ones, which make the analysis of current methods harder. At the age of the Davis Putnam Logemann Loveland (DPLL) procedure [4, 10], typically before the 2000's, the need for studying the behavior of algorithms was indeed less crucial. DPLL was a typical systematic backtrack search algorithm, relying on strong (costly and lookahead-based) heuristics for decisions: solvers were often spending most of their time at each node of the search tree computing their heuristics values to make careful decisions. As a consequence, the architecture of these solvers was mostly understood by the mathematical definition of the heuristics allowed to infer some general results on the size of the search tree [11], at least on random instances. On more structured problems, heuristics gave a very strong intuition explaining why the algorithm was working efficiently (*e.g.* branch on most frequent and balanced variables in shortest clauses, ...).

This work was supported by the French Project SATAS ANR-15-CE40-0017.

Within a few years, thanks to the introduction of the CDCL framework, the picture had considerably changed. The systematic backtrack search of the original DPLL is now ensured by the learning mechanism in CDCL, but aggressive clauses database cleanings [2, 6], very reactive heuristics and very fast restarts [7] makes the final procedure very complex to analysis. More importantly, the state of the search in a CDCL is not static anymore (*i.e.* based on some counters analyzing the current formula at a point of the search tree). The state of the search is now based on the entire past of the solver. In addition, each component of so called “Modern” SAT solvers are tightly connected together and any intrusive change in any of them may have considerable side effects on other components. It is thus very difficult to study these solvers, and we argue that we need to consider them as *complex systems*, needing an important experimental study in order to understand their strengths and weaknesses.

This work can be viewed as extending a few previous works [1, 8] that tried to connect theoretical measures with observed behavior of SAT solvers. However, no previous work has focused on some structural properties of generated proofs. In [1], they used a modified version of `satz` [10] to study the evolution of the space needed by the solver on a set of random and industrial problems. In another work [8], it was proposed to build a particular set of formulas (pebbling puzzles) to study the relationship between the minimal proofs for the initial formula and the behavior of CDCL solvers. The shape of proofs produced by CDCL solvers was also previously identified as a possible bottleneck for their efficient parallelization [9]. In some sense, we follow the same kind of idea in this paper but we focus on demonstrating a very particular structure of the proofs produced by sequential and parallel solvers. Our study is inspired by the work of [18] that already proposed to study the proofs, post-mortem, but by extending this work and focusing our study on the existence (and importance for the search) of a very dense subgraph (the K-Core) in all the proofs produced by CDCL SAT solvers.

Our hypothesis is that the existence of a K-Core strongly forces the search of the SAT solver, which implies that the study of K-Cores of dependency graphs is a necessary step towards a better understanding of SAT solvers. Our paper is thus an experimental paper, the aim of which is to report the existence – and the importance – of a dense subgraph in the dependency graph generated by SAT solvers and not (yet) to improve the performances¹ We organized our work as follows: after a few preliminaries presenting the essential notions of SAT solvers and K-Cores in graphs, we demonstrate the presence of K-Cores in dependency graphs produced by CDCL SAT solvers. Then, we show that the analysis of the dependency graph allows to make good predictions of the usefulness of learnt clauses. Then, we extend our findings in two directions. Firstly, we show that our results can be generalized to parallel proof. Secondly, we show that the analysis of the dependency graph can be done in any CDCL SAT solver with no additional memory required and at very small cost.

¹ For the reviewing process, figures are in colors. We plan to make two versions of the paper if accepted, with a black and white version of the figures for the proceedings.

2 Preliminaries

We assume the reader familiar with SAT but let us just recall here the global schema of CDCL solvers [3,6,14]: a branch is a sequence of decisions (taken accordingly to the VSIDS heuristic), followed by unit propagations, repeated until a conflict is reached. Each decision literal is assigned at a distinct, increasing, decision level, with all propagated literals assigned at the same decision level. Each time a conflict is reached, a series of resolution steps, performed during conflict analysis, allows the solver to extract a new clause to learn. This clause is then added to the clause database and a *backjumping* is triggered, forcing the last learnt clause to be unit and then propagated. Solvers also incorporate other important components such as preprocessing [5], restarts and learnt clause database reduction policies. It was shown in [2] that the strategy based on Literal Block Distance (LBD) was a good way of scoring clauses. The LBD is computed during conflict analysis: it simply measures the number of distinct decisions levels occurring in the learnt clause.

Parallel SAT solvers are roughly independent SAT engines (duplicating all their clauses) that can exchange clauses after each conflicts following some politics. Classically, binary clauses and clauses of small LBD are exchanged between solvers, even if more sophisticated techniques have been proposed.

2.1 Dependency Graph Induced by the Proof

The idea behind our study relies on the earlier work of [18], by extending it in many ways. We base our study on the same notion of *dependency graph* produced by a CDCL solver (we took `Glucose` as a reference, which the author of [18] gave us). Each conflict generates a new node in the graph (nodes in the dependency graph either match an original or a learnt clause, and for simplicity we may refer to a node or a clause for the same object in the following), and an (oriented) edge is added from each learnt clause to all its ancestors: all the clauses viewed during conflict analysis or clause minimization, oriented from the learnt clause to its ancestor.

If the notion of Dependency Graph (DG) holds for SAT and UNSAT formulas, we focus in this article on UNSAT formulas only (except in Sect. 6), for which the DG can be considered as an UNSAT (resolution) proof. Such a proof is thus here a direct acyclic graphs (DAG), with a subset of original clauses as leaves, learnt clauses as internal nodes and the empty clause at the top. We added the ability of `Glucose` to produce a DG. For this, we had to consider the special case of unary clauses. In `Glucose` and `Minisat`, unary clauses are simulated by adding a virtual decision at level 0, forcing the assignment of the literal occurring in the unit-clause. Here, we had to keep unit clauses in memory too, in order to keep track of them during conflict analysis.

Learnt clauses are totally ordered by the number of conflicts they were produced at. In addition, for each clause, once the total DG is generated, we compute a set of features. First of all, its *usefulness*. We call *useful* a clause necessary of the proof, and *useless* if it's not. Useful clauses are in other words the clauses

connected with the top clause. Do notice that, with this definition of usefulness, we do not consider useful a clause that would have been crucial for propagating a literal during the search, if no resolution was done on this literal during any conflict analysis. The generalization of DGs to parallel SAT solvers proofs is easy, by simply considering clauses exchanged by SAT solvers as unique, to keep track of the origin of each clause (parallel proofs will be considered only in Sect. 5).

In graph theory, a **k-degenerate** graph is an undirected graph in which every subgraph has a vertex of degree at most k . Similarly, a **k-core** of a graph G is an undirected subgraph of G in which all vertexes have degree at least k [16]. Thus, a k -core can be seen as a certificate for the non k -degeneracy of the graph. We are here interested only in the maximal k -core of graphs, such that the graph is not k -degenerate but $(k+1)$ -degenerate. We will use the notation K -core for the maximal k -core of the graph. As these notions are defined over directed graphs, we will simply consider DG as an undirected graph for computing them. We also defined the notion of *coreness* of a graph as the value of the largest k such that the graph is not k -degenerate.

2.2 Selection of UNSAT Problems

Modern SAT solvers can run for millions of conflicts, each one involving hundreds of resolution steps. They may thus quickly produce very large graphs preventing any costly analysis. In this article, we will consider two sets of problems. One suitable for in-depth analysis (described below), and one demonstrating that our findings on relatively small problems holds on larger classical problems (all problems from the last 5 competitions, from 2013 to 2017, see Sect. 6).

For the first part of our analysis, as aforementioned, we needed a set of not too hard, not too easy problems. For simplicity, we took the same approach as [18] and selected the same set of 60 problems from past competitions, selecting as many distinct series of problems as possible. The strategy to select the 60 UNSAT problems was to choose at least two benchmarks per family of problems that needed less than one million conflicts to be solved on the original formula (non shuffled, after preprocessing). In the same family, “harder” benchmarks were selected first (thus trying to limit the number of too easy problems). Once the benchmarks were selected, the `SatElite` preprocessing [5] was used on each of them. When mentioned, we also considered shuffled versions of these problems (random reordering of clauses order, literals positions in clauses and random renaming of literals). Without any other precisions, median values are considered when reporting statistics over shuffled problems. We used 50 shuffled instance per original problem. We used the exact same list of problems than in [18]. The interested reader can refer to this previous work for more details on the list of benchmarks.

Experiments were conducted on a cluster of Xeon E7-4870 processors from the *Mesocentre Aquitain de Calcul Intensif* with at least one hour CPU time (more CPU time will be allocated for simulating parallel solvers in the later).

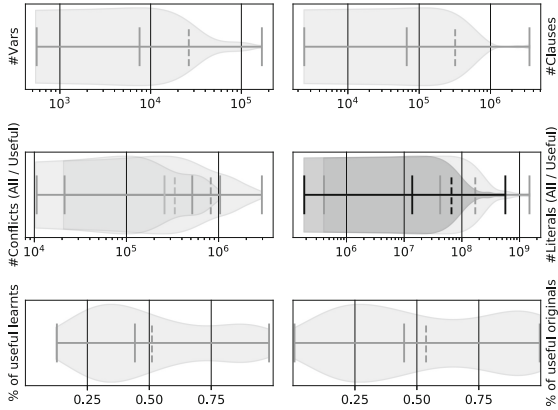


Fig. 1. Violin plots showing basic statistics on Dependency graph over the 60 problems. Dashed line is the mean value, middle line the median (over 50 shuffled instances). From left to right: Number of initial variables and clauses. Second line: number of internal nodes (or conflicts) and sum of learnt clause sizes. In light gray (more padded to the right) the total number and in darker gray the useful ones. Last line: percentage of useful learnt clauses and useful original clauses. All numbers are median values over shuffled and original problems.

2.3 Basic Dependency Graph Properties

Some important properties about DG were already identified in [18]. It was for instance observed that, on average, only 50% of learnt clauses were useful. Moreover, only 21% of clauses that were unit-propagated were seen in any conflict analysis. Given the fact that, in order to be used again in another conflict analysis, a clause must be unit-propagated again (due to the backjumping mechanism that unset all literals seen during last analysis), only a little more than 10% of unit propagations are used to derive a useful clause. If we approximate the time taken by a CDCL by the time taken by its unit-propagation engine (which is a reasonable assumption), we can thus observe that 90% of the time taken by a CDCL is “useless” (or only useful to update branching heuristics, ...). Understanding the characteristics of the Dependency Graph may be thus crucial to improve CDCL performances. Figure 1 shows some of the main characteristics of the formulas and the DG graphs we studied. It confirms the above conclusions made in [18]. We can also check that the formulas are indeed well chosen: they have different sizes and the median effort to solve them is around 500,000 conflicts, with a maximum of a few millions ones.

3 Characterization of K-Cores

To give a first intuition of the high density of DG generated by SAT solvers, we represent Fig. 2 a graphical representation of a very easy problem (less than 100,000 conflicts). Do notice the rotation of the graph around a very dense center:

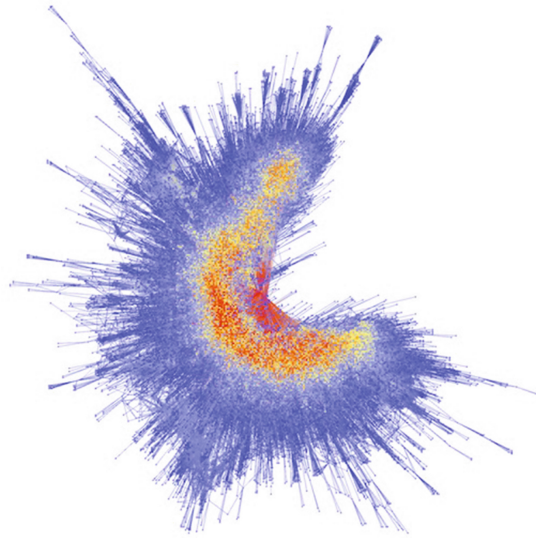


Fig. 2. Force-Directed layout of the Dependency Graph for the benchmark `een-pico-prop-05`. The color shows the *coreness* of the node.

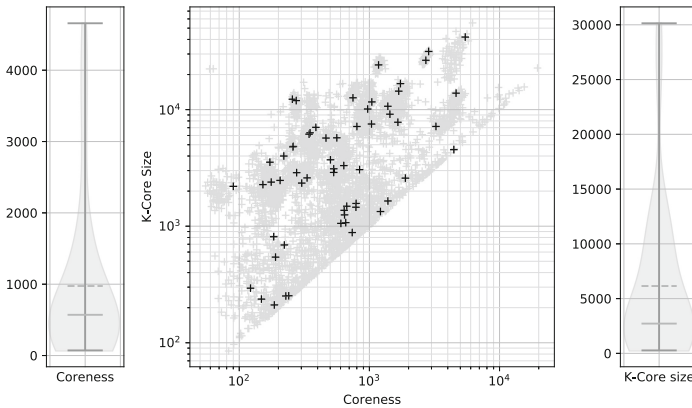


Fig. 3. Characteristics of K-Cores in the set of 60 problems. In the middle, darker plots are original problems, lighter shuffled problems. A log scale has been used to represent the wide range of obtained values. On left and right, we show the distribution of median values over shuffled instances.

the K-Core. Figure 3 shows the K-Core values (the coreness of the graph) and the sizes of the corresponding subgraph on the set of 60 problems, including shuffled ones. Let us first focus on the two violin plots. We can see (left) that the K-Cores can be very large. The median is larger than 500 and some problems can have K-Core values of a few thousands. On the right, we can see that the dense

subgraphs can be quite large too, containing typically more than 2,000 clauses. Now, if we focus on the scatter plot, it is striking to see how many problems are very close the $y = x$ line. Those problem have a K-Core size that is very close to the K-Core value, showing almost a clique of clauses as a K-Core.

3.1 Evolution of K-Cores Along the Computation

Let us now study how the K-Core evolves along the computation. Figure 4 shows its characteristics at two points of the search w.r.t its final values. For this, we computed the K-Core values considering (1) only the first 20,000 conflicts and (2) only the first half of the conflicts. The first set of points (after 20,000) conflicts shows that the computed values can be very far from the final values. However, it is very encouraging to notice that, at half of the run, the values are really close to the final values, showing that a study of the K-Core at this point of the computation may probably provide good informations about the final K-Core.

One of the main hypothesis in this work is that the K-Core is strongly forcing the search of the SAT solver into a close search space. To illustrate this, let us discuss now the results shown Fig. 4-right. The figure reports the CDF of the distance of a certain percentage of learnt clauses to the K-Core, in terms of resolutions. Clauses in the K-Core are at distance 0. A learnt clause obtained by resolution with at least a clause of distance n is at most at distance $n + 1$. Similarly, a clause used in conflict analysis to produce at least a clause of distance n is, also, at most at distance $n + 1$ (we thus consider distances on the undirected DG). On Fig. 4-right, each curve correspond to a CDF plot. The CDF for P% shows how many problems (x) have at least P% of its learnt clauses at distance smaller than y . We considered here all the problems (original and learnt). We can see that, in the very large majority of the cases, at least half of the learnt clauses are at distance 5 or less. It is also striking to notice that, on our selection

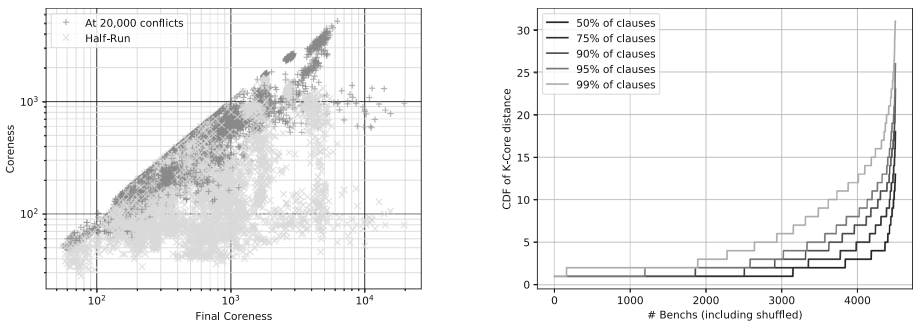


Fig. 4. (Left). Evolution of the K-Core characteristics after 20,000 conflicts and at the middle of the run (in terms of conflicts). (Right) Distance of the clauses w.r.t. the K-Core. Original and 50 shuffled versions of the 60 problems are considered. A very large fraction of the learnt clauses are often very close to the K-Core.

of problems, half of the problems have 99% of the learnt clauses at distance 5 or less. A distance of 15 resolutions seems also to be a very good bound in the majority of the cases. Most of the learnt clauses are very close to the K-Core in terms of resolutions.

3.2 K-Cores Structure

Let us now say a few words on the structure of the K-Core by reading the Fig. 5. First of all, let us point out that it is composed by original and learnt clauses, with a majority of learnt clauses, but with more than 30% of original clauses (see the median and mean values of the top-right violin plot). More surprisingly, despite its central role in the creation of all the learnt clauses (see section above), it is not entirely composed of useful clauses (see top-left violin plot): only around 75% of its clauses are used to derive the final contradiction. Let us now focus on the sizes of clauses in K-Core (see the 4 bottom plots of Fig. 5). By construction, original clauses are often limited in size (SAT encodings may prevent very large clauses to be built, and it is common to have a very large majority of binary clauses), but also contain a few large clauses. However, it is striking to see how short original clauses are in the K-Core (the median of the clauses are binary or ternary clauses only). This is emphasized with the comparison of the same statistics values, but not restricted to original K-Core clauses: the plot in darker gray shows these values computed on all the original clauses of each problem. We can thus see that original K-Core clauses are indeed short, compared to

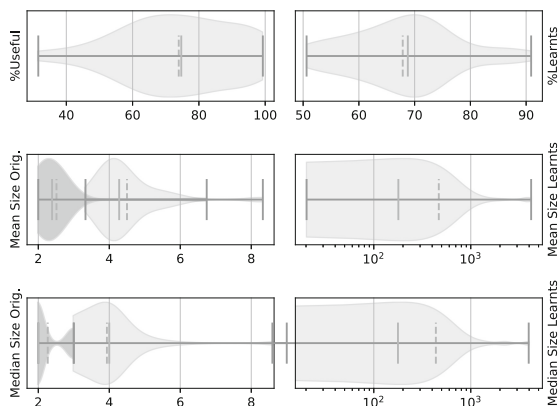


Fig. 5. The two top violin plots represents the percentages of useful clauses (left) and learnt clauses (right). The 4 others violin plots summarize the sizes of the clauses in the K-Core, split between original and learnt clauses. Darker gray violins (padded at the left) are the same values on the whole original problem (instead of original clauses in K-Core in light gray). A log scale has been used for reporting sizes of learnt clauses in the K-Core. All numbers are median values over 50 shuffled and original problems over our set of 60 problems.

the formula. As opposite, we had to use a log scale to represent the very large discrepancy in the size of K-Core learnt clauses. Here, clauses larger than 100 are frequent.

Two conclusions can be drawn from this experiment. Firstly, it is interesting to notice that some clauses of the K-Core, despite its central role in the learning mechanism, are useless. This is potentially an interesting point for improving SAT solver performances: identifying these clauses could potentially help the solver not generating useless clauses. Secondly, the fact that the K-Core contains very large clauses casts also a new light on detecting important clauses. We also observed that K-Core clauses are not necessarily clauses of small LBD (experiment not reported here), and thus there is a high chance that they could be removed during the clause database reduction. In order to explain this apparent paradox, we make the hypothesis that these clauses stay because of an implementation trick in the clause database reduction: in `Glucose`, only clauses of small LBD are kept but things are a little bit more complicated. In fact, the clause database reduction is generally triggered right after a regular backjump after a conflict analysis, but not after a restart. During the reduction, clauses that are currently unit-propagated are not removed. Thus, very large clauses that are very likely to be unit-propagated are also likely to be kept, which is probably the case of K-Core clauses.

4 On Predictions Based on Dependency Graph Analysis

The existence of large K-Cores is something that is not uncommon in real-life graphs, in which its analysis can even be used to detect, for instance in social graphs, to efficiently detect the most important nodes [17], *i.e.* nodes that have the most influence on other nodes. We tried a few measures and report here two interesting results we obtained. The first one tries to identify which clauses will be useful. The second measure tries to identify literals that will occur frequently in learnt clauses until the very end of the computation. In both experiments, we report the analysis of the DG after 20,000 conflicts and at half-run, by simply removing from DG all the nodes and edges added after the limit.

4.1 Predicting Useful Clauses

In this experiment, we used a flow algorithm on the DG, without considering the K-Core. We initialize all root nodes (clauses without descendants) with a constant weight and propagate the weight of each node to its ancestors by dividing the weight equally between them (following a topological order). For each node, the idea is to measure the clauses that occurs in the maximal number of paths to a root node (the graph can have many root nodes before the end).

The results we obtained are summarized Fig. 6 (left and right). Figure 6-left shows the results of our prediction by ranking 10,000 clauses according to the above flow values. Of course the darker curves (finished rune) give good results, even if they must not be considered as having any practical interest (it is easy

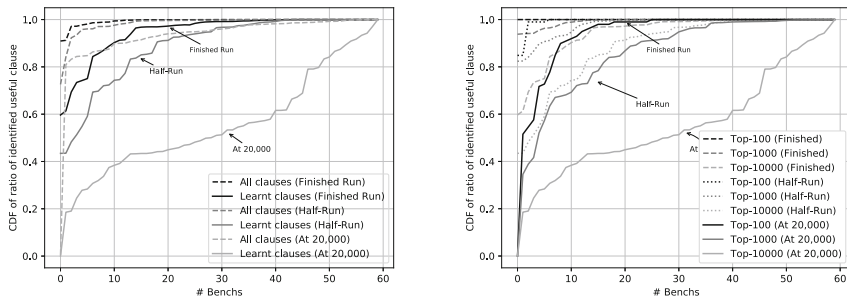


Fig. 6. (Left): Fraction of clauses that have been predicted as useful and that are useful. “All clauses” includes 10,000 clauses that can be original or learnt. “Learnt clauses” restricts the computation to only learnt clauses. A prediction is made after 20,000 conflicts, at half-run, and at the end of the run. Reported values are median over original and 50 shuffled problems. (Right): Other parameters for the same experiment. Top-10000 curves are also represented on the left figure.

at the end to detect useful clauses with no errors by another method). However, we can see that a simple flow algorithm, with no knowledge of which root node is the contradiction, can already identify useful clauses with a good precision.

The second observation follows the bottom light gray curve (at 20,000 conflicts), which clearly gives very bad results for identifying useful learnt clauses. This is in fact not surprising given the fact that we try to identify 10,000 good learnt clauses after only 20,000 conflicts. Here, our technique is possibly not better than a random guess. However, if we try to predict 10,000 useful clauses including original ones, we see that we correctly guess 80% of the 10,000 clauses, probably because we mostly bet on original clauses. This is already encouraging: we can identify useful original clauses at the very beginning of the computation. We can imagine, for instance, a search procedure that focuses on working on these clauses in priority. More importantly, we can see that, at half of the computation, results are very good: on half of the problems, we correctly guess at least 90% of useful learnt clauses over 10,000 guesses. This is even better when considering original clauses too.

Let us now focus on Fig. 6-right. In this figure, we try to identify 100, 1000 or 10,000 clauses. What this figure shows is that, if detecting 10,000 clauses can somehow give bad results, we can identify, with a great confidence, 100 learnt clauses that will be useful in the future, even after only 20,000 conflicts. At half-run the prediction is even better (see the dotted CDF line). We have around 95% good classification for 100 learnt clauses.

Let us temperate our findings. In fact, despite the importance of being able to detect a useful clause early in the computation (for instance for splitting the search space), it did not allowed us to improve `Glucose` yet. This is probably due to the fact that useful clauses may still have be deleted with no harm. It is for instance possible that only a descendant of the clause is needed for the remaining

computation, and thus our attempts to protect this clauses for further deletion did not helped. We are however still hopping that this kind of detection could help solvers to work on only part of the initial formula.

4.2 Detecting Future Learnt Clauses

In the above study, we were not able to detect which clauses will be important for the future of the search. In the following experiment, we try to guess which variables will occur in the last learnt clauses, just before deriving the final contradiction. For this, we propose to study the variables occurring in the K-Core at half of the computation. One first idea could be to consider all K-Core’s variables, but, as we can see on the top-right violin plot Fig. 7, some problems can have large K-Cores, containing all their variables. We thus propose to simply consider only the most frequent variables in the K-Core (the 5, 10 and 20 most frequent variables, respectively). The middle-right violin plot shows how frequent the top-5 variables are (median larger than 800) in K-Cores.

Let us now comment the left part of Fig. 7. These plots report the percentage of top variables we observe in the very last learnt clauses of the computation. The top plot shows the percentage of the top-5 variables that occurs in the last 20 clauses, w.r.t the total number of literals in the last 20 clauses (some clauses can be very long, see the bottom-right plot). We can see that, on average, 1% of the literals in the last 20 learnt clauses are from the top-5 variables in the K-Core. Numbers are of course better if we consider the top-10 and top-20 variables (on the violin plots below, we also plots the above distribution plots to emphasize the differences in the prediction with 5, 10 and 20 variables). We already observe a pretty good prediction, given the very large number of variables of our problems (median at 8,000 variables, see Fig. 1). We can also report our predictions by the

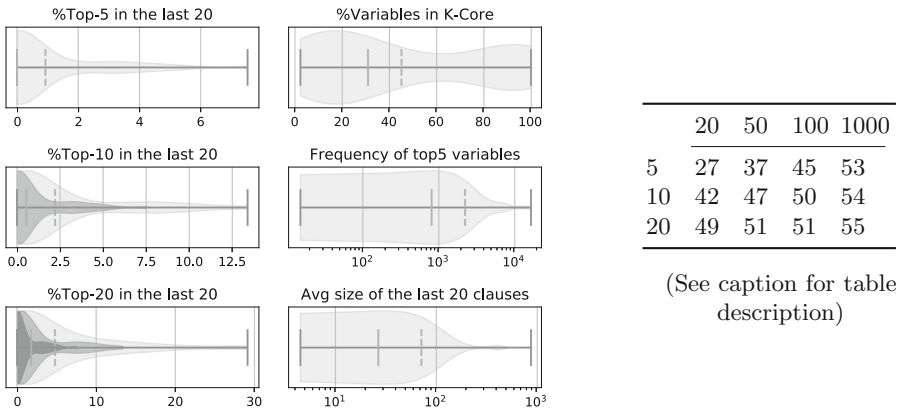


Fig. 7. (Left) Results of our prediction for literals occurring in the last learnt clauses. (Right) Over 60 problems, how many problems have at least one of the top-Y variables (rows) in the last X learnt clauses (columns)

lens of the table, right side of the same figure. We can see that for 27 problems over the 60 ones, at least one of the top-5 variables we identified occurred at least once in the last 20 clauses. At least a variable from the top-20 occurs in the 20 last learnt clauses, for 49 problems over the 60 (median values on all original and shuffled problems).

We did not have enough space to report another interesting experiment we also performed, but we noticed that, generally, the top variables are assigned by learning a unit clause only at the very end of the computation.

5 Analysis of Parallel Proofs

As we mentioned it in the first sections of this paper, sequential CDCL solvers are not well understood. The question is even more critical for parallel SAT solvers. They are generally a portfolio of SAT solvers exchanging clauses, based on some criterion (size, LBD are the most common ones). Clauses are thus shared amongst threads but no study have ever been conducted on the impact of parallelizing search over the final proof. Here, we study how the proof evolves w.r.t. the number of threads. For this, we simulated a parallel SAT solver on the top of our `Glucose` solver, hacked to handle dependency graphs. In our simulation, each simulated thread successively generates a clause by conflict analysis and offer to share it. Before each conflict, each thread can thus pick any last learnt clause by the other threads. In our setting, we imported clauses that had size strictly smaller than 8 and LBD strictly smaller than 4. We keep track of the origin of each imported clause. In the DG, imported clauses are not duplicated.

Let us now describe Fig. 8. On the two topmost sub-figures, each plot is a violin plot reporting a value for 1, 2, 4, 8, 12, 16, 20 and 24 cores, based on runs on the original problems only. We can firstly verify that parallelizing does not change the property of the proof: we have large K-Cores with a high coreness value. It can be observed that the K-Core seems to increase in comparison to the sequential version. Another finding is the evolution of the depth of the proof (initial clauses are of depth 0). Clearly enough, the depth is smaller as the number of cores increases. On the second sub-figure, we can see that, as opposite, the size of the proof tends to increase: proofs are shallower but larger as the number of threads increases. It is also interesting to measure how much effort is wasted when going parallel. “Useless learnts” reports the number of learnt clauses that do not occur in the proof. Clearly enough, the more we add threads, the more we produce useless clauses. This is emphasized by the “Efficiency” plot, that report the percentage of useful learnt clauses over the total number of learnt clauses. This clearly confirm the loss of efficiency in parallel solvers.

At last, we study, thanks to the bottom sub-figure of Fig. 8, the composition of the K-Cores in terms of clause origins (which thread produced the clause) with 24 cores. For each run, we measured the percentage of clauses from each thread in the final K-Core, and sorted it. Then, on the figure, we represent each curve in a cumulative way, normalized to 1.0. The result is in fact simple to read. Most of the problems have a K-Core composed by clauses from all the threads.

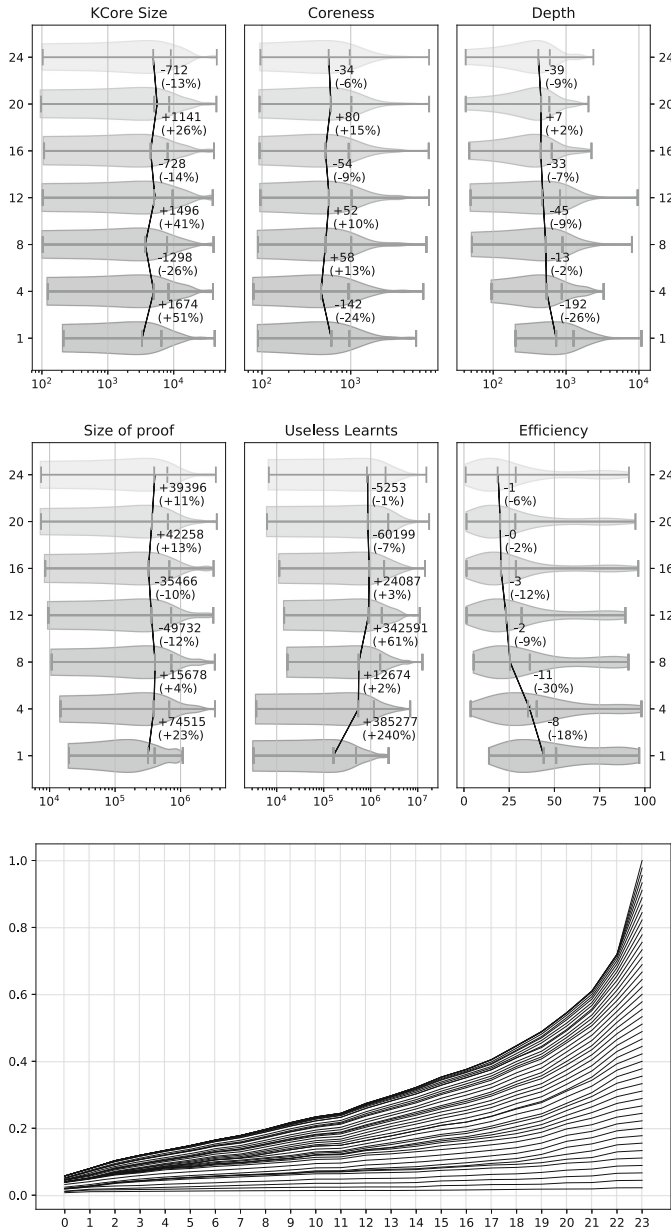


Fig. 8. (Two top figures) Characteristics of K-Cores when using from 1 to 24 threads. Changes in medians are kept track for each violin plot. Experiments are done on the 60 original problems, not shuffled. (Bottom) Cumulative plot of sorted origin of clauses in a 24-threads parallel solver over the 60 original problems

Some threads have a stronger presence in the Kcore, but not really significantly, except for one thread, especially on a small set of problems (most of the curves are almost “parallel” except for a few curves at the top of the plot: a perfect parallel curve to the curve below means that all threads are equally represented in the K-Core). Our conclusion is that, somehow surprisingly, the K-Core is composed of all the cores, despite the strong bottleneck in clauses exchanges. Even in a parallel setting, the presence of a K-Core strongly guide the search.

6 Fast Dependency Graph Analysis

Until know, we used a modified version of `Glucose` that kept track of all the dependencies (even for removed clauses). The memory consumption of our technique is thus not suitable for improving any CDCL performances: it is simply too costly to maintain all the informations. The goal of this short section is to show that is possible to analyze a simplified version of the DG in a regular CDCL with almost no cost. The idea is to built the DG just once, by using vivification on all the learnt clauses in memory at the time we want to build the DG [12, 15] (of course, a lot of learnt clauses have been removed). This step, performed once in our setting after 110,000 conflicts, allows us to find reasons for each learnt clauses in memory (this is the same method that is somehow used when rebuilding DRAT proofs [19]). Of course we cannot find the original dependencies but, as we can see Fig. 9 we were able to find very similar results as previously reported (large coreness of the DG), despite the partial information we have about the current proof (limited to the set of current learnt clauses). Let us point out here that we were able to compute this values for all the problems from the 5 last SAT competitions (2013–2017 included). Do notice also that we included in this experiment SAT and UNSAT problems, as well as problems which timed out after the computation of the K-Core. This last

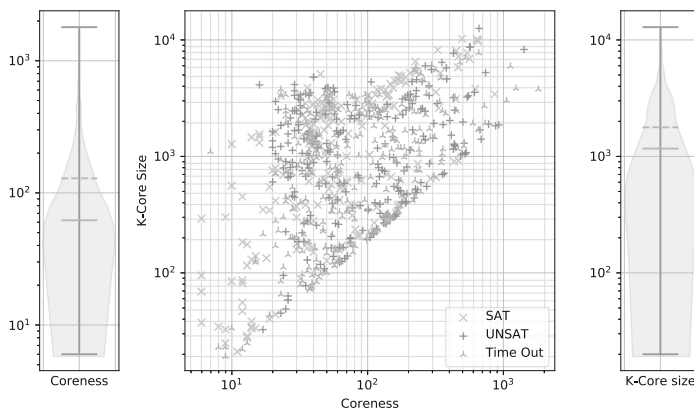


Fig. 9. Characteristics of K-Cores computed with vivification on the remaining learnt clauses after 110,000 conflicts, for all the problems from competitions 2013 to 2017.

experiment clearly demonstrates that the presence of K-Cores is a strong and general characteristics of DG, for SAT and UNSAT problems (do notice also that, on many problems, we also observed a K-Core size of the coreness value, showing a number of problems with large cliques in their DG).

7 Conclusion

Our experimental report points out a hidden structure behind the learning mechanism of CDCL SAT solvers. We demonstrated that all the proof graphs generated by SAT solvers share a common important characteristics: dependency graphs are non-degenerated with high values. This shows that there is a very dense subgraph that seem to play an important role in the search. In order to emphasize this role, we demonstrated that, by analyzing the dependency graph at half of the run, we were already capable of identifying a set of learnt clauses that will be necessary for deriving the final contradiction, with high confidence. The analysis of the K-Core also allowed us to identify a very small set of variables that will occur in the very last learnt clauses. We also demonstrated that most of the learnt clauses are very close to the K-Core in terms of resolution steps. We demonstrated that, even in parallel, a K-Core composed by clauses of all the threads is also dominating the proof. Additionally, parallel proofs are generally larger, even if they are shallower.

At last, we shown that the analysis of the dependency graph can be done in any CDCL solver at minimal cost. It is sufficient to analyze the current set of learnt clauses to build a dependency graph that also exhibit a strong K-Core. This last experiment also demonstrates that the existence of K-Cores also holds for SAT and UNSAT problems. We hope that our work will cast new research directions on the reasons why SAT solvers are so efficient. We hypothesis that one of the reasons for their efficiency is their ability to produce and handle proofs with these properties. In some sense, this proves that CDCL proofs are really far away from tree-like resolutions (non-degenerate graphs are clearly far from trees). We also think that our analysis will allow to efficiently reduce the proof sizes generated by SAT solvers. A short proof for UNSAT can be essential in many applications.

References

1. Ansótegui, C., Bonnet, M.L., Levy, J., Many, F.: Measuring the hardness of sat instances. In: Proceedings of AAAI, pp. 222–228 (2008)
2. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: IJCAI (2009)
3. Darwiche, A., Pipatsrisawat, K.: Complete Algorithms, Chap. 3, pp. 99–130. IOS Press (2009)
4. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. JACM **5**, 394–397 (1962)

5. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005). https://doi.org/10.1007/11499107_5
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
7. Huang, J.: The effect of restarts on the efficiency of clause learning. In: IJCAI 2007, pp. 2318–2323 (2007)
8. Järvisalo, M., Matsliah, A., Nordström, J., Živný, S.: Relating proof complexity measures and practical hardness of SAT. In: Milano, M. (ed.) CP 2012. LNCS, pp. 316–331. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33558-7_25
9. Katsirelos, G., Sabharwal, A., Samulowitz, H., Simon, L.: Resolution and parallelizability: barriers to the efficient parallelization of sat solvers. In: Proceedings of AAAI (2013)
10. Li, C.M., Anbulagan, A.: Heuristics based on unit propagation for satisfiability problems. In: Proceedings of IJCAI, pp. 366–371 (1997)
11. Li, C.M., Gérard, S.: On the limit of branching rules for hard random unsatisfiable 3-sat. In: Proceedings of ECAI, pp. 98–102 (2000)
12. Luo, M., Li, C.M., Xiao, F., Manyá, F., Lu, Z.: An effective learnt clause minimization approach for CDCL SAT solvers. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17), pp. 703–711 (2017)
13. Marques-Silva, J.P., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. *IEEE Trans. Comput.* **48**(5), 506–521 (1999)
14. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of DAC, pp. 530–535 (2001)
15. Piette, C., Hamadi, Y., Sas, L.: Vivifying propositional clausal formulae. In: ECAI, pp. 525–529 (2008)
16. Seidman, S.B.: Network structure and minimum degree. *Soc. Netw.* **5**(3), 269–287 (1983)
17. Shin, K., Eliassi-Rad, T., Faloutsos, C.: Graph mining using k-core analysis - patterns, anomalies and algorithms. In: IEEE 16th International Conference on Data Mining ICDM, pp. 469–478 (2016)
18. Simon, L.: Post mortem analysis of sat solver proofs. In: Berre, D.L. (ed.) POS-14. Fifth Pragmatics of SAT Workshop. EPiC Series in Computing, vol. 27, pp. 26–40. EasyChair (2014). <https://doi.org/10.29007/gpp8>, <https://easychair.org/publications/paper/N3GD>
19. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_31



Sequential Precede Chain for Value Symmetry Elimination

Graeme Gange^{1(✉)} and Peter J. Stuckey^{1,2}

¹ Faculty of Information Technology, Monash University, Melbourne, Australia
graeme.gange@monash.edu

² Data61, CSIRO, Melbourne, Australia

Abstract. The main global constraint used for removing value symmetries is the VALUE-PRECEDE-CHAIN constraint which forces the first occurrences of values in an ordered list to be appear in order. We introduce the SEQ-PRECEDE-CHAIN constraint for the restricted, but common, case where the values are $1, 2, \dots, k$, and variables in the constraint do not take values higher than k . We construct an efficient domain consistent propagator for this constraint, and show how we can generate explanations for its propagation. This leads us to an efficient domain consistent decomposition. We show how we can map any VALUE-PRECEDE-CHAIN to use instead SEQ-PRECEDE-CHAIN. Experiments show that the new propagator and decomposition are better than existing approaches to propagating VALUE-PRECEDE-CHAIN.

1 Introduction

The VALUE-PRECEDE-CHAIN constraint introduced by Law and Lee [1] is one of the principal symmetry breaking constraints for removing value symmetries. The constraint $\text{VALUE-PRECEDE-CHAIN}([t_1, t_2, \dots, t_k], X)$ holds if in the sequence of values taken by variables X , at least one occurrence of t_i occurs earlier in the sequence than any occurrence of t_j for all $1 \leq i < j \leq k$.

Law and Lee also introduce the constraint $\text{VALUE-PRECEDE}(t_1, t_2, X)$ which requires that an occurrence of t_1 occurs before any occurrence of t_2 in X . They describe a domain consistent propagator for VALUE-PRECEDE and implement VALUE-PRECEDE-CHAIN by decomposition into a series of VALUE-PRECEDE:

$$\text{VALUE-PRECEDE-CHAIN}([t_1, t_2, \dots, t_k], X) = \bigwedge_{i=2}^k \text{VALUE-PRECEDE}(t_{i-1}, t_i, X)$$

Law and Lee [1] observe that adding redundant VALUE-PRECEDE constraints obtains stronger propagation. However, even with redundant constraints, this encoding is not domain consistent.

Example 1. Consider the constraint $\text{VALUE-PRECEDE-CHAIN}([1, \dots, 5], X)$, where $X = [x_1, \dots, x_9]$, with domains $\{\{0, 1\}, \{0, 1, 5\}, \{0, 3\}, \{0, 1, 2, 4\}, \{0, 1, 3\}, \{1, 3\}, \{2, 3, 4, 5\}, \{4, 5\}, \{0, 1, 2, 3\}\}$. The domains of the variables are

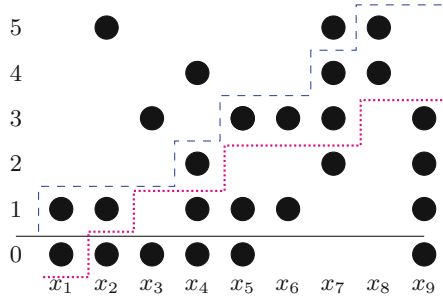


Fig. 1. A sequence of variables $[x_1, \dots, x_9]$, with corresponding upper and lower frontiers. (Color figure online)

illustrated in Fig. 1. We can clearly reduce the upper bound of x_2 and x_3 to 1 since there is no earlier occurrence of 2. Similarly we can reduce the upper bound of x_4 to 2 since there is no earlier occurrence of 2. Similarly we can reduce the upper bound of x_7 to 4 since there is no earlier occurrence of 4. A decomposition into VALUE-PRECEDE constraints cannot obtain any further propagation: there are at least two possible occurrences each of 2 and 3 to the left of x_8 , so neither can be assigned. And because all other variables may be lower, nothing else can propagate.

But we can also increase the lower bound of x_4 to 2 (fixing it to 2) since there is no other value 2 capable of supporting $x_8 = 4$. We know that at least one of x_5, x_6 and x_7 must take the value 3 to support this value, and hence x_4 cannot possibly take the value 1. \square

It is observed in [2] that a domain consistent encoding may be obtained by reformulating VALUE-PRECEDE-CHAIN as a REGULAR [3] constraint, for which appropriate propagators [4–6] and decompositions [7,8] exist. However, this reformulation is rather inefficient: the automaton, shown in Fig. 2 has $k \times |\mathcal{D}(X)|$ edges, which are unfolded $|X|$ times in constructing the REGULAR constraint.

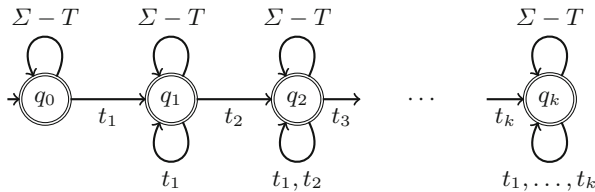


Fig. 2. Regular automata for VALUE-PRECEDE-CHAIN

In this paper we investigate a special case of VALUE-PRECEDE-CHAIN and show how it can be propagated efficiently, or implemented by a decomposition.

We then show how it can be used to encode any VALUE-PRECEDE-CHAIN constraint. The contributions of this paper are:

- an efficient domain consistent propagator for SEQ-PRECEDE-CHAIN which explains its propagations;
- an efficient decomposition of SEQ-PRECEDE-CHAIN;
- a proof that the decomposition maintains domain consistency;
- an encoding of VALUE-PRECEDE-CHAIN using SEQ-PRECEDE-CHAIN which is domain consistent; and
- experiments showing that the propagator and decomposition are both highly competitive with existing approaches to VALUE-PRECEDE-CHAIN.

2 The Common Case: value-precede-chain ([1, 2, . . . , k], X)

The most frequent use of VALUE-PRECEDE-CHAIN is in cases where either all values in the domain are indistinguishable (i.e. colouring problems), or have exactly one distinguished value (i.e. concert hall scheduling, optional bin packing). We introduce a new global SEQ-PRECEDE-CHAIN(X) equivalent to VALUE-PRECEDE-CHAIN([1, 2, . . . , k], X) where k is the maximum value appearing in the domain on any $x \in X$.

Given this new simplified form of the constraint we can build a global domain consistent propagator that is more efficient. Let \mathcal{D} be the current domain of the variables, thought of as a set of unary constraints. Let \mathcal{D}_x be the restriction of \mathcal{D} to constraints only mentioning variables x . We use notation $\mathcal{D}(x)$ to refer to the current domain of x , that is the set of all values it can take $\mathcal{D}(x) = \{v \mid x = v \rightarrow \mathcal{D}_x\}$, and lb and ub to refer to the least and greatest value x can take in the current domain, $lb(x) = \min \mathcal{D}(x)$ and $ub(x) = \max \mathcal{D}(x)$.

2.1 Propagation

Propagation proceeds in two passes: a forward pass to find the upper frontier up , tightening upper bounds as it proceeds. Then it walks backwards to collect the lower frontier low , setting the value of any variable where the frontiers coincide.

The algorithm shown in Fig. 3 is fairly straightforward. In a forward pass the algorithm maintains up the largest value seen in a chain of the form $1, \dots, up$. If the upper bound of a variable is more than one greater than up we reduce it to $up + 1$. If the new upper bound is $up + 1$ we increment up and record in $first[up]$ the first position where the new bound is met. We similarly keep track of the highest lower bound seen so far low , and store in $last[low]$ the last position where the increasing sequence can take a value below low . Note that the sequence $[ub(x_1), \dots, ub(x_n)]$ (after propagation) records the lexicographically greatest solution to the constraint.

The backwards pass pulls up low bounds. If low is first possible at position i then i must take that value, so we propagate that. If low is in the domain of $x[i]$ and this is before $last[low]$ we then reduce the lower bound. We also store

```

seq-precede-chain( $x$ )
   $up := 0$ 
   $low := 0$ 
   $last[0] := -\infty \triangleright last[i] = \infty$  for  $i > 0$ 
  for ( $i \in 1..n$ )  $\triangleright$  impose upper bounds
    if ( $ub(x[i]) > up + 1$ )
      propagate  $x[i] \leq up + 1$ 
    if ( $ub(x[i]) = up + 1$ )
       $up := up + 1$ 
       $first[up] := i$ 
    if ( $low < lb(x_i)$ )
       $last[lb(x_i)] := i$ 
       $low := lb(x_i)$ 
  for ( $i \in n..1$ )  $\triangleright$  impose lower bounds
     $least[i] := lb(x_i)$ 
    if ( $first[low] = i$ )
      propagate  $x[i] \geq low$ 
    if ( $i \leq last[low] \wedge low \in dom(x[i])$ )
       $least[i] := low$ 
       $last[low] := i$ 
       $low := low - 1$ 
    
```

Fig. 3. Propagation algorithm for SEQ-PRECEDE-CHAIN.

the new *last* position information. The code to create *least*[*i*] generates the lexicographically least solution to the constraint, [*least*[1], ..., *least*[*n*]]. This is not required for propagation, but useful for proofs.

Example 2. Consider the problem of Example 1 using instead the constraint SEQ-PRECEDE-CHAIN(X). The propagator sets $first[1] = 1$ propagates $x_2 \leq 1$, $x_3 \leq 1$ and $x_4 \leq 2$, sets $first[2] = 4$, $first[3] = 5$ and propagates $x_7 \leq 4$ and sets $first[4] = 7$ and $last[1] = 7$, then sets $first[5] = 8$, and $last[4] = 8$. The blue line in Fig. 1 shows the progress of *up* (drawn above its value). In the backwards pass *low* takes values 4, 4, 3, 3, 3 setting $last[3] = 7$. Then we detect that $first[2] = 4$ and propagate $x_4 \geq 2$, and set $last[2] = 4$, then *low* takes values 2, 2, 1, 0, setting $last[1] = 2$ and $last[0] = 1$. The red line in Fig. 1 shows the progress of *low* (drawn underneath its value). The sequence *least* is calculated as [0, 1, 0, 2, 0, 1, 3, 4, 0]. \square

Theorem 1. *The propagator for SEQ-PRECEDE-CHAIN(X) is correct.*

Proof. Given a solution $\theta = [v_1, \dots, v_n]$ of SEQ-PRECEDE-CHAIN(X) we show that the propagator never removes it. Suppose the upper bound pruning removes earliest value v_i , then *up* at iteration $i - 1$ has value less than $v_i - 1$, but then there can be not $j < i$ with value $v_j = v_i - 1$ earlier in θ , otherwise *up* would have reached this level. Contradiction.

Suppose the lower bound pruning removes latest value v_i , then $v_i < u$ where $first[u] = i$, and *low* at iteration $i + 1$ has value $u + 1$. Since $first[u] = i$ then

clearly $v_j < u, 1 \leq j < i$. Let k be the first position after i where $v_k = low$ in the k^{th} iteration. Then the sequence of low in between only steps down to $u + 1$ hence there is no position in θ taking value u before a value of $u + 1$. Contradiction. \square

Theorem 2. *The propagator for SEQ-PRECEDE-CHAIN(X) enforces domain consistency.*

Proof. Given a value $v_i \in dom(x_i)$ after the propagation of SEQ-PRECEDE-CHAIN(X) we show that there exists a solution $[v_1, \dots, v_n]$ where $v_j \in \mathcal{D}(x_j), 1 \leq j \leq n$ supporting this value. If $v_i = ub(x_i)$ then we can take the sequence of $[ub(x_1), \dots, ub(x_n)]$. Suppose $first[ub(x_i)] \neq i$ then an alternate solution is clearly $[ub(x_1), \dots, ub(x_{i-1}), v_i, ub(x_{i+1}), \dots, ub(x_n)]$.

It remains to consider where $first[ub(x_i)] = i$ and $v_i < ub(x_i)$. We show that the sequence $\theta = [ub(x_1), \dots, ub(x_{i-1}), v_i, least[i + 1], \dots, least[n]]$ is a solution of the constraint. First note that $[least[1], \dots, least[n]]$ is a solution of the constraint, since we only step down by at most one, and $least[j] \in dom(x_j), 1 \leq j \leq n$. Now $least[i] \neq ub(x_i)$ otherwise we would have propagated that $x_i \geq last[i]$ and then $v_i \notin dom(x_i)$. The sequence θ is a solution since $least[i] \leq v_i < ub(x_i)$ is neither too large for the predecessor sequence nor too large for the successor sequence by definition. \square

2.2 Incrementality

Unfortunately, successive runs of the propagator are frequently quite wasteful. The propagator will perform a full forward/backward pass, even if neither frontier has changed. We can do better by identifying the circumstances where a frontier may change: (1) the upper-bound of a variable on the upper frontier is decreased, (2) the lower-bound of a variable is increased *above* the lower frontier, or (3) k is removed from the domain of $last[k]$. These changes are also *directional*: if the upper-bound of x_i changes, only variables right of x_i may be affected. Similarly, if the lower frontier is shifted, the change can only cascade to the left. Thus, if we start repairing the frontiers as the result of a change, we can stop whenever the repaired frontier coincides with the existing one.

However, *first* and *last* are not quite enough for an incremental propagator: when $\mathcal{D}(x_i)$ changes, we only want to execute the propagator if one of the frontiers has been affected. However while *first* and *last* tell us for a given value which variable supports it, we need to know for a given variable whether it supports a frontier step.

We thus maintain two persistent arrays, *first_val* and *last_val*, maintaining the invariant $first[k] \leq n \rightarrow first_val[first[k]] = k$ (similarly for *last* and *last_val*). That is, $first_val[x_i] = k$ whenever x_i supports the k^{th} step of the upper frontier (but is unconstrained if x_i is not a support). This allows us to quickly determine if changes to $\mathcal{D}(x_i)$ have invalidated the frontier (by testing if $last_val[x_i] \notin \mathcal{D}(x_i) \wedge last[last_val[x_i]] = x_i$), while only updating cells along the frontier. The incremental propagator is shown in Fig. 4 where *wakeup* looks at domain

```

wakeup(changes)
  for( $x_i \in changes$ )
     $k := first\_val[x_i]$ 
    if( $first[k] = x_i \wedge ub(x_i) < k$ )
      if(repair_upper( $k$ ) = FAIL)
        return FAIL
  for( $x_i \in changes$ )
     $k := last\_val[x_i]$ 
    if( $last[k] = x_i \wedge k \notin \mathcal{D}(x_i)$ )
      repair_lower( $k$ )
  return OKAY

repair_lower( $k$ )
   $i := last[k]$ 
  while( $k > 0$ )
    if( $k \in \mathcal{D}(x_i)$ )
       $last[k] := i$ 
       $last\_val[i] := k$ 
      if( $first[k] = i$ )
        propagate  $\langle x_i \geq k \rangle$ 
       $k := k - 1$ 
      if( $last[k] < i$ )
        return
   $i := i - 1$ 

repair_upper( $k$ )
   $i := first[k]$ 
   $lim := last[k]$ 
  while( $i < lim$ )
    if( $ub(x[i]) > k$ )
      propagate  $\langle x[i] \leq k \rangle$ 
    if( $k \in \mathcal{D}(x[i])$ )
       $first[k] := i$ 
       $first\_val[i] := k$ 
       $k := k + 1$ 
      if( $i < first[k]$ )
        return OKAY
   $i := i + 1$ 
  if( $i < N$ )
     $\triangleright$  First occurrence of  $k$  is too late.
  return FAIL
    
```

Fig. 4. Incremental propagation algorithms for SEQ-PRECEDE-CHAIN

changes since last execution and `repair_upper` and `repair_lower` exactly reflect the corresponding stage of the basic seq-precede-chain propagator.

2.3 Explanation

For integration into lazy clause generation [9], every propagator must be able to *explain* itself: for each inference a it make, the propagator p must be able to produce a set of antecedents E such that $\mathcal{D} \Rightarrow E$, and $c \wedge E \Rightarrow a$.

The SEQ-PRECEDE-CHAIN propagator performs two kinds of propagation: on the forward pass, it tightens upper bounds to the reachable frontier. On the backward pass, it fixes values which we discover *must* be on the frontier.

Explaining $x_i \leq k$ is straightforward: if x_i is greater than j , some variable to the left of x_i must take the value k . Thus $\bigwedge_{j=1}^{i-1} x_j \neq k \rightarrow x_i \leq k$. However, if there are no occurrences of k , we know there can also be no occurrences of values $k' > k$. Thus, we can use the more general explanation $\bigwedge_{j=1}^{i-1} x_j < k \rightarrow x_i \leq k$.

Example 3. Recall the upper-bound propagation from Example 1. The explanation for $x_4 \leq 2$ is illustrated in Fig. 5(a), and is simply: $\langle x_1 \leq 1 \rangle \wedge \langle x_2 \leq 1 \rangle \wedge \langle x_3 \leq 1 \rangle \rightarrow \langle x_4 \leq 2 \rangle$. \square

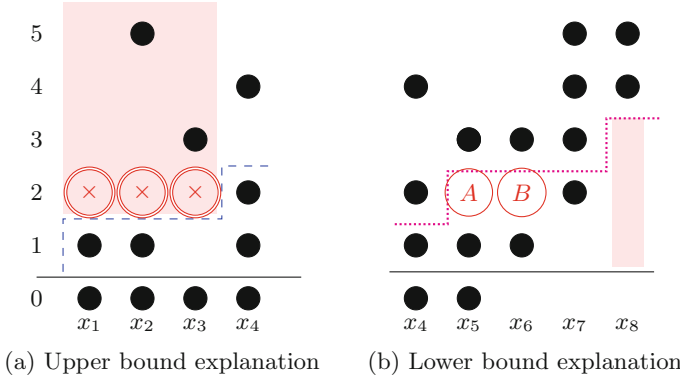


Fig. 5. Example explanations for upper- and lower-bound pruning.

The reasoning for lower-bound propagation is slightly more intricate. Lower bounds are updated when a lower bound pulls the frontier upwards, forcing the upper and lower frontiers to coincide at a variable. The explanation then consists of 3 parts: why the frontier cannot be relaxed to the left, why the lower bound was lifted, and why there is no slack in the frontier.

Example 4. Now consider the lower-bound propagation from Example 1. We must explain why the lower and upper frontiers coincide at $x_4 = 2$. The reason for the upper frontier is given in Example 3.

For the lower frontier, we have to explain why the lower frontier could not be lower. Hence we need to prevent any step downs which did not occur from the tight lower bound, until the propagation. For the explanation of the example we need the tight bound $\langle x_8 \geq 4 \rangle$ and the prevention of step downs $\langle x_6 \neq 2 \rangle$ and $\langle x_5 \neq 2 \rangle$. Note that position x_7 where there is a step down is not relevant. The full explanation is $\langle x_1 \leq 1 \rangle \wedge \langle x_2 \leq 1 \rangle \wedge \langle x_3 \leq 1 \rangle \wedge \langle x_5 \neq 2 \rangle \wedge \langle x_6 \neq 2 \rangle \wedge \langle x_8 \geq 4 \rangle \rightarrow \langle x_4 \geq 2 \rangle$. \square

In general the procedure can be formalized using the equations below.

$$\begin{aligned}
 \text{explain}(\langle x_i \leq k \rangle) &= \bigwedge_{j=0}^{i-1} \langle x_j < k \rangle \\
 \text{explain}(\langle x_i \geq k \rangle) &= \text{ex}_L(x_{i+1}, k) \wedge \bigwedge_{j=0}^{i-1} \langle x_j < k \rangle
 \end{aligned}$$

$$\text{where } \text{ex}_L(x_i, k) = \begin{cases} \langle x_i \geq k \rangle & \text{if } \text{lb}(x_i) \geq k \\ \text{ex}_L(x_{i+1}, k + 1) & \text{if } k \in \mathcal{D}(x_i) \\ \langle x_i \neq k \rangle \wedge \text{ex}_L(x_{i+1}, k) & \text{otherwise} \end{cases}$$

3 Domain Consistent Decomposition for seq-precede-chain

Given the nature of the propagation for the SEQ-PRECEDE-CHAIN(X) constraint we devised a simple decomposition, introducing intermediate variables H_0, \dots, H_n :

$$\text{SEQ-PRECEDE-CHAIN}([x_1, \dots, x_n]) = \exists H_0, H_1, \dots, H_n. \begin{matrix} H_0 = 0 \\ \bigwedge_{i=1}^n H_i \leq H_{i-1} + 1 \wedge \\ H_i = \max(x_i, H_{i-1}) \end{matrix}$$

The intuition is that H_i stores the *highwater mark* u of the increasing subsequence $1, \dots, u$ seen in the x values from positions 1 to i . A MiniZinc definition (slightly different to avoid using H_0) is given below.

```

1 predicate seq_precede_chain(array[int] of var int: X) =
2   let { int: l = lb_array(X); % least possible value
3       int: u = ub_array(X); % greatest possible value
4       int: f = min(index_set(X));
5       array[index_set(X)] of var 1..u: H; } in
6   H[f] <= 1 /\ H[f] = max(X[f], 0) /\
7   forall(i in index_set(X) diff {f})
8     (H[i] <= H[i-1] + 1 /\ H[i] = max(X[i], H[i-1]));
    
```

Example 5. Consider the problem of Example 1. Passing forward across the constraints sets $H_0 = 0$, $\mathcal{D}(H_1) = \mathcal{D}(H_2) = \mathcal{D}(H_3) = \{0, 1\}$, propagating $x_2 \leq 1$ and $x_3 \leq 1$, $\mathcal{D}(H_4) = \{0, 1, 2\}$, propagating $x_4 \leq 2$, $\mathcal{D}(H_5) = \mathcal{D}(H_6) = \{0, 1, 2, 3\}$, $\mathcal{D}(H_7) = \{0, 1, 2, 3, 4\}$ propagating $x_7 \leq 4$ and $\text{dom}(H_8) = \{4, 5\}$. Now passing backwards over the constraints sets $\mathcal{D}(H_7) = \mathcal{D}(H_6) = \{3, 4\}$, then $\mathcal{D}(H_5) = \{3\}$ and $\mathcal{D}(H_4) = \{2\}$ which propagates $x_4 \geq 2$ and finally $\mathcal{D}H_3 = \{1\}$. The decomposition mimics the global where $\text{dom}(H_i) = \{\text{low}..\text{up}\}$ for the values reached after the i^{th} iteration. \square

We now show that this decomposition is correct and maintains domain consistency if the max propagator and \leq propagators are domain consistent, and we do not branch to introduce holes in domains of H variables.

Theorem 3. *The decomposition for SEQ-PRECEDE-CHAIN(X) is correct.*

Proof. Given a solution $\theta = [v_1, \dots, v_n]$ of SEQ-PRECEDE-CHAIN(X). We show that the decomposition never removes it. Now ignoring the constraints $H_i \leq H_{i-1} + 1$ we can inductively show that the max constraints enforce $H_i = \max_{j \in 1..i}(v_j)$. The max constraints are satisfied. Suppose the inequalities were violated then the maximum value before $i - 1$ is $H_{i-1} < v_i - 1$ and hence θ is not a solution. \square

Lemma 1. *The decomposition maintains $\mathcal{D}(H_i)$ as range domains, assuming these variables are not branched on to introduce holes.*

Proof. The proof is by induction. Assuming all domains of H_i are range domains, we show that no propagation can cause a hole to be created. First $H_i \leq H_{i-1} + 1$ can never create holes. Consider $H_i = \max(x_i, H_{i-1})$ this may create a hole in the domain of H_i if there is hole in $v \notin \mathcal{D}(x_i)$. Now either $v < lb(H_{i-1})$ in which case $lb(H_i) = lb(H_{i-1})$ and there is no hole, or $v \geq lb(H_{i-1})$ in which case either $v \leq ub(H_{i-1})$ and then no hole is punched since $v \in \mathcal{D}(H_{i-1})$. The remaining case is $v > ub(H_{i-1})$ then $ub(x_i) > v$ and $ub(H_i) = ub(x_i)$ but this would violate the constraint $H_i \leq H_{i-1} + 1$ so it would reduce the upper bounds to $ub(H_{i-1})$ and no hole is created. \square

Lemma 2. *The constraints $H_i \leq H_{i-1} \wedge H_i = \max(x_i, H_{i-1})$ maintain domain consistency of the conjunction assuming $\mathcal{D}(H_i)$ and $\mathcal{D}(H_{i-1})$ are range domains.*

Proof. We first show that an arbitrary $v \in \mathcal{D}(x_i)$ can be extended to a solution. If $v = ub(H_i)$ then we have solution (H_{i-1}, x_i, H_i) of $(v - 1, v, v)$, and $v - 1 = ub(H_{i-1})$ by propagation of the inequality. If $\exists v' \in \mathcal{D}(H_i) \cap \mathcal{D}(H_{i-1}), v' \geq v$ we have solution (v', v, v') . Otherwise we have $\forall v' \in \mathcal{D}(H_i) \cap \mathcal{D}(H_{i-1}), v' < v$. Since they are range domains this means that $ub(H_{i-1}) < v$ and thus $ub(H_i) \leq v$, which implies $v = ub(H_i)$.

Next we show that an arbitrary value $v \in \mathcal{D}(H_{i-1})$ can be extended to a solution. If $v' = lb(x_i) \leq v$ then (v, v', v) is a solution since the max constraint ensures $v \in \mathcal{D}(H_i)$. Otherwise if $v' = v + 1$ then (v, v', v') is a solution since the max constraint ensures $v' \in \mathcal{D}(H_i)$. Otherwise $v' > v + 1$, but then $lb(H_i) = v'$ and the inequality would have removed v from the $\mathcal{D}(H_{i-1})$.

Finally we show that an arbitrary value $v \in \mathcal{D}(H_i)$ can be extended to a solution. Now $v' = lb(x_i) \leq v$ hence either $v' < v$ in which case (v, v', v) is a solution since the max constraint ensures $v \in \mathcal{D}(H_{i-1})$. Or $v' = v$ and either $v \in \mathcal{D}(H_{i-1})$ leading to solution (v, v, v) or $v > ub(H_{i-1})$ meaning $v = ub(H_i)$ and hence $(v - 1, v', v)$ is a solution. \square

Theorem 4. *The decomposition for SEQ-PRECEDE-CHAIN(X) enforces domain consistency, assuming no hole punching branching on H variables.*

Proof. Lemma 1 shows that the decomposition will always have range domains for H_i if holes are not punched by branching. Lemma 2 shows that the constraint $c(H_{i-1}, x_i, H_i) = H_i \leq H_{i-1} + 1 \wedge H_i = \max(x_i, H_{i-1})$ is domain consistent assuming range domains for H . Since the decomposition treating $c(H_{i-1}, x_i, H_i)$ as a single constraint is Berge acyclic, the decomposition enforces domain consistency. \square

Note that the decomposition $c(H_{i-1}, x_i, H_i)$ does *not* maintain domain consistency if we permit holes in the H domains.

Example 6. Consider $c(H_{i-1}, x_i, H_i)$, with variable domains $\mathcal{D}(H_{i-1}) = \{1, 4\}$, $\mathcal{D}(x_i) = [1, 5]$, $\mathcal{D}(H_i) = \{1, 3, 5\}$.

The value $H_i = 3$ is infeasible, as neither 2 nor 3 are in the domain of H_{i-1} . However $H_i \leq H_{i-1} + 1$ is satisfied by assignment $(4, -, 3)$, and $H_i = \max(x_i, H_{i-1})$ is satisfied by $(1, 3, 3)$.

The value $x = 3$ is also infeasible, as $x > 2$ would require H_{i-1} to take value 4, which forces H_i to be 5. But $H_{i-1} = 4 \wedge H_i = 5$ forces $x_i = 5$, which contradicts our x_i assignment. However $H_i \leq H_{i-1} + 1$ is independent of x , and $H_i = \max(x_i, H_{i-1})$ is satisfied by $(1, 3, 3)$. \square

The advantage of the decomposition over the global is that it is incremental by its nature. For a learning solver there is another advantage, explanations for propagation and failure can use the H variables, this allows the summary of lots of previous behaviour in a way that is reusable. For example given the lower bound propagation in Example 1 is explained by

$$\langle H_3 \leq 1 \rangle \wedge \langle H_4 \geq 2 \rangle \rightarrow \langle x_4 \geq 2 \rangle.$$

There could be many reasons why $H_4 \geq 2$ rather than the particular history shown in Fig. 5(b).

4 Mapping value-precede-chain to seq-precede-chain

While the case for SEQ-PRECEDE-CHAIN may seem somewhat restricted we can use it to model arbitrary VALUE-PRECEDE-CHAIN constraints by using a mapping to the case that SEQ-PRECEDE-CHAIN supports.

A MiniZinc decomposition for the encoding is given below. Because the mapping using ELEMENT is domain consistent (including the view used to encode $X[i] - l + 1$), the resulting decomposition is also domain consistent, since the constraint structure is a tree [10].

```

1 predicate value_precede_chain(array[int] of int: T, array[int] of var int: X)
2 = if min(index_set(T)) = 1 /\ forall(i in index_set(T))(T[i] = i)
3   /\ max(T) = ub_array(X) then seq_precede_chain(X)
4   else
5     let { int: l = lb_array(X);
6         int: u = ub_array(X);
7         array[1..u-1+1] of int: p
8           = [ sum(i in index_set(T) where T[i] = j)(i)
9             | j in 1..u ];
10        array[index_set(X)] of var 0..length(T): Y;
11    } in
12      forall(i in index_set(X))
13        (element(X[i] - l + 1, p, Y[i])) /\
14        seq_precede_chain(Y)
15    endif;

```

Example 7. Consider the constraint VALUE-PRECEDE-CHAIN($[2, -2, 1, -1], X$) where X variables range over values $-3..3$, then we use element constraints of the form ELEMENT($x_i + 4, [0, 2, 4, 0, 3, 1, 0], y_i$) to map X to Y , together with SEQ-PRECEDE-CHAIN(Y) to encode this constraint. \square

5 Experimental Results

We implemented the new propagator and decomposition in `CHUFFED` [11], a lazy clause generation CP solver. For comparison, we also tested several alternate encodings of `VALUE-PRECEDE-CHAIN`:

- dec** Decomposition into $k - 1$ `VALUE-PRECEDE` constraints, enforced with the propagator described in [1].
- ddec** As `dec`, but the `VALUE-PRECEDE` constraints encoded using the standard `MiniZinc` decomposition.¹
- reg** As a finite-state automaton of Fig. 2, enforced using an incremental MDD-based propagator [6].
- chain** The new propagator given in Sect. 2.
- seq** The new decomposition given in Sect. 3.

The modified version of `chuffed`, `MiniZinc` models and data files are available at <https://github.com/gkgange/chuffed/releases/tag/data-2018-cp>.

5.1 Concert Hall

The *concert hall scheduling problem* [12] considers a set H of identical halls, and a set C of concerts each having a start time s_c , end time e_c and profit p_c . Each concert may either be allocated to a hall $h \in H$, or not scheduled. As the halls are interchangeable, we break symmetries by imposing a `SEQ-PRECEDE-CHAIN` constraint over the set of assignments. We also add dominance-breaking constraints, to prefer shorter, more profitable concerts. Branch first on the set of concerts to include (including concerts ordered by $\frac{p_c}{e_c - s_c + 1}$), then assign these concerts in input order to the first available hall.

Figure 6 reports performance of the five `SEQ-PRECEDE-CHAIN` implementations on concert hall scheduling problems, with and without lazy clause generation. It is clear that `dec`, decomposition into `VALUE-PRECEDE` propagators, is at all not competitive. Without learning, `chain` and `seq` are clearly superior. With learning (where we omit the uncompetitive `dec`, those results being far outside the current chart bounds), the gap closes dramatically, with `ddec` nearly catching the new methods, and `reg` slightly behind.

A direct comparison of `seq` and `chain` is given in Fig. 7. Interestingly, although `chuffed`'s propagator for $Z = \text{MAX}(X, Y)$ is only `bounds(Z)` consistent, with learning disabled the propagator and decomposition always explored the same number of nodes. The incremental propagator was up to 30% faster than the decomposition, but typically closer to 10%.

Enabling learning introduces a good deal of variance, as the changes in nogoods can result in dramatic changes in search. Nevertheless, we see similar results: the incremental propagator is typically slightly faster than the decomposition, both of which consistently outperform existing encodings.

¹ That is, $\exists b_1, \dots, b_n. ((b_1 \leftrightarrow x_1 = s) \wedge \bigwedge_{i=2}^n b_i \leftrightarrow (b_{i-1} \vee \langle x_{i-1} = s \rangle) \wedge \langle x_i = t \rangle \rightarrow b_{i-1})$. Here b_i records whether there is an occurrence of s no later than x_i .

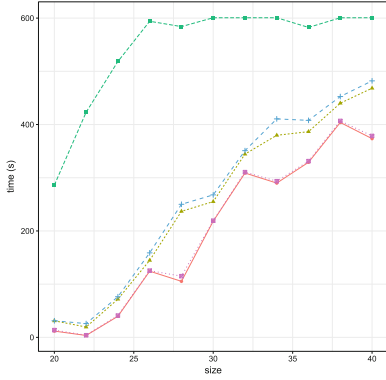
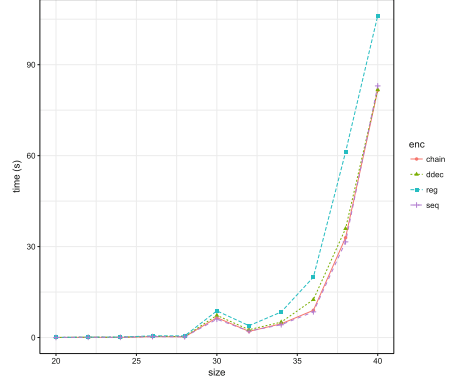
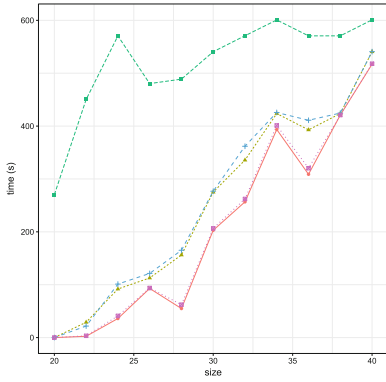
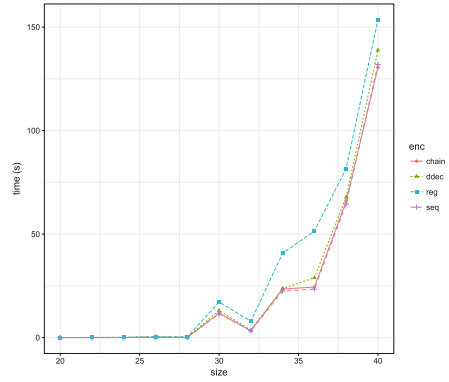
 $|H| = 10$, without learning $|H| = 10$, with learning $|H| = 14$, without learning $|H| = 14$, with learning

Fig. 6. Comparison of SEQ-PRECEDE-CHAIN implementations, for different sizes of H . We omit DEC from plots for learning, being totally non-competitive.

5.2 Capacitated Concert Hall

To evaluate these techniques on domains with multiple sets of indistinguishable values, we extend the concert hall problem with *capacities*: each offer now has a minimum size requirement, and each hall has a maximum capacity (and the ‘not scheduled’ hall can fit any concert). We then add the constraint:

```
1 constraint forall (o in Offers) (size[o] <= capacity[assign[o]] );
```

While the halls are no longer indistinguishable, they may still be partitioned into equivalence classes based on the set of concerts which they may hold. We break this symmetry by adding a VALUE-PRECEDE-CHAIN constraint for each equivalence class of halls. When using `seq` and `chain` these are mapped to a SEQ-PRECEDE-CHAIN constraints using the decomposition of Sect. 4.

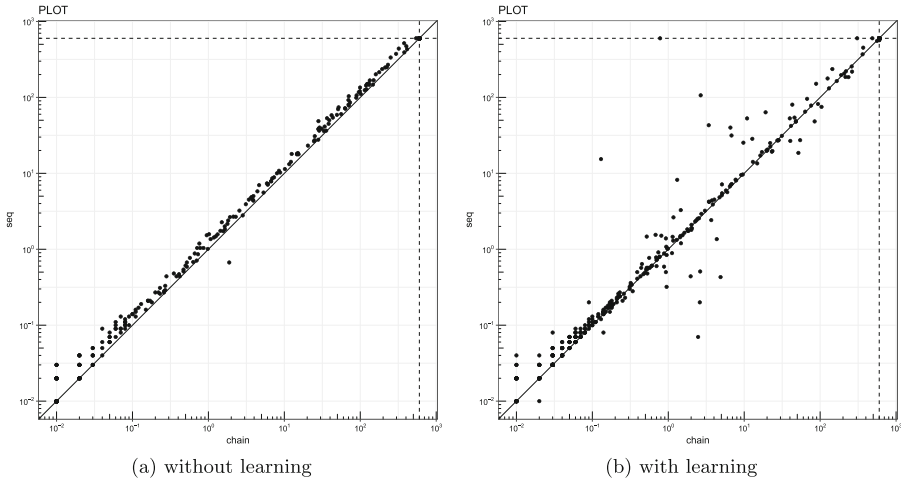


Fig. 7. Comparison of incremental SEQ-PRECEDE-CHAIN propagator versus the sequential decomposition, on concert hall scheduling instances (a) without and (b) with learning.

We extended each instance used in the previous section with a capacity and size for each hall and offers. We generate capacities in a similar manner as the other parameters described in [1]: we generate each equivalence-class of halls by first selecting the class size uniformly in $[1, r]$, then choose the capacity uniformly in the range $[200, 1000]$. This is repeated until the m hall capacities have been generated. We generate concert sizes using the same procedure.² Search follows the same strategy as for the previous problem: include concerts by profitability, then assign to the first hall in input order.

Figure 8 illustrates the performance of each encoding on the capacitated concert hall instances. On these problems, the differences between encodings are more pronounced – interestingly, here DDEC performs worse than DEC, which remains uncompetitive; CHAIN and SEQ again outperform REG. On these problems, we did observe differences in backtrack counts, but SEQ nevertheless appears to be the more robust approach (except on instances where both methods complete in $< 0.1s$, where initialisation time dominates).

² Note that the size equivalences are generated independently of other instance parameters, so may break existing dominance relationships.

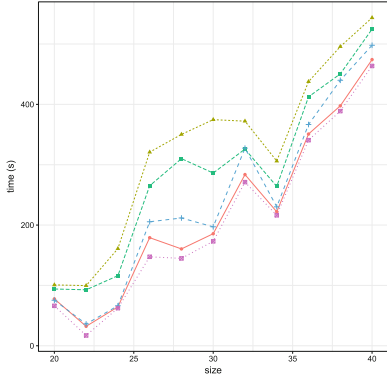
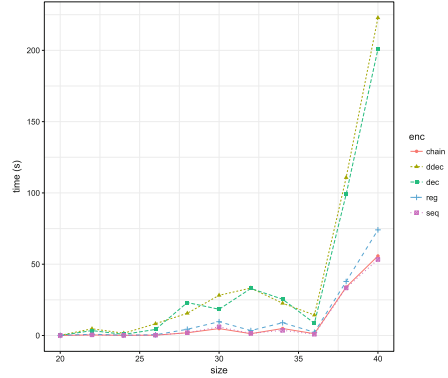
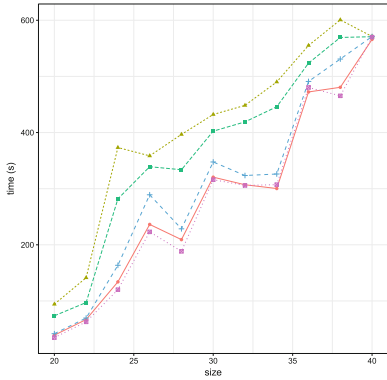
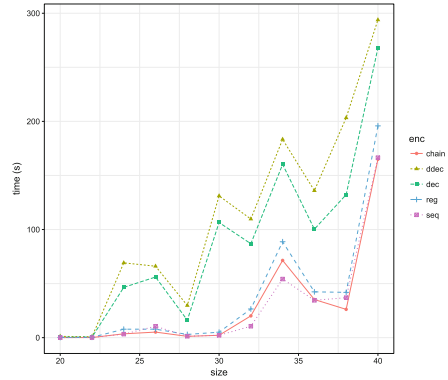

 (a) $|H| = 10$, without learning

 (b) $|H| = 10$, with learning

 (a) $|H| = 14$, without learning

 (b) $|H| = 14$, with learning

Fig. 8. Mean time to solve capacitated concert hall scheduling instances for 10 halls (above) and 14 (below) for each VALUE-PRECEDE-CHAIN implementation.

5.3 Graph Colouring

We also evaluated the VALUE-PRECEDE-CHAIN implementations on the graph colouring problems from [12]. In these problems, vertices are partitioned into equivalence classes. Each class is either complete or empty; and if there is an edge between any members of two equivalence classes, there are edges between all pairs of members of the two classes. For search, we simply assign the minimum colour to vertices in input order. The results are shown in Fig. 9. Without learning, most of the solving time is spent in disequality reasoning, so differences between encodings are less pronounced. Even so, the new encodings are consistently faster. With learning enabled, we see the performance of REG degrade rapidly as the instance sizes increase.

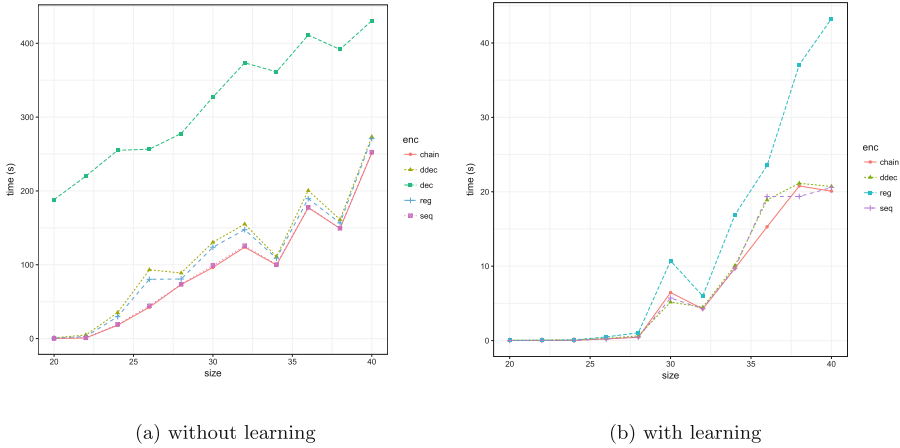


Fig. 9. Mean time required to solve graph colouring instances, using each VALUE-PRECEDE-CHAIN implementation. Results for DEC are again omitted in (b).

The main lesson we take from these experiments is that SEQ is the fastest, most robust encoding of VALUE-PRECEDE-CHAIN for both learning and non-learning solvers. The domain-consistent propagator CHAIN has similar performance, but does not outperform SEQ enough to justify its implementation.

6 Conclusion

We define SEQ-PRECEDE-CHAIN to encode the most common usages of VALUE-PRECEDE-CHAIN. We define a new efficient global propagator for this constraint, and how to explain its propagations. This led us to an efficient decomposition, which actually creates more reusable explanations, and is competitive in propagation speed with the global because it is naturally incremental. We propose that the standard decomposition for VALUE-PRECEDE-CHAIN in the MiniZinc library make use of our decomposition.

Acknowledgements. This research is supported by the Australian Research Council through grant DE160100568 and the Asian Office of Aerospace Research and Development grant 15-4016.

References

1. Law, Y.C., Lee, J.H.M.: Global constraints for integer and set value precedence. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 362–376. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30201-8_28
2. Beldiceanu, N., Carlsson, M., Régim, J., Demassey, S.: Global constraint catalogue: `int_value_precede_chain`. http://sofdem.github.io/gccat/gccat/Cint_value_precede_chain.html. Accessed April 2018

3. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 482–495. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30201-8_36
4. Cheng, K.C.K., Yap, R.H.C.: Maintaining generalized arc consistency on ad hoc r -ary constraints. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 509–523. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85958-1_34
5. Perez, G., Régin, J.-C.: Improving GAC-4 for table and MDD constraints. In: O’Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 606–621. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_44
6. Gange, G., Stuckey, P.J., Szymanek, R.: MDD propagators with explanation. *Constraints* **16**(4), 407–429 (2011)
7. Katsirelos, G., Narodytska, N., Walsh, T.: Reformulating global grammar constraints. In: van Hoes, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 132–147. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01929-6_11
8. Abío, I., Gange, G., Mayer-Eichberger, V., Stuckey, P.J.: On CNF encodings of decision diagrams. In: Quimper, C.-G. (ed.) CPAIOR 2016. LNCS, vol. 9676, pp. 1–17. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33954-2_1
9. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. *Constraints* **14**(3), 357–391 (2009)
10. Freuder, E.C.: A sufficient condition for backtrack-free search. *J. ACM* **29**(1), 24–32 (1982)
11. Chu, G.: Improving combinatorial optimization. Ph.D. thesis, Department of Computing and Information Systems, University of Melbourne (2011)
12. Law, Y.C., Lee, J.H.: Symmetry breaking constraints for value symmetries in constraint satisfaction. *Constraints* **11**(2–3), 221–267 (2006)



An Incremental SAT-Based Approach to Reason Efficiently on Qualitative Constraint Networks

Gael Glorian¹(✉), Jean-Marie Lagniez¹, Valentin Montmirail¹,
and Michael Sioutis²

¹ CRIL, Artois University and CNRS, 62300 Lens, France
{glorian,lagniez,montmirail}@cril.fr

² Örebro Universitet, MPI@AASS, Örebro, Sweden
michael.sioutis@oru.se

Abstract. The $\mathcal{RCC8}$ language is a widely-studied formalism for describing topological arrangements of spatial regions. Two fundamental reasoning problems that are associated with $\mathcal{RCC8}$ are the problems of *satisfiability* and *realization*. Given a qualitative constraint network (QCN) of $\mathcal{RCC8}$, the satisfiability problem is deciding whether it is possible to assign regions to the spatial variables of the QCN in such a way that all of its constraints are satisfied (*solution*). The realization problem is producing an actual spatial model that can serve as a solution. Researchers in $\mathcal{RCC8}$ focus either on symbolically checking the satisfiability of a QCN or on presenting a method to realize (valuate) a satisfiable QCN. To the best of our knowledge, a combination of those two lines of research has not been considered in the literature in a unified and homogeneous approach, as the first line deals with native constraint-based methods, and the second one with rich mathematical structures that are difficult to implement. In this article, we combine the two aforementioned lines of research and explore the opportunities that surface by interrelating the corresponding reasoning problems, viz., the problems of satisfiability and realization. We restrict ourselves to QCNs that, when satisfiable, are realizable with rectangles. In particular, we propose an *incremental* SAT-based approach for providing a framework that reasons about the $\mathcal{RCC8}$ language in a counterexample-guided manner. The incrementality of our approach also avoids the usual blow-up and the lack of scalability in SAT-based encodings. Specifically, our SAT-translation is parsimonious, *i.e.*, constraints are added incrementally in a manner that guides the embedded SAT-solver and forbids it to find the same counterexample twice. We experimentally evaluated our approach and studied its scalability against state-of-the-art solvers for reasoning about $\mathcal{RCC8}$ relations using a varied dataset of instances. The approach scales up and is competitive with the state of the art for the considered benchmarks.

Keywords: \mathcal{RCC} · Qualitative Spatial and Temporal Reasoning
SAT · CEGAR

1 Introduction

Qualitative Spatial and Temporal Reasoning (QSTR) is a major field of study in Artificial Intelligence, and in particular in Knowledge Representation & Reasoning, that deals with the fundamental cognitive concepts of space and time in an abstract, qualitative, and human-like manner. By way of illustration, in natural language, one uses expressions such as *inside*, *before*, and *north of* to spatially or temporally relate one object with another object or oneself, without resorting to providing quantitative information about these entities. Formally, QSTR restricts the vocabulary of rich mathematical theories that deal with spatial and temporal entities to simple qualitative constraint languages. Thus, QSTR provides a concise framework that allows inexpensive reasoning about entities located in space and time. This framework boosts research and applications to a plethora of areas and domains that include, but are not limited to, ambient intelligence, dynamic GIS, cognitive robotics, spatio-temporal design, and qualitative model generation from video [1–4].

Regarding qualitative spatial reasoning, Randell et al. developed in [5] one of the most well-known and dominant spatial calculi in QSTR, viz., the Region Connection Calculus (*RCC*). It studies the different relations that can be defined between regions in some topological space; these relations are based on the primitive relation of connection. For example, the relation *disconnected* between two regions X and Y suggests that none of the points of region X connects with a point of region Y , and vice versa. Two fragments of *RCC*, namely, *RCC8* and *RCC5* (a sub-language of *RCC8* where no significance is attached to boundaries of regions), have been used in several real-life applications. In particular, Bouzy in [6] used *RCC8* in programming the Go game, Lattner et al. in [7] used *RCC5* to set up assistance systems in intelligent vehicles, Heintz et al. in [8] used *RCC8* in the domain of autonomous *unmanned aircraft systems* (UAS), and Randell et al. in [9] used *RCC8* to correct segmentation errors for images of hematoxylin and eosin (H&E)-stained human carcinoma cell line cultures. Other typical applications of *RCC* involve robot navigation, high level vision, and natural language processing [2]. *RCC8* (which will be the focus of this paper) is based on the following eight relations: equals (**EQ**), partially overlaps (**PO**), externally connected (**EC**), disconnected (**DC**), tangential proper part (**TPP**) and its inverse (**TPP**⁻¹), and non-tangential proper part (**NTPP**) and its inverse (**NTPP**⁻¹). These spatial relations are illustrated in Fig. 1.

Given a qualitative constraint network (QCN) of *RCC8*, we are particularly interested in its *satisfiability* problem, which is the problem of deciding if there exists a spatial interpretation of the variables of the QCN that satisfies its constraints. The satisfiability problem for *RCC8* (and *RCC5*) is NP-complete [10]. Once a QCN of *RCC8* is known to be satisfiable, thus having only one relation at each edge without any choice possible, one typically deals with the *realization problem* in order to produce an actual spatial model that can serve as a solution, which is a tractable problem (see [11]). Other fundamental reasoning problems include the *minimal labeling* (or *deductive closure*) problem and the *redundancy* problem [12]. The minimal labeling problem is the problem of

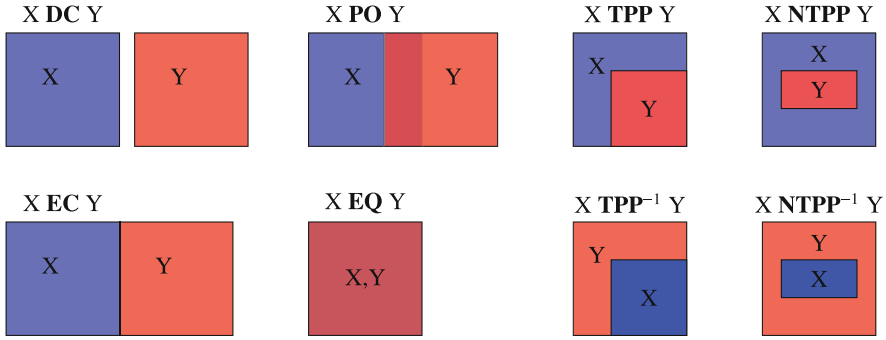


Fig. 1. Illustration of the base $\mathcal{RCC8}$ relations

finding the strongest implied constraints of the QCN, and the redundancy problem is the problem of determining if a given constraint in the QCN is entailed by the rest of the network (that constraint being called redundant, as its removal does not change the solution set of the QCN). The problems of redundancy, minimal labeling, and satisfiability are all equivalent under polynomial Turing reductions [13].

Research in $\mathcal{RCC8}$ usually focuses either on symbolically checking the satisfiability of a QCN or on presenting a method to realize (valuate) a satisfiable QCN. To the best of our knowledge, combining those two lines of research in an interrelating manner has not been considered in the literature, as the first line deals with native constraint-based methods, and the second one with rich mathematical structures that are difficult to implement. In this paper, we bind those two lines of research together in a unified and homogeneous approach by means of an incremental SAT-based technique known as CEGAR, which stands for **C**ounter-**E**xample **G**uided **A**bstraction **R**efinement [14]. The idea is as follows: instead of creating an equisatisfiable propositional formula as per the state of the art [15], we generate an *under-approximation* formula (a formula which is under-constrained, also called *relaxation* in other domains). Meaning, if an under-approximation is unsatisfiable, then by construction the original formula is unsatisfiable; otherwise, the SAT solver outputs a model that can then be checked. It could be the case that the approach is lucky and the model of the under-approximation is also a model of the original formula, in which case the problem is decided. In general, the under-approximation is constantly refined, *i.e.*, it comes closer to the original formula and, in the worst-case, it will eventually become equisatisfiable with the original formula after a finite number of refinements. Notably, CEGAR has been successfully proposed in many problems such as Bounded Model Checking [14], Satisfiability Modulo Theory [16], Planning [17], the Hamiltonian Cycle Problem [18] and more recently within Quantified Boolean Formulas (QBF) [19,20].

The idea of abstracting decision problems with a CEGAR-under approach is well known in the SAT-community. However, the CP/OR community is probably

more familiar with the Logic-based Benders decomposition (LBB) [21], which can be viewed as the CEGAR-under approach for optimization. It is used in many domains where we want to abstract and then solve an optimization problem. LBB approaches are orders of magnitude faster than state-of-the-art MIP for all problems where it has been applied [22–24], just as their CEGAR counterparts are against a direct encoding. One could also see the CEGAR-under approach as a kind of Lazy-SMT approach [25, 26], where the problem-specific knowledge that is extracted from an abstraction is used to guide the refinement process, instead of a theory solver.

2 Preliminaries

In this section, we will assume that the reader is familiar with basic notions from graph theory and topology such as chordal graph, open and closed sets, the interior and closure operators and with basic notions from geometry.

2.1 Region Connection Calculus

The Region Connection Calculus (\mathcal{RCC}) [5] is a first order theory for representing and reasoning about mereotopological information between regions of some topological space. Its relations are based on a connectedness relation \mathbf{C} . In particular, using \mathbf{C} , a set of binary relations is defined. From this set, the $\mathcal{RCC8}$ fragment can be extracted: $\{\mathbf{DC}, \mathbf{EC}, \mathbf{PO}, \mathbf{EQ}, \mathbf{TPP}, \mathbf{NTPP}, \mathbf{TPP}^{-1}, \mathbf{NTPP}^{-1}\}$. These eight ones are jointly exhaustive and pairwise disjoint, meaning that only one of those can hold between any two regions. As noted in the introduction, this fragment (illustrated in Fig. 1), will be referred to simply as $\mathcal{RCC8}$ for convenience.

We can view regions in \mathcal{RCC} as non-empty regular subsets of some topological space that do not have to be internally connected and do not have a particular dimension, but that are usually required to be *closed* [27] (*i.e.*, the subsets equal the closure of their respective interiors). Let $R(\mathcal{X})$ denote the set of all regions of some topological space \mathcal{X} . Then, we can have the following interpretation for the basic relations of $\mathcal{RCC8}$, where \mathbf{R}_i denotes the interpretation of \mathbf{R} for two instantiated region variables. Semantically, binary relation \mathbf{R} contains all the possible instantiations of its pair of region variables.

Definition 1 (Set Notation of $\mathcal{RCC8}$). *Given two regions X and Y in $R(\mathcal{X})$, then:*¹

$$\begin{array}{lll}
 \mathbf{EQ}_i(X, Y) & \text{iff} & X = Y \\
 \mathbf{DC}_i(X, Y) & \text{iff} & X \cap Y = \emptyset \\
 \mathbf{EC}_i(X, Y) & \text{iff} & \overset{\circ}{X} \cap \overset{\circ}{Y} = \emptyset, X \cap Y \neq \emptyset
 \end{array}$$

¹ $\overset{\circ}{A}$ denotes the interior of A .

$$\begin{array}{lll}
 PO_i(X, Y) & \text{iff} & \dot{X} \cap \dot{Y} \neq \emptyset, X \not\subseteq Y, Y \not\subseteq X \\
 TPP_i(X, Y) & \text{iff} & X \subset Y, X \not\subseteq \dot{Y} \\
 TPP_i^{-1}(X, Y) & \text{iff} & Y \subset X, Y \not\subseteq \dot{X} \\
 NTPP_i(X, Y) & \text{iff} & X \subset \dot{Y} \\
 NTPP_i^{-1}(X, Y) & \text{iff} & Y \subset \dot{X}
 \end{array}$$

Given two basic relations \mathbf{R} and \mathbf{S} of $\mathcal{RCC8}$ that involve the pair of variables (i, j) and (j, k) respectively, the *weak composition* of \mathbf{R} and \mathbf{S} , denoted by $CT(\mathbf{R}, \mathbf{S})$, yields the strongest relation of $\mathcal{RCC8}$ that contains $\mathbf{R} \circ \mathbf{S}$, *i.e.*, it yields the smallest set of basic relations such that, each of which can be satisfied by the instantiated variables i and k for some possible instantiation of variables i, j, k with respect to relations \mathbf{R} and \mathbf{S} . We remind the following definition of the weak composition operation from [28]:

Definition 2 (Weak Composition CT). For two basic relations \mathbf{R}, \mathbf{S} of $\mathcal{RCC8}$, their weak composition $CT(\mathbf{R}, \mathbf{S})$ is defined to be the smallest subset $\{\mathbf{T}_1, \mathbf{T}_2, \dots, \mathbf{T}_n\}$ of $2^{\mathcal{RCC8}}$ such that $\mathbf{T}_i \cap (\mathbf{R} \circ \mathbf{S}) \neq \emptyset \forall i \in \{1, \dots, n\}$.

The result of the weak composition operation for each possible pair of basic relations of $\mathcal{RCC8}$ is provided by a dedicated table, called the *weak composition table* [29] ($\mathcal{RCC8}$ CT for short), shown in Table 1. The weak composition operation for two general $\mathcal{RCC8}$ relations can be computed by unifying the results (sets) of the weak composition operations for all ordered pairs of basic relations that involve a basic relation from the first general relation and a basic relation

Table 1. The $\mathcal{RCC8}$ CT , where * specifies the universal relation

CT	DC	EC	PO	TPP	NTPP	TPP^{-1}	$NTPP^{-1}$	EQ
DC	*	DC EC PO TPP NTPP	DC EC PO TPP NTPP	DC EC PO TPP NTPP	DC EC PO TPP NTPP	DC	DC	DC
EC	DC EC PO TPP^{-1} $NTPP^{-1}$	DC EC PO TPP TPP^{-1} EQ	DC EC PO TPP NTPP	EC PO TPP NTPP	PO TPP NTPP	DC EC	DC	EC
PO	DC EC PO TPP^{-1} $NTPP^{-1}$	DC EC PO TPP^{-1} $NTPP^{-1}$	*	PO TPP NTPP	PO TPP NTPP	DC EC PO TPP^{-1} $NTPP^{-1}$	DC EC PO TPP^{-1} $NTPP^{-1}$	PO
TPP	DC	DC EC	DC EC PO TPP NTPP	TPP NTPP	NTPP	DC EC PO TPP TPP^{-1} EQ	DC EC PO TPP^{-1} $NTPP^{-1}$	TPP
NTPP	DC	DC	DC EC PO TPP NTPP	NTPP	NTPP	DC EC PO TPP NTPP	*	NTPP
TPP^{-1}	DC EC PO TPP^{-1} $NTPP^{-1}$	EC PO TPP^{-1} $NTPP^{-1}$	PO TPP^{-1} $NTPP^{-1}$	PO TPP TPP^{-1} EQ	PO TPP NTPP	TPP^{-1} $NTPP^{-1}$	$NTPP^{-1}$	TPP^{-1}
$NTPP^{-1}$	DC EC PO TPP^{-1} $NTPP^{-1}$	PO TPP^{-1} $NTPP^{-1}$	PO TPP^{-1} $NTPP^{-1}$	PO TPP^{-1} $NTPP^{-1}$	PO TPP NTPP TPP^{-1} $NTPP^{-1}$ EQ	$NTPP^{-1}$	$NTPP^{-1}$	$NTPP^{-1}$
EQ	DC	EC	PO	TPP	NTPP	TPP^{-1}	$NTPP^{-1}$	EQ

from the second one. Henceforward, a general \mathcal{RCCS} relation will be represented by the set of its basic relations.

In order to concretely capture the qualitative spatial information that is entailed by a knowledge base of \mathcal{RCCS} relations, we will use the notion of a Qualitative Constraint Network (QCN), defined as follows:

Definition 3 (Qualitative Constraint Networks (QCN)). A QCN of \mathcal{RCCS} is a pair $\mathcal{N} = (V, C)$ where V is a non-empty finite set of variables (each one corresponding to a region), and C is a mapping associating a relation $C(v, v') \in 2^{\mathcal{RCCS}}$ with each pair (v, v') of $V \times V$. Further, mapping C is such that $C(v, v) \subseteq \{\mathbf{EQ}\}$ and $C(v, v') = (C(v', v))^{-1}$.

Concerning a QCN $\mathcal{N} = (V, C)$, we have the following definitions: An instantiation of V is a mapping σ defined from V to the domain $R(\mathcal{X})$. A solution (realization) σ of \mathcal{N} is an instantiation of V such that for every pair (v, v') of variables in V , $(\sigma(v), \sigma(v'))$ satisfies $C(v, v')$, i.e., there exists a basic relation $b \in C(v, v')$ such that $(\sigma(v), \sigma(v')) \in b$. \mathcal{N} is satisfiable if and only if it admits a solution. The constraint graph of a QCN \mathcal{N} is the graph (V, E) , denoted by $G_{\mathcal{N}}$, for which we have that $\{v, v'\} \in E$ if and only if $C(v, v') \neq \mathcal{RCCS}$ (i.e., $C(v, v')$ corresponds to a non-universal relation) and $v \neq v'$.

In this article, we restrict ourselves to QCNs that, when satisfiable, are realizable with rectangles. As pointed out in [30, Example 1], there exist QCNs for which it is not possible to find a rectangular realization using a single rectangle for a region (though it is still possible if more rectangles per region are used). However, for hard-to-solve instances, which concern us here and which typically involve a lot of indefinite knowledge and, hence, are flexible in terms of realizing them, it is rarely (if ever) the case that a rectangular realization will not be attainable for a satisfiable QCN (see Sect. 5).

2.2 Propositional Logic

The Propositional Logic will be denoted by CPL . It is the logic of reasoning about what is **True** and what is **False**. The syntax of CPL can be formally defined as follows:

Definition 4 (Language of the Propositional Logic). Let \mathbb{P} be a countably infinite non-empty set of propositional variables. The language of propositional logic (denoted by CPL) is the set of formulas containing \mathbb{P} , closed under the set of propositional connectives $\{\neg, \wedge\}$.

Without loss of generality, we will assume all the formulas of CPL to be in Conjunctive Normal Form (CNF) because any formula can be translated into an equisatisfiable CNF formula using the Tseitin algorithm [31]. Regarding the semantics aspect of the propositional logic, the notion of *interpretation* is important. It is defined as follows:

Definition 5 (Interpretation). An interpretation is a set of valuations of propositional variables. Formally, it is a mapping $\mathbb{P} \rightarrow \{\mathbf{True}, \mathbf{False}\}$.

An interpretation is a *model* of ϕ if ϕ is true for that interpretation. If a formula ϕ has *at least one* model \mathcal{M} , we will say that this formula is *satisfiable*; $\mathcal{M} \models \phi$ will denote that \mathcal{M} satisfies ϕ . Formally, the satisfiability relation is defined as follows:

Definition 6 (Satisfaction Relation in CPL). *The relation \models between Interpretations \mathcal{M} and formulas ϕ in CPL is recursively defined as follows:*

$\mathcal{M} \models p$	iff	$p \in \mathcal{M}$
$\mathcal{M} \models \neg\phi$	iff	$\mathcal{M} \not\models \phi$
$\mathcal{M} \models \phi_1 \wedge \phi_2$	iff	$\mathcal{M} \models \phi_1$ and $\mathcal{M} \models \phi_2$
$\mathcal{M} \models \phi_1 \vee \phi_2$	iff	$\mathcal{M} \models \phi_1$ or $\mathcal{M} \models \phi_2$

If a formula is satisfiable by any interpretation, we will say that this formula is *valid*; in that case the formula is a *tautology* and we will denote it by $\models \phi$. If a formula is false for any interpretation, we will say that this formula is *unsatisfiable*.

2.3 CEGAR Preliminaries

Counter-Example-Guided Abstraction Refinement, CEGAR for short, is an incremental way to decide the satisfiability of formulas in classical propositional logic (CPL). It has been originally designed for model checking [14], *i.e.*, to answer questions such as “Does $S \models P$ hold?” or, likewise, “Is $S \wedge \neg P$ unsatisfiable?”, where S describes a system and P a property. In such highly structured problems, it is often the case that only a small part of the formula is needed to answer the question. The keystone of CEGAR is to replace $\phi = S \wedge \neg P$ by an abstraction ϕ' , where ϕ' should be easier to solve in practice than ϕ . There are two kinds of abstractions: an over-abstraction (resp. under-abstraction) of ϕ is a formula $\hat{\phi}$ (resp. $\check{\phi}$) such that $\hat{\phi} \models \phi$ (resp. $\phi \models \check{\phi}$) holds. $\hat{\phi}$ has at most as many models as ϕ and $\check{\phi}$ has at least as many models as ϕ . Usually, ϕ is in CNF. An illustration of a CEGAR approach using under-abstraction is given in Fig. 2.

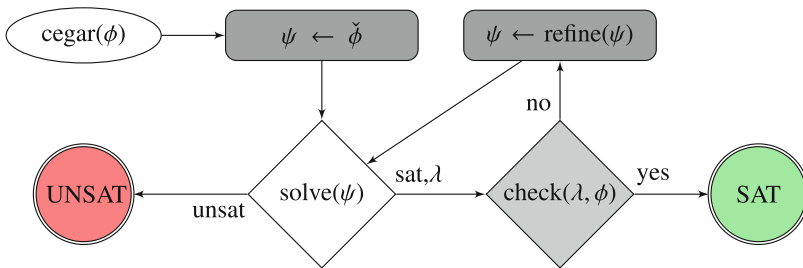


Fig. 2. The CEGAR framework with under-abstraction

To be sound, complete, and to terminate, a CEGAR approach has to verify the following assumptions (proofs can be obtained from [32, Theorems 1, 2 and 3]):

1. “solve” is sound, complete, and terminates;
2. if $\check{\phi}$ is unsatisfiable, then ϕ is unsatisfiable;
3. “check(λ, ϕ)” returns true if λ is a model of ϕ ;
4. $\exists n \in \mathbb{N}$ such that $\text{refine}^n(\check{\phi})$ is equisatisfiable with ϕ .

As we will use a SAT solver in our approach, we will suppose that the embedded SAT solver is well coded and that it is sound, complete, and terminates, so that CEGAR-under Assumption (1) is satisfied. Having introduced the notions needed to understand the contributions, we can proceed to the first of them, which is encoding $\mathcal{RCC8}$ into propositional logic in such a convenient way that it allows us to easily verify the CEGAR-under Assumptions (2) and (4).

3 Encoding $\mathcal{RCC8}$ into SAT

To obtain a SAT encoding of the $\mathcal{RCC8}$ satisfiability problem, we need to define how to translate the different possible relations. We will represent a region i as a set of four variables $\{x_i^-, y_i^-, x_i^+, y_i^+\}$ as illustrated in Fig. 3.

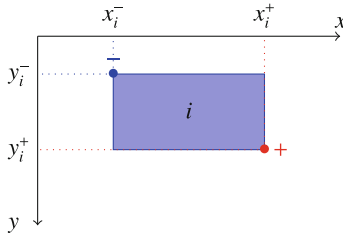


Fig. 3. Illustration of how a region is represented

All the possible cases for every relation may be found and proved, along with their link with Point Algebra, in [33, Table 6.2]. From this encoding, we can then propose the following SAT encoding which translates all the edges possible:

Definition 7 (SAT Translation – tr). For all relations R in all the given edges (i, j) of the input problem \mathcal{N} we have:

$$\text{tr}(\mathcal{N}) := \bigwedge_{\forall (R,i,j) \in \mathcal{N}} \text{tr}(R, i, j)$$

Then from [33, Table 6.2], if we want to translate for example the relation **EC** between nodes i and j (the procedure is similar for other $\mathcal{RCC8}$ relations), we will have the following SAT encoding as per Definition 7:

Definition 8 (SAT Translation of EC on the edge i-j)

$$\text{tr}(\mathbf{EC}, i, j) := \mathbf{EC}(i, j) \rightarrow (\mathbf{EC}_r(i, j) \vee \mathbf{EC}_l(i, j) \vee \mathbf{EC}_u(i, j) \vee \mathbf{EC}_d(i, j))$$

From this definition, we can see that the relation \mathbf{EC} for the edge (i, j) can only be satisfied by 4 different cases, viz., *left*, *right*, *up*, *down*. Each case is defined as follows:

$$\begin{array}{ll} \mathbf{EC}_r(i, j) \rightarrow ((x_i^- < x_j^-) \wedge (x_i^- < x_j^+)) \wedge & \mathbf{EC}_u(i, j) \rightarrow ((x_i^- < x_j^-) \vee (x_i^- = x_j^-)) \wedge \\ ((x_i^- = x_j^+) \wedge (x_i^+ < x_j^+)) \wedge & ((x_i^- > x_j^+) \vee (x_i^- = x_j^+)) \wedge \\ ((y_i^- < y_j^+) \vee (y_i^- < y_j^-)) \wedge & ((y_i^- < y_j^-) \wedge (y_i^- < y_i^+)) \wedge \\ ((y_i^+ < y_j^-) \vee (y_i^+ < y_j^-)) & ((y_i^+ = y_j^-) \wedge (y_i^+ < y_j^+)) \end{array}$$

$$\begin{array}{ll} \mathbf{EC}_l(i, j) \rightarrow ((x_i^- > x_j^-) \wedge (x_i^- = x_j^+)) \wedge & \mathbf{EC}_d(i, j) \rightarrow ((x_i^- < x_j^-) \vee (x_i^- = x_j^-)) \wedge \\ ((x_i^- > x_j^+) \wedge (x_i^+ > x_j^+)) \wedge & ((x_i^- > x_j^+) \vee (x_i^- = x_j^+)) \wedge \\ ((y_i^- < y_j^+) \vee (y_i^- < y_j^+)) \wedge & ((y_i^- > y_j^-) \wedge (y_i^- = y_i^+)) \wedge \\ ((y_i^+ < y_j^-) \vee (y_i^+ < y_j^-)) & ((y_i^+ > y_j^-) \wedge (y_i^+ > y_j^+)) \end{array}$$

The inverse relations are defined as usual: $\mathbf{TPP}^{-1}(i, j) = \mathbf{TPP}(j, i)$ and $\mathbf{NTPP}^{-1}(i, j) = \mathbf{NTPP}(j, i)$. For every node in the QCN with N nodes that we want to solve, we will add the following constraint assuring that all the point coordinates are in good order:

$$\bigwedge_{i=1}^N ((x_i^- < x_i^+) \wedge (y_i^- < y_i^+))$$

We want to point out that, if the propositional variable $(A < B)$ is true, then the variables $(A > B)$ and $(A = B)$ are false. To express this, we use the following clauses:

$$\text{AMO} := \bigwedge_{a \in \{x, y\}} \bigwedge_{c_1 \in \{-, +\}} \bigwedge_{c_2 \in \{-, +\}} \bigwedge_{i=1}^N \bigwedge_{j=1}^N \left(\begin{array}{l} ((a_i^{c_1} < a_j^{c_2}) \vee (a_i^{c_1} = a_j^{c_2}) \vee (a_i^{c_1} > a_j^{c_2})) \wedge \\ (\neg(a_i^{c_1} < a_j^{c_2}) \vee \neg(a_i^{c_1} = a_j^{c_2})) \wedge \\ (\neg(a_i^{c_1} < a_j^{c_2}) \vee \neg(a_i^{c_1} > a_j^{c_2})) \wedge \\ (\neg(a_i^{c_1} = a_j^{c_2}) \vee \neg(a_i^{c_1} > a_j^{c_2})) \wedge \end{array} \right) \quad (1)$$

Thanks to Eq. 1 (AMO – At Most One), we can thus replace, for example, in $\mathbf{EC}(i, j)$ (u), $(x_i^- < x_j^+) \vee (x_i^- = x_j^+)$ by $\neg(x_i^- > x_j^+)$. The same applies for all the disjunctions in [33, Table 6.2]. Last but not least, we want to ensure the transitivity of the relations on all the possible coordinates; this will have the biggest impact on the size of the generated CNF. For all the triplets (i, j, k) in a triangulation (chordal completion of the constraint graph of an input QCN), we

must add the following rules for every combination $(c1, c2)$ that can be assured by transitivity $\in \{(-, -), (-, +), (+, -), (+, +)\}$ and for both axis $a \in \{x, y\}$:

$$\text{transitivity}(i, j, k) := \bigwedge \left(\begin{array}{l} ((a_i^{c1} = a_j^{c1}) \wedge (a_j^{c1} = a_k^{c2})) \rightarrow (a_i^{c1} = a_k^{c2}) \\ ((a_i^{c1} < a_j^{c1}) \wedge \neg(a_j^{c1} > a_k^{c2})) \rightarrow (a_i^{c1} < a_k^{c2}) \\ ((a_i^{c1} > a_j^{c1}) \wedge \neg(a_j^{c1} < a_k^{c2})) \rightarrow (a_i^{c1} > a_k^{c2}) \\ ((a_j^{c1} > a_k^{c2}) \wedge \neg(a_i^{c1} < a_j^{c1})) \rightarrow (a_i^{c1} > a_k^{c2}) \\ ((a_j^{c1} < a_k^{c2}) \wedge \neg(a_i^{c1} > a_j^{c1})) \rightarrow (a_i^{c1} < a_k^{c2}) \end{array} \right)$$

We will not enter the details of how a graph can be made chordal; it is a standard procedure and we redirect the reader to [34] for more information about how it can be done. It is worth noting that triangulating a graph can take linear time in the size of the output chordal graph. Before moving to the CEGAR part, we need to prove that the encoding we designed is sound and complete.

Theorem 1. *Let $\mathcal{N} = (V, C)$ be a QCN of RCC8, and G a chordal supergraph of the constraint graph of \mathcal{N} . If $\text{toSAT}(\mathcal{N})$ is defined as follows:*

$$\begin{aligned} \text{toSAT}(\mathcal{N}) := & \text{tr}(\mathcal{N}) \wedge \text{AMO} \wedge \bigwedge_{i=1}^N ((x_i^- < x_i^+) \wedge (y_i^- < y_i^+)) \\ & \wedge \bigwedge_{(i,j,k) \in G} \text{transitivity}(i, j, k) \end{aligned}$$

then $\text{toSAT}(\mathcal{N})$ is equisatisfiable with \mathcal{N} .

Proof. We need to show that $\text{toSAT}(\mathcal{N})$ is equisatisfiable with \mathcal{N} . In order to do so, we split $\text{toSAT}(\mathcal{N})$ in two parts. The first one ($\text{tr}(\mathcal{N}) \wedge \text{AMO} \wedge \bigwedge_{i=1}^N ((x_i^- < x_i^+) \wedge (y_i^- < y_i^+))$) is obviously the input problem; this representation comes from [33] and the relation with Point Algebra (PA). As proven in [35], it is enough to check the path consistency with respect to the chordal graph, so the real difficulty of this proof is demonstrating why by adding a finite number of transitivity constraints does the translation become equisatisfiable. The intuition is that, each time we add a transitivity constraint $\text{transitivity}(i, j, k)$, we force the SAT solver to find only relations in this triangle which match the weak composition table CT (Table 1). For this purpose, we need to enumerate all the cases pertaining to the CT and show that, each time, the transitivity constraints force the solver to find only relations allowed by the CT. Let us consider the following case: (i, j) has the relation **EQ** and (j, k) has the relation **NTPP**. Then by the CT, the transitivity constraints should force to find **NTPP** on (i, k) . Because of what we assume true, we have the following propositional variables assigned to true: $(x_i^- = x_j^-), (y_i^- = y_j^-), (x_i^- < x_j^+), (y_i^- < y_j^+), (x_i^+ > x_j^-), (y_i^+ > y_j^-), (x_i^+ = x_j^+), (y_i^+ = y_j^+)$ (which encodes **EQ** (i, j)) and

$(x_j^- > x_k^-), (y_j^- > y_k^-), (x_j^- < x_k^-), (y_j^- < y_k^+), (x_j^- > x_k^+), (y_j^- > y_k^-), (x_j^- < x_k^+), (y_j^- < y_k^+)$ (which encodes **NTPP**(j, k)). Then, we want to obtain the following:

$$\begin{array}{ll}
 (1.a) \ (x_i^- > x_k^-) & (1.b) \ (y_i^- > y_k^-) \\
 (2.a) \ (x_j^- < x_k^-) & (2.b) \ (y_j^- < y_k^+) \\
 (3.a) \ (x_j^- > x_k^+) & (3.b) \ (y_j^- > y_k^-) \\
 (4.a) \ (x_j^- < x_k^+) & (4.b) \ (y_j^- < y_k^+)
 \end{array}$$

- 1.a We have $(x_i^- = x_j^-)$ and $(x_j^- > x_k^-)$. Due to the transitivity constraint **transitivity**(i, j, k) at one point, we add the clause: $((x_i^- = x_j^-) \wedge \neg(x_j^- < x_k^-) \rightarrow (x_i^- > x_k^-))$. Then, because of **AMO**, we have $((x_j^- > x_k^-) \rightarrow \neg(x_j^- < x_k^-))$. Thus we have $(x_i^- > x_k^-)$.
- 2.a We have $(x_i^- = x_j^-)$ and $(x_j^- < x_k^+)$. Due to the transitivity constraint **transitivity**(i, j, k) at one point, we add the clause: $(\neg(x_i^- < x_j^-) \wedge (x_j^- < x_k^+) \rightarrow (x_i^- < x_k^+))$. Then, because of **AMO**, we have $((x_i^- = x_j^-) \rightarrow \neg(x_i^- < x_j^-))$. Thus we have $(x_i^- < x_k^+)$.
- 3.a We have $(x_i^+ = x_j^+)$ and $(x_j^+ > x_k^-)$. Due to the transitivity constraint **transitivity**(i, j, k) at one point, we add the clause: $(\neg(x_i^+ < x_j^+) \wedge (x_j^+ > x_k^-) \rightarrow (x_i^+ > x_k^-))$. Then, because of **AMO**, we have $((x_i^+ = x_j^+) \rightarrow \neg(x_i^+ < x_j^+))$. Thus we have $(x_i^+ > x_k^-)$.
- 4.a We have $(x_i^+ = x_j^+)$ and $(x_j^+ < x_k^+)$. Due to the transitivity constraint **transitivity**(i, j, k) at one point, we add the clause: $(\neg(x_i^+ > x_j^+) \wedge (x_j^+ < x_k^+) \rightarrow (x_i^+ > x_k^+))$. Then, because of **AMO**, we have $((x_i^+ = x_j^+) \rightarrow \neg(x_i^+ > x_j^+))$. Thus we have $(x_i^+ < x_k^+)$.

The cases on the y-axis (1.b, 2.b, 3.b and 4.b) are similar to the one on the x-axis. From this point forward, we just have to enumerate in that way all the possible cases pertaining to the weak composition table. This would be extremely space-consuming so we leave it as exercise for the reader. \square

We just saw that if we encode all the transitivity cases for every triangle in the respective chordal graph in accordance with our description, we obtain an equisatisfiable SAT encoding. However, when we take a closer look at what can be time-consuming, function **transitivity** is exactly what we want to avoid at any cost due to the size of its encoding, and that is why we propose a CEGAR approach to circumvent it.

4 Translating Parsimoniously the Transitivity Constraints

As we explained earlier, the function **transitivity** can be costly and it hence needs to be avoided if we want to have a competitive approach. This is exactly the hypothesis on which our CEGAR approach rests. Let us take the example

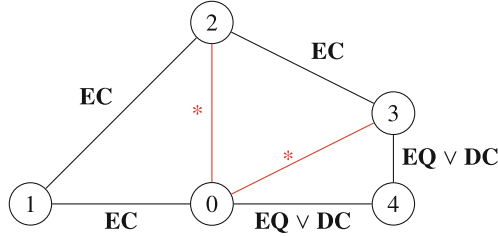


Fig. 4. An example of the constraint graph of an $\mathcal{RCC8}$ problem ϕ , (the labels denote the corresponding $\mathcal{RCC8}$ relations), in red the edges added to make the graph chordal

illustrated in Fig. 4, a $\mathcal{RCC8}$ problem with 5 nodes and 5 relations given as input (**the relations are shown in black in Fig. 4**). Thus, when we translate this problem into a propositional logic formula, we obtain the following rules:

$$\begin{aligned}
 \text{under}(\phi) = & \text{tr}(\mathbf{EC}, 0, 1) \wedge \text{tr}(\mathbf{EC}, 1, 2) \wedge \text{tr}(\mathbf{EC}, 2, 3) \wedge \text{tr}(\mathbf{EC}, 0, 1) \\
 & \wedge \text{tr}(\mathbf{EQ}, 3, 4) \wedge \text{tr}(\mathbf{DC}, 3, 4) \wedge \text{tr}(\mathbf{EQ}, 4, 0) \wedge \text{tr}(\mathbf{DC}, 4, 0) \\
 & \wedge EC_{0,1} \wedge EC_{1,2} \wedge EC_{2,3} \wedge (EQ_{3,4} \vee DC_{3,4}) \wedge (EQ_{4,0} \vee DC_{4,0}) \\
 & \wedge \mathbf{AMO} \wedge \bigwedge_{i=0}^4 ((x_i^- < x_i^+) \wedge (y_i^- < y_i^+))
 \end{aligned}$$

Theorem 2. *Let ϕ be a QCN, then $\text{under}(\phi)$ is an under-abstraction of ϕ (i.e., it has at least the same amount of models).*

Proof. $\text{under}(\phi)$ is a subset of clauses of the equisatisfiable encoding (Theorem 1). Thus if $\text{under}(\phi)$ is unsatisfiable, then ϕ is also unsatisfiable by definition of the logical conjunction.

At this point, the translation is an under-abstraction of the original problem, i.e., if it is unsatisfiable, then for sure the problem is unsatisfiable, but a model of this translation does not imply that it exists a model for the original problem. The CEGAR Assumption (2) is respected by construction of the translation, to obtain this equisatisfiability we need to have:

$$\begin{aligned}
 \text{toSAT}(\phi) = & \text{under}(\phi) \\
 & \wedge \text{transitivity}(0, 1, 2) \\
 & \wedge \text{transitivity}(0, 2, 3) \\
 & \wedge \text{transitivity}(0, 3, 4)
 \end{aligned}$$

As the number of triangles in a chordal graph is bounded by a number N (worst case: $N = |V|^3$ when the graph is complete), we can now easily see that after we translate the transitivity of each triangle (an operation that we will call a refinement in what follows), we can refine the problem N times and obtain an equisatisfiable formula. This allows us to respect the CEGAR Assumption (4).

Algorithm 1. CEGAR- $\mathcal{RCC8}(\mathcal{N})$

Data: $\mathcal{N}=(V,C)$ with n variables**Result:** A realization of \mathcal{N} if it is possible to obtain one, UNSAT otherwise

```

1  $G \leftarrow (V, E \leftarrow E(G_{\mathcal{N}}))$ ;
2  $\text{setOfTriangles} \leftarrow \text{Chordal}(G)$ ;
3  $\text{transitivity} \leftarrow \top$ ;
4  $\psi \leftarrow \text{under}(N)$ ; // under-abstraction step
5 while ( $\text{setOfTriangle} \neq \emptyset$ ) do
6    $\lambda \leftarrow \text{SAT-Solver}(\psi \wedge \text{transitivity})$ ; // solve step
7   if ( $\lambda = \perp$ ) then return UNSAT;
8    $\text{res} \leftarrow \text{check}(\lambda, N)$ ; // check step
9   if ( $\text{res} = \text{null}$ ) then return  $\text{interpret}(\lambda)$ ;
10  else
11     $\text{setOfTriangle.remove}(\text{res})$ ;
12     $\text{transitivity} \leftarrow \text{transitivity} \wedge \text{transitivity}(\text{res})$ ; // refinement step
13  $\lambda \leftarrow \text{SAT-Solver}(\psi \wedge \text{transitivity})$ ; // worst case: equisatisfiability
14 if ( $\lambda = \perp$ ) then return UNSAT;
15 else return  $\text{interpret}(\lambda)$ ;

```

Concerning the CEGAR Assumption (3), we need to find a way to check efficiently if a returned model of the under-abstraction is also a model of the original formula. For this purpose, we used the algorithm Directional Path Consistency (DPC) presented in [36–38]. Our function `check` performs the model-checking and returns the triangle which results in the assignment of the empty set to some relation. From this point forward, if the checker returns the triangle (i, j, k) and we consequently add the transitivity constraint $\text{transitivity}(i, j, k)$ in the propositional formula, then it is impossible for the checker to return once again the same triangle. As discussed earlier, the maximum set of transitivity constraints that we need to add is of finite size, at which point we will have an equisatisfiable formula.

We now have all the pieces to create two different ways to solve the satisfiability and at the same time the realization problem in $\mathcal{RCC8}$. The first one is by using a direct encoding (with the function $\text{toSAT}(\mathcal{N})$). The second one is by using a CEGAR approach for it, like the one presented in Algorithm 1, which in the worst-case (the case where all the transitivity rules must be considered) will end-up being just a slightly slower version of the direct encoding; however, this has been experimentally found to never occur in practice (see Sect. 5). Moreover, every time we have that the instance is satisfiable, we also obtain an interpretation of the model returned by the SAT solver. In other words, we solve the satisfiability and realization problems together.

5 Experimental Results

Now that we have a new SAT encoding and a CEGAR approach for solving the satisfiability and realization problems in $\mathcal{RCC8}$, we want to compare

against the state-of-the-art. For this purpose, we implemented the approach within the solver Churchill² and we used Glucose [39,40] as an internal SAT solver. We will compare Churchill in direct-encoding and CEGAR mode against the state-of-the-art qualitative spatial reasoners for $\mathcal{RCC8}$, which are GQR [41], Renz-Nebel01 [42], RCC8SAT [15], PPyRCC8 [35], and Chordal-Phalanx [43]. Each solver is using default settings, except GQR, which is using the flag “-c horn”. By using the flag “-c horn”, GQR decomposes an $\mathcal{RCC8}$ relation into horn sub-relations (which is standard behavior for the rest of the solvers), instead of basic relations; this changes the branching factor from 4 to ~ 1.4 . Moreover, PPyRCC8 and Chordal-Phalanx are run using PyPy as recommended by their authors to improve the overall performance. We compare these solvers on four categories of benchmarks.

1. The first set consists of random hard instances that have been generated with: “`gencsp -i 100 -n 100 -d 10 15 0.5 -r nprels`”.³ In particular, it consists of 100 instances of 100 nodes for every average degree from 10.0 to 15.0 with a 0.5 step and using only relations that result to NP-completeness (*nprels*); thus, a total of 1100 QCNs were generated.
2. The second set is generated exactly like the first one but with 500 nodes instead of 100 and for a range of d between 10.0 and 20.0, consisting of a total of 2100 QCNs.
3. The third set is the *random-scale-free-like-instances* [44], which consists of 300 instances, 30 instances for every size from 1000 to 10000 nodes with a 1000 step. These instances are normal to hard.
4. The fourth set is the *random-scale-free-like-np8-instances* [44], which consists of 70 instances, 10 for every size from 500 to 3500 nodes with a 500 step. These instances are hard to very hard as they are defined solely by *nprels*.

Regarding sets 3 and 4, scale-free networks are networks whose degree distribution follows a power law [45]; notably, structured networks have been used extensively in the recent literature [15,44]. The experiments were run on a cluster of Xeon, 4 cores, 3.3 GHz with CentOS 7.0 with a memory limit of 32 GB and a runtime limit of 900 s per solver per benchmark. All solvers’ answers were checked by verifying if all the solvers gave the same output for each benchmark. No discrepancy was found. This means that the cases where satisfiable QCNs are not realizable with rectangles did not appear in our datasets.

Regarding Figs. 5 and 6, which show the runtime distributions of the different solvers for the first and second set of instances, we can see that Churchill and GQR are extremely fast to solve the respecting sets. Indeed, it took at most 1.42 s for Churchill to solve the hardest instance of the first set and 10.10 s for GQR, and 32.70 s on the second set for Churchill. For Churchill, the speed-up is because we perform in average a small number of CEGAR loops (avg: 8.90 for first set and 11.87 for second set). However, when we take a look at the direct

² The name comes from the historical figure who used to also do a lot of CEGAR.

³ The generator comes with the Renz-Nebel01 solver.

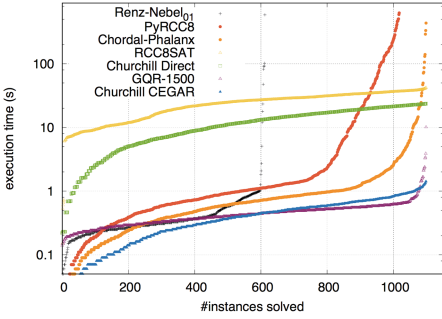


Fig. 5. Runtime distribution on the first set

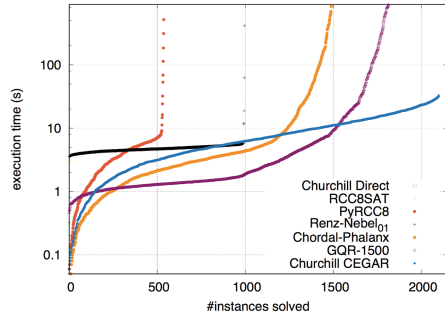


Fig. 6. Runtime distribution on the second set

translations (RCC8SAT and Churchill Direct), 500 nodes is already too much and this blows up the allowed memory.

Table 2 shows the number of benchmarks solved for sets 3 and 4. The best results of a given row are presented in bold and the number of benchmarks which cannot be solved because of lack of memory is provided between parenthesis (if such benchmarks do not exist a dash is displayed). The line VBS represents the Virtual Best Solver (a practical upper-bound on the performance achievable by picking the best solver for each benchmark). On the third set, we can see the

Table 2. Results on sets 3 (left) and 4 (right)

	<i>random-scale-free-like-instances</i>						<i>random-scale-free-like-np8-instances</i>						
#Nodes (x1000)	< 6	6	7	8	9	10	0.5	1	1.5	2	2.5	3	3.5
#Instances	150	30	30	30	30	30	10	10	10	10	10	10	10
Renz-Nebel01	0	0	0	0	0	0	0	0	0	0	0	0	0
	-	-	-	-	-	-	-	-	-	-	-	-	-
RCC8SAT	0	0	0	0	0	0	0	0	0	0	0	0	0
	(150)	(30)	(30)	(30)	(30)	(30)	(10)	(10)	(10)	(10)	(10)	(10)	(10)
PPyRCC8	134	19	20	21	23	23	10	9	10	8	7	3	3
	(14)	(8)	(7)	(7)	(7)	(4)	-	-	-	-	(1)	(5)	(7)
Chordal-Phalanx	150	30	30	30	30	30	10	10	10	10	10	10	10
	-	-	-	-	-	-	-	-	-	-	-	-	-
GQR-1500	150	30	30	30	30	30	10	10	9	6	10	8	9
	-	-	-	-	-	-	-	-	-	-	-	-	-
Churchill Direct	0	0	0	0	0	0	0	0	0	0	0	0	0
	(150)	(30)	(30)	(30)	(30)	(30)	(10)	(10)	(10)	(10)	(10)	(10)	(10)
Churchill CEGAR	150	30	30	18	8	6	10	10	10	10	10	10	10
	-	-	-	(10)	(20)	(24)	-	-	-	-	-	-	-
VBS	150	30	30	30	30	30	10	10	10	10	10	10	10

scalability of a CEGAR approach against a direct encoding (Churchill Direct) or via a CP representation. The results are clear, when the number of nodes is too big, the SAT approaches require too much memory or too much time for the translation of the problem and, hence, become inefficient. The bigger the network, the more time Churchill CEGAR spends model-checking the output of the SAT solver and the more space is required to add transitivity constraints. In some cases, we just reach the space limit and are unable to solve instances.

On the fourth set, we study the scalability on very hard instances that are of reasonable size. We can see here that SAT solvers still have a hard time with the size of the input, and that using a CEGAR approach instead of a direct encoding leads to a huge improvement. Indeed, Churchill CEGAR managed to solve all the instances, but, unfortunately, it took more time than Chordal-Phalanx to do so in most cases (median: 37.97 s for Churchill vs 16.78 for Chordal-Phalanx); however, it was faster in the worst-case (max: 163.10 s for Churchill vs 714.12 s for Chordal-Phalanx). This is mainly due to the fact that model-checking many times, which is typically the case when the network has a size between 2000 and 3500 nodes, is time-consuming. In fact, a sum-up of how the runtime of Churchill is distinguished by Triangulation time, Checking time, and Solving time is given in Table 3. When we analyze the results given in Table 3, the result is clear: when the network is small (first and second sets) the main percentage of the time is spent in the triangulation of the graph. When the network is big (third and fourth sets) the main percentage of the time is spent in the Checking time. But in any case, the SAT solver is not the bottleneck here. For all the results, it is worth remembering that even if we are a little bit slower on sets 3 and 4, we are solving in the same time the realization problem, *i.e.*, we output a solution for the input problem, not only a decision about the satisfiability of that problem.

Table 3. Sum-up of times for the three steps in Churchill

Time (s)	Triangulation			Checking			Solving		
	min	med	max	min	med	max	min	med	max
First set	0.220	0.390	0.630	0.015	0.030	0.151	0.002	0.003	0.010
Second set	3.910	12.910	20.153	0.430	1.170	2.057	0.015	0.030	0.370
Third set	8.708	55.96	128.96	7.900	222.3	668.57	0.015	0.860	16.38
Fourth set	3.765	21.530	68.230	0.560	24.410	58.950	0.012	0.250	15.73

6 Conclusion

In this paper, a new approach for solving the satisfiability and realization problems in $\mathcal{RCC8}$ using an under-abstraction refinement approach within the CEGAR framework has been proposed. We showed that our encoding is sound and complete for satisfiable QCNs that are realizable with rectangles and we

instantiated it within the solver Churchill. We compared our approach against solvers representing, to the best of our knowledge, the state-of-the-art for practical *RCC8* solving, on a wide range of benchmarks of different size and difficulty. We concluded that a basic direct-encoding approach is not competitive at all, because many of the available benchmarks are huge and require a lot of transitivity constraints in the SAT encoding. However, our CEGAR approach, mixing SAT and UNSAT shortcuts, outperformed the other solvers on most of the benchmarks considered.

As future work, seeing Table 3, we could improve the checker. Indeed, one could think about avoiding checking unmodified sub-graphs twice by flagging some nodes, *i.e.*, checking only the part which was modified due to the previous assignment. Moreover, to make the approach sound in any case and not just rectangles, one could think about extending the CEGAR approach into a RECAR one [32] where the over-abstraction would be to consider more and more complex shapes (points then rectangles then rectangles allowing holes, and so on).

Acknowledgments. The authors would like to thank the anonymous reviewers for their insightful comments. Part of this work was supported by the French Ministry for Higher Education and Research, the Haut-de-France Regional Council through the “Contrat de Plan État Région (CPER) DATA” and an EC FEDER grant.

References

1. Sioutis, M., Alirezaie, M., Renoux, J., Loufi, A.: Towards a synergy of qualitative spatio-temporal reasoning and smart environments for assisting the elderly at home. In: *IJCAI Workshop on Qualitative Reasoning (2017)*
2. Bhatt, M., Guesgen, H., Wölf, S., Hazarika, S.: Qualitative spatial and temporal reasoning: emerging applications, trends, and directions. *Spat. Cogn. Comput.* **11**, 1–14 (2011)
3. Dubba, K.S.R., Cohn, A.G., Hogg, D.C., Bhatt, M., Dylla, F.: Learning relational event models from video. *J. Artif. Intell. Res.* **53**, 41–90 (2015)
4. Story, P.A., Worboys, M.F.: A design support environment for spatio-temporal database applications. In: Frank, A.U., Kuhn, W. (eds.) *COSIT 1995*. LNCS, vol. 988, pp. 413–430. Springer, Heidelberg (1995). <https://doi.org/10.1007/3-540-60392-1-27>
5. Randell, D.A., Cui, Z., Cohn, A.: A spatial logic based on regions and connection. In: *KR (1992)*
6. Bouzy, B.: Les concepts spatiaux dans la programmation du go. *Revue d’Intelligence Artificielle* **15**, 143–172 (2001)
7. Lattner, A.D., Timm, I.J., Lorenz, M., Herzog, O.: Knowledge-based risk assessment for intelligent vehicles. In: *KIMAS (2005)*
8. Heintz, F., de Leng, D.: Spatio-temporal stream reasoning with incomplete spatial information. In: *ECAI (2014)*
9. Randell, D.A., Galton, A., Fouad, S., Mehanna, H., Landini, G.: Mereotopological correction of segmentation errors in histological imaging. *J. Imaging* **3**(4), 63 (2017)
10. Renz, J., Nebel, B.: On the complexity of qualitative spatial reasoning: a maximal tractable fragment of the region connection calculus. *Artif. Intell.* **108**(1–2), 69–123 (1999)

11. Li, S.: On topological consistency and realization. *Constraints* **11**, 31–51 (2006)
12. Renz, J., Nebel, B.: Qualitative spatial reasoning using constraint calculi. In: Aiello, M., Pratt-Hartmann, I., Van Benthem, J. (eds.) *Handbook of Spatial Logics*, pp. 161–215. Springer, Dordrecht (2007). https://doi.org/10.1007/978-1-4020-5587-4_4
13. Golumbic, M.C., Shamir, R.: Complexity and algorithms for reasoning about time: a graph-theoretic approach. *J. ACM* **40**, 1108–1133 (1993)
14. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* **50**(5), 752–794 (2003)
15. Huang, J., Li, J.J., Renz, J.: Decomposition and tractability in qualitative spatial and temporal reasoning. *Artif. Intell.* **195**, 140–164 (2013)
16. Brummayer, R., Biere, A.: Effective bit-width and under-approximation. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) *EUROCAST 2009*. LNCS, vol. 5717, pp. 304–311. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04772-5_40
17. Seipp, J., Helmert, M.: Counterexample-guided cartesian abstraction refinement. In: Borrajo, D., et al. (eds.) *Proceedings of ICAPS 2013*. AAAI (2013)
18. Soh, T., Le Berre, D., Roussel, S., Banbara, M., Tamura, N.: Incremental SAT-based method with native boolean cardinality handling for the hamiltonian cycle problem. In: Fermé, E., Leite, J. (eds.) *JELIA 2014*. LNCS (LNAI), vol. 8761, pp. 684–693. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11558-0_52
19. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. *Artif. Intell.* **234**, 1–24 (2016)
20. Pulina, L.: The ninth QBF solvers evaluation - preliminary report. In: Lonsing, F., Seidl, M. (eds.) *Proceedings of QBF@SAT 2016*, CEUR Workshop Proceedings, vol. 1719. CEUR-WS.org (2016)
21. Hooker, J.N.: Logic-based methods for optimization. In: Borning, A. (ed.) *PPCP 1994*. LNCS, vol. 874, pp. 336–349. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58601-6_111
22. Chu, Y., Xia, Q.: A hybrid algorithm for a class of resource constrained scheduling problems. In: Barták, R., Milano, M. (eds.) *CPAIOR 2005*. LNCS, vol. 3524, pp. 110–124. Springer, Heidelberg (2005). https://doi.org/10.1007/11493853_10
23. Hooker, J.N.: A hybrid method for the planning and scheduling. *Constraints* **10**(4), 385–401 (2005)
24. Tran, T.T., Beck, J.C.: Logic-based benders decomposition for alternative resource scheduling with sequence dependent setups. In: *Proceedings of ECAI 2012* (2012)
25. de Moura, L., Rueß, H., Sorea, M.: Lazy theorem proving for bounded model checking over infinite domains. In: Voronkov, A. (ed.) *CADE 2002*. LNCS (LNAI), vol. 2392, pp. 438–455. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45620-1_35
26. Ji, X., Ma, F.: An efficient lazy SMT solver for nonlinear numerical constraints. In: *Proceedings of WETICE 2012* (2012)
27. Renz, J.: A canonical model of the region connection calculus. *JANCL* **12**, 469–494 (2002)
28. Renz, J., Ligozat, G.: Weak composition for qualitative spatial and temporal reasoning. In: van Beek, P. (ed.) *CP 2005*. LNCS, vol. 3709, pp. 534–548. Springer, Heidelberg (2005). https://doi.org/10.1007/11564751_40
29. Li, S., Ying, M.: Region connection calculus: its models and composition table. *Artif. Intell.* **145**, 121–146 (2003)
30. Long, Z., Schockaert, S., Li, S.: Encoding large RCC8 scenarios using rectangular pseudo-solutions. In: *Proceedings of KR 2016* (2016)

31. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Siekmann, J.H., Wrightson, G. (eds.) *Automation of Reasoning*, pp. 466–483. Springer, Heidelberg (1983). https://doi.org/10.1007/978-3-642-81955-1_28
32. Lagniez, J.M., Le Berre, D., de Lima, T., Montmirail, V.: A recursive shortcut for CEGAR: application to the modal logic K satisfiability problem. In: *Proceedings of IJCAI 2017* (2017)
33. Long, Z.: *Qualitative spatial and temporal representation and reasoning: efficiency in time and space*. Ph.D. thesis, Faculty of Engineering and Information Technology, University of Technology Sydney (UTS), January 2017
34. Savicky, P., Vomlel, J.: Triangulation heuristics for BN2O networks. In: Sossai, C., Chemello, G. (eds.) *ECSQARU 2009*. LNCS (LNAI), vol. 5590, pp. 566–577. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02906-6_49
35. Sioutis, M., Koubarakis, M.: Consistency of chordal RCC-8 networks. In: *Proceedings of ICTAI 2012*. IEEE Computer Society (2012)
36. Dechter, R., Meiri, I., Pearl, J.: Temporal constraint networks. *Artif. Intell.* **49**(1–3), 61–95 (1991)
37. Long, Z., Sioutis, M., Li, S.: Efficient path consistency algorithm for large qualitative constraint networks. In: *Proceedings of IJCAI 2016* (2016)
38. Sioutis, M., Long, Z., Li, S.: Leveraging variable elimination for efficiently reasoning about qualitative constraints. *Int. J. Artif. Intell. Tools* (2018, in press)
39. Audemard, G., Lagniez, J.-M., Simon, L.: Improving glucose for incremental SAT solving with assumptions: application to MUS extraction. In: Järvisalo, M., Van Gelder, A. (eds.) *SAT 2013*. LNCS, vol. 7962, pp. 309–317. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39071-5_23
40. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) *SAT 2003*. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
41. Westphal, M., Wölfl, S., Gantner, Z.: GQR: a fast solver for binary qualitative constraint networks. In: *Proceedings of the AAI Spring Symposium*. AAI (2009)
42. Renz, J., Nebel, B.: Efficient methods for qualitative spatial reasoning. *J. Artif. Intell. Res.* **15**, 289–318 (2001)
43. Sioutis, M., Condotta, J.-F.: Tackling large qualitative spatial networks of scale-free-like structure. In: Likas, A., Blekas, K., Kalles, D. (eds.) *SETN 2014*. LNCS (LNAI), vol. 8445, pp. 178–191. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07064-3_15
44. Sioutis, M., Condotta, J., Koubarakis, M.: An efficient approach for tackling large real world qualitative spatial networks. *Int. J. Artif. Intell. Tools* **25**, 1–33 (2016)
45. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. *Science* **286**(5439), 509–512 (1999)



Clause Learning and New Bounds for Graph Coloring

Emmanuel Hebrard¹(✉) and George Katsirelos²(✉)

¹ LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France
hebrard@laas.fr

² MIAT, UR-875, INRA, Toulouse, France
gkatsi@gmail.com

Abstract. Graph coloring is a major component of numerous allocation and scheduling problems.

We introduce a hybrid CP/SAT approach to graph coloring based on exploring Zykov’s tree: for two non-neighbors, either they take a different color and there might as well be an edge between them, or they take the same color and we might as well merge them. Branching on whether two neighbors get the same color yields a symmetry-free tree with complete graphs as leaves, which correspond to colorings of the original graph.

We introduce a new lower bound for this problem based on Mycielskian graphs; a method to produce a clausal explanation of this bound for use in a CDCL algorithm; and a branching heuristic emulating Brelaz on the Zykov tree.

The combination of these techniques in both a branch-and-bound and in a bottom-up search outperforms Dsatur and other SAT-based approaches on standard benchmarks both for finding upper bounds and for proving lower bounds.

1 Introduction

A *coloring* of a graph is a labeling of its vertices such that adjacent vertices have distinct labels. Let a labeling of the graph $G = (V, E)$ be a mapping from its set of vertices V to the integers. A labeling c such that $c(v) \neq c(u)$ for every edge $(uv) \in E$ is a coloring of G , and its cardinality is $|\{c(v) \mid v \in V\}|$. The *chromatic number* $\chi(G)$ of a graph G is the cardinality of its smallest coloring.

The problem of finding a minimum coloring of a graph is NP-hard, but has numerous applications. For instance when allocating frequencies, devices on nearby locations should not be assigned the same frequency to avoid interferences. The chromatic number of this distance-induced graph is therefore the minimum span of frequencies that is required [1, 18]. In compilers, finding an optimal register allocation can be cast as a coloring problem on an *interference* graph of value live ranges [3].

G. Katsirelos—Partially supported by the french “Agence nationale de la Recherche”, project DEMOGRAPH, reference ANR-16-C40-0028.

One of the oldest and most successful technique for coloring a graph is Brelaz' *Dsatur* algorithm [2]: when branching, the vertex with highest *degree of saturation* is chosen and colored with the lexicographically least candidate. The degree of saturation of a vertex v is the number of assigned colors within its neighborhood $N_G(v)$ in G . In case of a tie, the vertex with largest number of uncolored neighbors is chosen among the tied vertices. This heuristic is often used within a branch-and-bound algorithm with one variable per vertex whose domain is the set of possible colors. It is known as **dom+deg** in the CSP literature [6]. The standard approach for computing a bound in these algorithms is to compute a heuristic approximation of the clique number $\omega(G)$ of the graph G (e.g., the size of a maximal clique) since $\omega(G) \leq \chi(G)$. This bound is known to be weak for some polynomially recognizable classes of graphs, such as Mycielskian graphs, which are triangle-free graphs with arbitrarily large chromatic number [16]. Moreover, within the search tree explored using Brelaz' heuristic, the clique has to be found only among vertices with degree of saturation equal to the number of colors in the current partial solution (i.e., adjacent to at least one vertex of every color used so far). Finally, this formulation exhibits value interchangeability [24]. One common way to break this symmetry is to arbitrarily color a clique, and never branch on colors larger than $k + 1$ when extending a solution with k colors [14, 23, 25].

Satisfiability [13, 15] offers an attractive approach to coloring, in part because it is trivial to encode the problem. In satisfiability, we express problems with Boolean variables \mathbf{X} . We say that a literal l is either a variable x or its negation \bar{x} . Constraints are disjunctions of literals, written interchangeably as sets of literals or as disjunctions, which are satisfied by an assignment if it assigns at least one literal to true. In order to encode graph coloring with satisfiability, one typically relies on *color* variables x_{vi} , where x_{vi} being true means vertex v takes color i . For every edge (uv) , there is a binary clause $\bar{x}_{vi} \vee \bar{x}_{ui}$ for every color i . Then, if K is the maximum number of colors, then there is a clause $\bigvee_{1 \leq i \leq K} x_{ui}$. Refinements to this encoding include Van Gelder's log encoding versions, where x_{vj} is true if the j -th bit of the binary encoding of the color taken by vertex v is 1 [22]. However, the use of modern SAT solving techniques like restarting [7, 8] and clause learning [13] are not straightforward to combine with symmetry breaking such as that of van Hentenryck et al. [23]. They can only be easily combined with starting from an arbitrary coloring to a clique, but that is incomplete. The `color6` solver [25] uses symmetry breaking branching but forgoes restarting to maintain complete symmetry breaking.

On the other hand, the search tree induced by Zykov's deletion-contraction recurrence [26] has no color symmetry and using the clique number as lower bound is easier and more powerful than in the color variable formulation.

Let $G/(uv)$ be the graph obtained by *contracting* u and v : the two vertices are identified to a single vertex $r(u) = r(v) = u$, every edge (vw) is replaced by $(r(v)w)$ and self edges are discarded. Conversely, let $G + (uv)$ be the graph obtained by adding the edge (uv) . The Zykov recurrence is thus:

$$\chi(G) = \min\{\chi(G/(uv)), \chi(G + (uv))\} \quad (1)$$

Indeed, given a minimum coloring of G , either the vertices u and v have distinct colors and therefore it is also a coloring of $G + (uv)$, or they have the same color and it is a coloring of $G/(uv)$.

Example 1. Fig. 1 illustrates the Zykov recurrence. From the graph G in Fig. 1a, we obtain the graph $G + (cd)$ shown in Fig. 1b by adding the edge (cd) and the graph $G/(cd)$ shown in Fig. 1c. One of these two graphs has the same chromatic number as G .

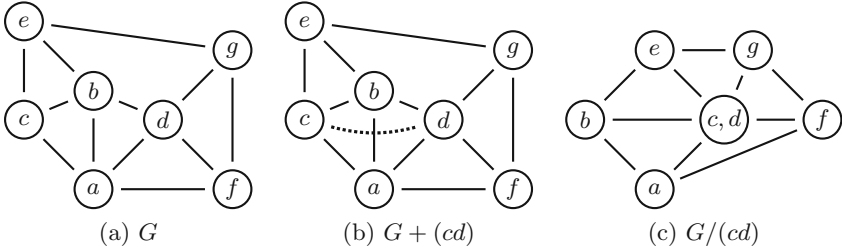


Fig. 1. Zykov recurrence

This branching scheme was successfully used in a branch-and-price approach to coloring [14]. In the context of satisfiability, Schaafsma et al. showed that a clause encoding of Zykov formulation is not efficient [19]. For every non-edge (uv) , the edge variable e_{uv} stands for the decision of contracting the vertices ($e_{uv} = 1$), or adding the edge ($e_{uv} = 0$). A difficulty is that a cubic number of clauses are required, three for every triplet u, v, w , in order to forbid that exactly two of the variables e_{uv}, e_{uw} and e_{vw} are true. This encoding proved too heavy and as a result less efficient than the formulations using color variables. However, Schaafsma et al. introduced a novel and clever way of taking advantage of Zykov’s idea: when learning a clause involving color variables, one can compactly encode all symmetric clauses using a single clause that only uses edge variables and propagates the same as if all the symmetric clauses were present.

In this work, we propose a constraint programming formulation of coloring in Sect. 2 that also uses the Zykov branching scheme. We use the idea of integrating constraint programming into clause learning satisfiability solvers by simply having each propagator label each pruning or failure by a clausal *reason* or *explanation* [10,17] to alleviate the cost of keeping the edge variables consistent (Sect. 2.1) and to integrate a lower bound based on either cliques (Sect. 2.2) or a more general bound based on Mycielskians (Sect. 2.3). Together with effective branching heuristics (Sect. 2.4) and search strategies that emphasize either upper or lower bounds (Sect. 2.5), we get a solver that clearly outperforms the state of the art in satisfiability-based coloring (Sect. 3).

2 Clause-Learning Approach

In our approach, similar to that Schaafsma et al., we use a model which leads to the exploration of the tree resulting from application of the Zykov recurrence. We have one Boolean variable e_{uv} for each non-edge of the input graph, that is for every $(uv) \notin E$. When e_{uv} is true, the vertices v and u are contracted, hence assigned the same color, and are separated otherwise, hence assigned different colors. We somewhat abuse notation in the sequel and write clauses using variables e_{uv} even when $(uv) \in E$ and assume that the variable is set to false at the root of the search tree.

With every (partial) assignment A to the edge variables, we can associate a graph G_A . For the empty assignment, we have $G_\emptyset = G$. For non-empty assignments it is the graph that results from contracting all vertices u, v of G for which A contains e_{uv} and adding an edge between all pairs of vertices u, v of G for which A contains \bar{e}_{uv} . When e_{uv} and e_{vw} are both true, this means that we contract u and v and then contract w and $r(v)$ and similarly for false literals. The operation of contracting vertices is associative and commutative, so we get the same graph G_A regardless of the order in which we process the literals in A .

The property of having the same color is transitive, so if e_{uv} and e_{vw} are true, then so is e_{uw} . Similarly, if e_{uv} is true and e_{vw} is false, then e_{uw} must also be false. We enforce this using the constraint

$$\text{TRIANGLE}(\{e_{uv} \mid (uv) \notin E\}) \quad (2)$$

We can enforce GAC on this constraint using a decomposition of size $O(|V|^3)$:

$$(\bar{e}_{uv} \vee \bar{e}_{vw} \vee e_{uw}) \quad \forall \text{ distinct } u, v, w \in V \quad (3)$$

Enforcing unit propagation on this decomposition therefore takes $O(|V|^3)$ time, amortized over a branch of the search tree. In our implementation, we have opted instead for a dedicated propagator for this, described in Sect. 2.1, whose complexity over a branch is only $O(|V|^2)$.

The model also includes a constraint COLORING which is satisfied by any assignment that corresponds to a coloring with fewer than k colors.

$$\text{COLORING}(\{e_{uv} \mid (uv) \notin E\}, k) \quad (4)$$

This constraint is clearly NP-hard. We describe two incomplete propagators for it in Sects. 2.2 and 2.3. The first computes either the well known clique lower bound (Sect. 2.2) and the second a novel, stronger, bound (Sect. 2.3). If that bound meets or exceeds k , the propagator fails and produces an explanation. Neither of these bounds is cheap to compute, hence the propagator runs at a lower priority than unit propagation and the TRIANGLE constraint.

Although we have experimented with pruning in this propagator, the rules we have found tend to be ineffective, in the sense that they generate very little pruning, barely reduce the overall search effort, and increase the overall runtime.

Discussion. Clearly, our approach is closely related to that of Schaafsma et al. However, there are some important differences. First, since we do not need the color variables to compute the size of the coloring, we completely eliminate the need for the clause rewriting scheme that they implement and get color symmetry-free search with no additional effort. In addition, since we our model uses a CP/SAT hybrid, we can use a constraint to compute a lower bound at each node, thus avoiding a potentially large number of conflicts.

The approach of Schaafsma et al. does not enforce triangle consistency except through the color variables (so that $X_v = X_u \iff e_{uv}$). In contrast, the triangle propagator maintains GAC on this constraint, without having to encode channeling between the edge variables and the color variables.

The main drawback of this model is that we need a large number of variables. This is especially problematic for large, sparse graphs, where the number of non-edges is quadratic in the number of vertices and significantly larger than the number of edges. Indeed, in 4 of the 125 instances we used in our experimental evaluation, our solver exceeded the memory limit.

The approach of Schaafsma et al. does not have the same limitation, as they introduce variables only when they are needed to rewrite a learnt clause, in a way similar to lazy model expansion [4]. It is possible that this approach of lazily introducing variables can be adapted to our model, but this, as well as other ways of reducing the memory requirements, remains future work.

2.1 Triangle Consistency Propagation

The propagator for the TRIANGLE constraints works as follows: for each vertex v , we keep a bag $b(v)$ to which it belongs. Initially, $b(v) = \{v\}$ for all v . When we set e_{uv} to true, we set $e_{u'v'}$ to true for all $v' \in b(v), u' \in b(u)$. We also set $e_{u'v'}$ to false for all $v' \in b(v)$ and $u' \in N(b(u) \setminus N(b(v)))$.¹ Finally, we set $B = b(u) \cup b(v)$ and update $b(v') = B$ for all $v' \in B$. In the case where we set e_{uv} to false, we set $e_{u'v'}$ to false for all $v' \in b(v), u' \in b(u)$.

A small but important optimization is that if the propagator is invoked for e_{uv} becoming true (resp. false) but u and v are already in the same bag (resp. already separated) then it does nothing. This ensures that it touches each non-edge exactly once, hence its complexity is quadratic over an entire branch. This is also optimal, since in the worst case every non-edge must be set either as a decision or by propagation.

This propagator uses the clauses (3) as explanations. The mapping from actions that it performs to explanations is fairly straightforward, using the vertices involved in the literal that woke the propagator as “pivots”. For example, if $b(v) = \{v, v'\}$, $b(u) = \{u, u'\}$ and it is woken on the literal e_{uv} , it sets $e_{uv'}$ using $(\bar{e}_{vv'} \vee \bar{e}_{uv} \vee e_{uv'})$ as the reason and then $e_{u'v'}$ using $(\bar{e}_{uv'} \vee \bar{e}_{uu'} \vee e_{u'v'})$.

¹ We abuse the neighborhood notation and write $N(S)$ for $\bigcup_{u \in S} N(u)$.

2.2 Clique-Based Lower Bound

As we already discussed, an important advantage of the edge-variable based model is that computing a lower bound for the current subproblem is as easy as for the entire problem. For example, if the partial assignment in the current node is A , the clique number of the graph G_A is a lower bound for the subproblem.

In order to find a large clique we use the following greedy algorithm. Let o be an ordering of the vertices, so we visit all vertices in the order v_{o_1}, \dots, v_{o_n} . We maintain an initially empty list of cliques. For each vertex, we add to all the cliques which admit it and if no clique admits it we put it in a new singleton clique. When this finishes, we iterate over the vertices one more time and add them to all cliques which admit them, because in the first pass a vertex v was not evaluated against cliques which were created after we processed v . We then pick the largest among these cliques as our lower bound.

If the lower bound meets or exceeds the upper bound k , the propagator reports a conflict. We construct a clausal conflict as follows: each vertex v of the current graph is the result of the contraction of 1 or more vertices of the original graph. In keeping with the notation for the triangle consistency propagator, we call this the *bag* $b(v)$. We arbitrarily pick one vertex $r(v)$ from the bag of each vertex v in the largest clique C , and set the explanation to

$$\bigvee_{v,u \in C} e_{r(v)r(u)} \quad (5)$$

We have experimented with producing explanations with mixed-sign literals and found that they tend to be much shorter and speed up search in terms of number of conflicts per second, but significantly increase the overall effort required, both in runtime and number of conflicts.

Preprocessing and Vertex Ordering. We tried a few different heuristics for ordering the vertices of the graph, including the inverse of the degeneracy order [11], which tends to produce large cliques [5,9]. However, we found that it works best to sort the vertices in order of decreasing bag size.

Lin et al. [12] recently proposed a reduction rule for graph coloring instances, which allowed them to reduce the size of large, sparse graphs.

Proposition 1 ([12]). *Let G be a graph with $\chi(G) \geq k$ and let I be an independent set of G such that for all $v \in I$, $d(v) \leq k$. Then, $k - 1 \leq \chi(G \setminus I) \leq \chi(G)$ and if $\chi(G \setminus I) = k - 1$ then $\chi(G) = k$.*

The rule of Proposition 1 can be used with any lower bound and applied iteratively until no more reduction is possible. Besides the obvious advantage of trimming the graph this reduction also helps improve the lower bound found by a heuristic maximal clique algorithm. The reason is that whatever heuristic we use for finding a maximal clique may make a suboptimal choice and this preprocessing step removes some obviously suboptimal choices from consideration.

We have used this result for preprocessing, as Lin et al. did, but observed very little benefit in our instance set, which comprises smaller and denser graphs

than the one that they used. We also used it, however, to improve the ordering for the greedy algorithm by placing such vertices at the end of the ordering. As we will show in Sect. 3, this has a small but measurable impact.

2.3 Mycielski-Based Bound

Although being a useful bound in practice, the clique number is both hard to compute and gives no guarantees on the quality of the bound. We propose here a new lower bound inspired by *Mycielskian graph*.

Definition 1 (Mycielskian graph [16]). The Mycielskian graph $\mu(G) = (\mu(V), \mu(E))$ of $G = (V, E)$ is defined as follows:

- $\mu(V)$ contains every vertex in V , and $|V| + 1$ additional vertices, constituted of a set $U = \{u_i \mid v_i \in V\}$ and another distinct vertex w .
- For every edge $v_i v_j \in E$, $\mu(E)$ contains $v_i v_j, v_i u_j$ and $u_i v_j$. Moreover, it contains all the edges between U and w .

The Mycielskian $\mu(G)$ of a graph G , has the same clique number, however its chromatic number is $\chi(G) + 1$. Indeed, consider a coloring of $\mu(G)$. For any vertex $v_i \in V$, we have $N(v_i) \subseteq N(u_i)$, and therefore we can safely use the same color v_i as for u_i . It follows that at least $\chi(G)$ colors are required for the vertices in U , and since $N(w) = U$, then w requires a $\chi(G) + 1$ -th color. Mycielski introduced these graphs to demonstrate that triangle-free graphs can have arbitrarily large chromatic numbers, hence the clique number does not approximate the chromatic number.

The principle of our bound is a greedy procedure that can discover embedded “pseudo” Mycielskians. Indeed, the class of embedded graphs that we look for is significantly broader than set of “pure” Mycielskians $\{M_2, M_3, M_4, \dots\}$. First, we look for a partial subgraph. Therefore, trivially, Mycielskians with extra edges also provide valid lower bounds. Moreover, we use as starting point a (potentially large) clique. Finally, the method we propose can also find Mycielskians

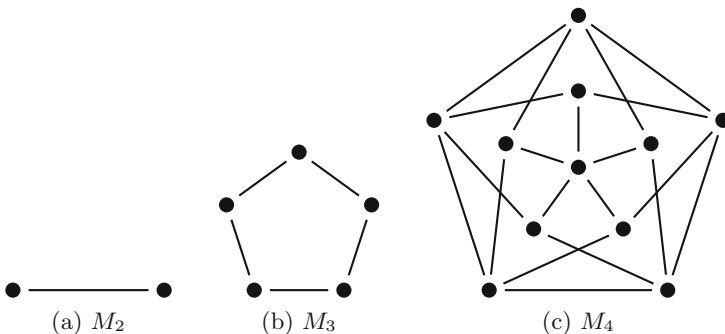


Fig. 2. A 2-clique $M_2 = \mu(\emptyset)$, its Mycielskians $M_3 = \mu(M_2)$ and $M_4 = \mu(M_3)$

modulo some vertex contractions. Clearly, those are also valid lower bounds since contracting vertices is equivalent to adding equality constraints to the problem.

Let $N_G(v)$ be the neighborhood of v in the graph G . Suppose that we have a partial subgraph $H = (V_H, E_H)$ of G such that $\chi(H) \geq k$. This can be for example a clique of size k . We define

$$S_v = \{u \mid N_H(v) \subseteq N_G(u)\} \tag{6}$$

Suppose that there exists a vertex w with at least one neighbor in every set S_v :

$$w \in \bigcap_{v \in V_H} N_G(S_v) \tag{7}$$

and let $u(v)$ be any element of S_v such that $u(v) \in N_G(w)$ and $U = \{u(v) \mid v \in V\}$, then:

Lemma 1. *The graph*

$$H' = (V \cup U \cup \{w\}, E \cup \bigcup_{v \in V} N_H(v) \times u(v) \cup \bigcup_{u \in U} \{(u, w)\})$$

is such that $\chi(H') \geq k + 1$.

Proof. The proof follows from the facts that H' is the Mycielskian graph of H possibly with contracted vertices, and is embedded in G .

Suppose first that, for each $v \in V$, $u(v) \neq v$ and $w \notin V$. Then we have $H' = \mu(H)$ by using $u(v_i)$ for the vertex u_i , and w for itself, in Definition 1.

Suppose now that $H' \neq \mu(H)$. This can only be because either:

- For some vertex v_i of H , we have $u(v_i) = v_i$. In this case, consider the graph $\mu(H)$ and contract u_i and v_i . The resulting graph $\mu(H)/(u_i v_i)$ has a chromatic number at least as high as $\mu(H)$. However, it is isomorphic to H' .
- The vertex w is the vertex v_i from the original subgraph H . Here again contracting v_i and w in $\mu(H)$ yields H' .

Notice that there is not a third case where w is taken among U since, for any $v \in V_H$, we have $u(v) \notin \bigcap_{v \in V_H} N_G(S_v)$ because $u(v)$ is not a neighbor of itself.

Finally, it is easy to see that H' is embedded in G since the edges added to H' are all edges of G . □

Example 2. Figure 3a shows the graph $G/(cd)$ obtained by contracting vertices c and d in the graph G of Fig. 1. Let H be the clique $\{a, b, c\}$. We have $S_a = \{a, e\}$, $S_b = \{b, f\}$ and $S_c = \{c\}$. Furthermore, $N_G(\{a, e\}) \cap N_G(\{b, f\}) \cap N_G(\{c\}) = \{b, c, g\} \cap \{a, e, c, g\} \cap \{a, b, e, g, f\} = \{g\}$, from which we can conclude that this graph has chromatic number at least 4. As shown in Fig. 3b, when called with $H = \{a, b, c\}$ Algorithm 1 will extend it with a first layer $U = \{e, c, f\}$ and an extra vertex $w = g$. Notice that the graph obtained by adding the edge (cd) has a 4-clique (see Fig. 1). Therefore, the graph G in Fig. 1a also has a chromatic number of at least 4.

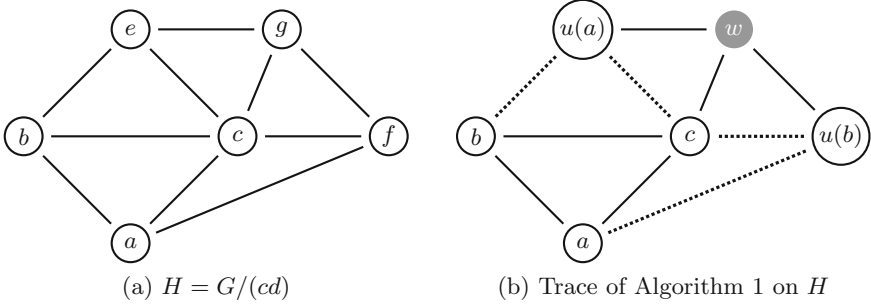


Fig. 3. Embedded Mycielski

Algorithm 1 greedily extends a partial subgraph $H = (V_H, E_H)$ of the graph G (with $\chi(H) \geq k$) into a larger partial subgraph $H' = (V'_H, E'_H)$, following the above principles. As long as this succeeds, in the outermost loop, we replace H by H' and iterate. The computed bound k is equal to $\chi(H)$ plus the number of successful iterations.

We compute the sets S_v (Eq. 6) and the set W of nodes with at least one neighbor in every S_v in Loop 1. Then, if it is possible to extend H (Line 4), we compute the pseudo Mycielskian (V'_H, E'_H) as shown in Lemma 1 and replaces H with it in Line 5 before starting another iteration.

Complexity. One iteration of Algorithm 1 requires $O(|V_H| \times |V|)$ bitset operations (Line 2 is 1 ‘AND’ operation and Line 3 is $O(|V|)$ ‘OR’ operations and 1

Algorithm 1. MYCIELSKIBOUND($k, H = (V_H, E_H), G = (V, E)$)

```

while  $|V_H| < |V|$  do
     $W \leftarrow V$ ;
     $\forall v \in V_H \ S_v \leftarrow \{v\}$ ;
1   foreach  $v \in V_H$  do
    |   foreach  $u \in V$  do
2   |   |   if  $N_H(v) \subseteq N_G(u)$  then  $S_v \leftarrow S_v \cup \{u\}$ 
3   |   |    $W \leftarrow W \cap N_G(S_v)$ ;
4   |   if  $W \neq \emptyset$  then
    |   |    $k \leftarrow k + 1$ ;
    |   |    $(V'_H, E'_H) \leftarrow (V_H, E_H)$ ;
    |   |    $w \leftarrow$  any element of  $W$ ;
    |   |   foreach  $v \in V_H$  do
    |   |   |    $V'_H \leftarrow V'_H \cup \{\text{any element of } (N_G(w) \cap S_v)\}$ ;
    |   |   |    $E'_H \leftarrow E'_H \cup \{(wu)\} \cup \{u \times N_H(v)\}$ ;
5   |   |    $(V_H, E_H) \leftarrow (V'_H, E'_H)$ ;
    |   |   else break
    |   |   return  $k$ ;

```

‘AND’). The second part of the loop, starting from Line 4, runs in $O(|V_H|^2)$ time. Typically, the number of iterations is very small. In the worst case, it cannot be larger than $\log |V|$ since the number of vertices in H is (more than) doubled at each iteration. It follows that Loop 1 is executed at most $2|V|$ times, and therefore, the worst case time complexity is $O(|V|^2)$ bitset operations (hence $O(|V|^3)$ time).

Explanation. Similarly to the clique based lower bound, the explanations that we produce here correspond to the set of all edges in the graph H :

$$\bigvee_{(v,u) \in E_H} e_{uv} \quad (8)$$

Adaptive Application of the Mycielskian Bound. In our experiments, we found that trying to find a Mycielskian subgraph in every node of the search tree was too expensive and did not pay off in terms of total runtime. Therefore, we adapted a heuristic proposed by Stergiou [20] which allows us to apply this stronger reasoning less often. In particular, we only compute the clique lower bound by default. But everytime there is a conflict, whether by unit propagation or by bound computation, we compute the Mycielskian lower bound in the next node. If that causes a conflict, we keep computing this bound until we backtrack to a point where even the stronger bound does not detect a bound violation. This has the effect that we compute the cheaper clique lower bound most of the time, but learn clauses based on the stronger bound.

2.4 Branching Heuristic

Brelaz’ branching heuristic remains extremely competitive for finding good colorings, as evidenced by the performance of `Dsatur` in our experimental evaluation (Sect. 3). Moreover, Schaafsma et al. observed that branching on color variables was significantly better than branching on edge variables.

But adding the color variables is not really desirable. First, it adds the overhead of propagating the reified equality constraints. Second, using these variables to follow the Brelaz heuristic requires branching on them, which in turn requires that we use some kind of symmetry breaking method, like the rewriting scheme of Schaafsma et al. So it would be preferable to get the benefit of the more effective branching heuristic without needing to introduce color variables.

In order to get behavior similar to that of Brelaz’ heuristic in the edge variable model, we proceed as follows: we pick a maximal clique C in the current graph. We pick the vertex v that maximises $|N(v) \cap C|$, breaking ties by highest $|N(v) \setminus C|$, and an arbitrary vertex $u \in C \setminus N(v)$ ². We then set e_{uv} to true. This uses the current maximal clique to implicitly construct a coloring and uses that to choose the next vertex to color as Brelaz’ heuristic does. If the assignments e_{uv} are refuted for all $u \in C$, then v is adjacent to all vertices in C and so $C \cup \{v\}$ is a larger clique, which corresponds to using a new color in Brelaz’ heuristic.

² We assume the graph is connected, otherwise u may not always exist.

This branching strategy can be more flexible than committing to a coloring by assigning the color variables. For example, unit propagation on learned clauses as we explore a branch of the search tree can make it so that the maximal clique C' at some deep level is not an extension of the maximal clique C at the root of the tree, i.e., $C \not\subseteq C'$. The Brelaz heuristic on the color variables commits to using C at the root, hence cannot take advantage of the information that C' is a larger clique. The modification that we present here achieves this.

2.5 Solution Strategies

Previous satisfiability-based approaches to coloring have mostly ignored the optimization problem of finding an optimal coloring of a graph and instead attack the decision problem of whether a graph is colorable with k colors. In our setting, we have the flexibility to do both. In particular, we implemented two search strategies: branch-and-bound and bottom-up. The former uses a single instance of a solver, finds a solution and then tightens the upper bound in the COLORING constraint and continues searching. This is similar to the top-down approach one would use when solving a series of decision problems, starting from a heuristic upper bound and decreasing that until we generate an unsatisfiable instance, in which case we have identified the optimum. The advantage of the branch-and-bound approach is that it does not discard accumulated information between solution: learned clauses and heuristic scores for variables. Moreover, it more closely resembles the typical approach used in constraint programming systems.

The other approach we implemented is bottom-up: start from a lower bound (such as those described in Sect. 2.2 or 2.3) and keep increasing until we find a satisfiable instance, which gives the optimum. This has none of the advantages of the branch-and-bound approach, as it is not safe to reuse clauses from a more constrained problem in one that is less constrained. Moreover, it cannot generate upper bounds before it finds the optimum. But it gains from the fact that the more constrained problems it solves may be easier. One particular behavior we have observed is that sometimes the lower bound computed at the root coincides with the optimum and finding is quite easy with the bottom-up strategy, but finding that solution with branch-and-bound can be very hard.

3 Experimental Evaluation

We implemented several variations of our approach using MINICSP³ as the underlying CDCL CSP solver, and retained two, one for each of the solution strategies described in Sect. 2.5.⁴ The former, `cdc1`, is a branch-and-bound algorithm, using Brelaz branching. The latter, `cdc1↑`, is a bottom-up algorithm, using VSIDS. In both cases, we use the adaptive application of the Mycielskian bound, as explained in Sect. 2.3. When computing the Mycielskian bound, we apply Algorithm 1 on all of the maximal cliques, and keep the best outcome.

³ Sources available at: <https://bitbucket.org/gkatsi/minicsp>.

⁴ Sources available at: <https://bitbucket.org/gkatsi/gc-cdcl/src/master/>.

We compared with the state-of-art SAT-based solver `color6` [25], a very efficient clause-learning algorithm for graph coloring proposed recently by Zhou et al. Similarly to our approach, it is based on a SAT solver (namely `zChaff`), however, it uses the color-based formulation. It was shown to outperform the state of the art on many instances. As `color6` solves satisfiability instances only (testing whether a coloring with a specific number of colors exists), we implemented a branch-and-bound wrapper on top of it, denoted `color6`, as well as a wrapper that implements the bottom-up strategy, denoted `color6↑`. We used the lower and upper bounds computed by our approach (respectively the maximal clique algorithm described in Sect. 2.2 and a greedy run of Brelaz) as initial bounds for `color6` and `color6↑`.

Moreover, we also compared with an implementation of `Dsatur` by Trick, and an integer programming formulation in `CPLEX`. The model we used for `CPLEX` is the trivial one using binary color variables (one for each vertex and each color), and one binary inequality per edge. However, observe that `CPLEX` actually computes maximal cliques in its preprocessing, so providing it with clique inequalities would have been useless. Moreover, we initialized the upper bound with the same method as for `color6`, and also arbitrarily fixed the colors of one maximal clique in order to break symmetries.

Unfortunately, we could not compare our method to the method of Schaafsma et al. (`Minicolor`) directly. Indeed, its implementation, generously provided by the authors, is difficult to use in the type of extensive experiments of the type we performed. Firstly, the algorithm is restricted to instances with at most 32 colors. Secondly it solves the satisfiability problem $\chi(G) \leq K$ and uses a file converter. Finally, the changes made to `Minisat`'s code do not seem to be robust and we experienced several occurrences of assertion failures.

We used 125 benchmark instances from Trick's graph coloring webpage (<http://mat.gsia.cmu.edu/COLOR/color.html>) and described in the proceedings of the workshop COLOR02 [21]. In the subsequent tables, however, we omit 22 of these instances that were trivial for every approach we used (i.e., that solved by every method to optimality). Every method was run with a time limit of one hour and a memory limit of 3.5 GB⁵ on 4 nodes, each with 35 Intel Xeon CPU E5-2695 v4 2.10 GHz cores running Linux Ubuntu 16.04.4.

The results in Tables 1 and 2 are averaged over instances from the same class and the number of instances in each class is given next to the class name. We show the ratio of instances for which a proof of optimality was found ('opt'), as well as the average upper bound ('ub') and lower bound ('lb'), for every method. The best results for each criterion are highlighted using colors. Table 1 focuses on top-down methods. `cdc1` is better on all but three classes of instances: `Insertions`, `qg` (quasigroup) and `queen`. Moreover, it finds the same coloring as the other methods in the `Insertions` class, and computes strictly more proofs of optimality than other solvers in the two other classes. Finally, on many classes it is strictly better than the second best solver (considering at least one criterion). Table 2 focuses on the two bottom-up methods. Here again there are far more

⁵ `cdc1` exceeded the memory limit on 4 instances, and `CPLEX` on 16 instances.

Table 1. Comparison with top-down methods (by classes of instances)

		cdcl			color6			CPLEX			Dsaturn		
		opt	ub	lb	opt	ub	lb	opt	ub	lb	opt	ub	lb
DSJ	(14)	0.07	76.00	30.71	0.07	77.57	28.93	0.07	86.07	29.79	0.00	77.86	27.64
FullIns	(14)	1.00	6.79	6.79	0.21	6.79	5.14	0.86	6.93	6.36	0.00	6.79	4.86
Insertions	(11)	0.27	5.18	2.55	0.36	5.18	2.82	0.36	5.18	3.64	0.00	5.18	2.00
abb313GPI	(1)	1.00	9.00	9.00	0.00	14.00	8.00	0.00	14.00	8.00	0.00	10.00	6.00
ash	(3)	1.00	4.00	4.00	0.67	4.67	3.67	0.33	5.67	3.33	0.00	4.33	3.00
flat	(6)	0.00	73.83	11.67	0.00	74.33	10.67	0.00	79.67	10.67	0.00	74.83	9.67
fpsol2	(1)	1.00	65.00	65.00	0.00	65.00	59.00	1.00	65.00	65.00	1.00	65.00	65.00
inithx	(1)	1.00	54.00	54.00	0.00	54.00	43.00	1.00	54.00	54.00	1.00	54.00	54.00
latin_square	(1)	0.00	116.00	90.00	0.00	125.00	90.00	0.00	159.00	90.00	0.00	129.00	90.00
le450	(10)	0.50	15.20	13.00	0.10	15.60	13.00	0.30	19.10	13.00	0.20	16.00	11.70
miles	(5)	1.00	34.80	34.80	0.00	36.40	33.40	1.00	34.80	34.80	1.00	34.80	34.80
mug	(4)	1.00	4.00	4.00	1.00	4.00	4.00	1.00	4.00	4.00	0.00	4.00	3.00
myciel	(5)	1.00	6.00	6.00	0.80	6.00	4.80	0.60	6.00	5.00	0.00	6.00	2.00
qg	(4)	0.75	66.00	57.50	0.25	63.25	57.50	0.25	72.50	57.50	0.25	59.50	57.50
queen	(13)	0.46	12.08	10.85	0.00	15.92	10.62	0.38	12.46	10.77	0.23	12.00	10.62
school1	(1)	1.00	14.00	14.00	0.00	26.00	14.00	1.00	14.00	14.00	0.00	14.00	13.00
wap0	(8)	0.12	46.50	41.25	0.00	47.62	40.00	0.00	51.12	40.00	0.00	48.00	30.38
will199GPI	(1)	1.00	7.00	7.00	0.00	10.00	6.00	1.00	7.00	7.00	0.00	7.00	6.00

Table 2. Comparison with bottom-up methods (by classes of instances)

		cdcl↑			color6↑		
		opt	ub	lb	opt	ub	lb
DSJ	(14)	0.07	86.93	33.79	0.07	86.93	35.79
FullIns	(14)	0.93	6.86	6.71	0.21	7.29	5.43
Insertions	(11)	0.36	5.36	4.27	0.36	5.36	4.09
abb313GPI	(1)	0.00	14.00	9.00	0.00	14.00	8.00
ash	(3)	1.00	4.00	4.00	0.67	4.67	3.67
flat	(6)	0.00	82.17	14.00	0.00	82.17	16.83
fpsol2	(1)	1.00	65.00	65.00	0.00	65.00	59.00
inithx	(1)	1.00	54.00	54.00	0.00	54.00	43.00
latin_square	(1)	0.00	159.00	90.00	0.00	159.00	90.00
le450	(10)	0.80	14.70	13.00	0.20	20.00	13.00
miles	(5)	1.00	34.80	34.80	0.00	36.40	33.40
mug	(4)	1.00	4.00	4.00	1.00	4.00	4.00
myciel	(5)	1.00	6.00	6.00	0.80	6.00	5.60
qg	(4)	0.25	72.50	57.50	0.75	66.00	57.50
queen	(13)	0.46	14.54	10.92	0.00	15.92	10.62
school1	(1)	1.00	14.00	14.00	1.00	14.00	14.00
wap0	(8)	0.12	50.88	41.25	0.00	51.12	40.00
will199GPI	(1)	1.00	7.00	7.00	0.00	10.00	6.00

classes where `cdcl↑` is better than classes (such as `qg` again) where the opposite is true. Moreover, although `cdcl↑` finds better lower bounds on two large classes (`DSJ` and `flat`) this does not translates to a higher proof ratio.

Table 3 shows results aggregated across all instances. We report the average ratio of instance proven optimal (‘optimal’) in the first column. Then in the second to the fifth columns, we report the arithmetic (‘avg’) and geometric averages (‘gavg’) for both the lower and upper bounds. Finally, we report the *mean normalised gap to the best upper bound*, and to the best lower bound. Let b (resp. w) be the value found by best (resp. worst) method. In the case of the

Table 3. Comparison with the state of the art: global results

Method	Optimal	ub		lb		gap (ub)	gap (lb)
	avg	gavg	avg	gavg	avg	avg	avg
cdc1	0.53398	15.247	30.107	10.790	18.689	0.0909	0.2254
cdc1 ↑	0.53398	16.248	33.427	11.846	19.427	0.4175	0.0740
CPLEX	0.41748	16.503	33.388	10.886	18.379	0.4014	0.2562
color6 ↑	0.23301	17.408	34.068	11.527	19.252	0.6408	0.2371
color6	0.19417	16.314	31.233	10.040	17.748	0.3201	0.4716
Dsatur	0.12621	15.506	30.495	8.754	16.524	0.1450	0.7248

lower bound, b will be the maximum, while it will be the minimum for the upper bound. The normalised gap $g(x)$ of the outcome x is:

$$g(x) = \begin{cases} 0 & \text{if } b = w \\ (b - x)/(b - w) & \text{otherwise} \end{cases}$$

A mean normalised gap of 0 (resp. 1) therefore indicates that the method systematically has the best (resp. worst) outcome.

Overall, the variants of **cdc1** are best for all criteria. **CPLEX** is third best for the number of optimality proofs. Although it requires a lot of memory, and is very poor in terms of solution quality, **CPLEX** often gives good lower bounds. This is not so surprising since the linear relaxation is quite potent on this formulation. For instance at the root node, since we fix the variables of a maximal clique, the lower bound from the linear relaxation can only be higher than that of our method. It should be noted, however, that in many cases it was not able to improve on the initial bounds provided to the model, even when memory was not an issue. **color6**↑ is second best for the lower bound, however, notice that the much larger mean normalised gap to the best lower bound than **cdc1**↑ indicates that it was often close but rarely better than our approach. Finally, **Dsatur**, even though extremely simple, is still a very good method to actually find small colorings and is a close second best for the upper bound.

Next, we tried to assess the impact of the new bounds, and of learning. To that purpose, we ran six variants. Let L denote the usage of clause learning, M the mycielskian-based lower bound and O the partition-based vertex ordering used to find maximal cliques, then $X \setminus S$ stands for the solver X where the options in S are turned off. The results reported in Table 4 clearly show the impact of each factor. There is an almost perfect correlation between turning off a feature, and moving down the ranking for any criterion. In particular, clause learning has clearly a very high impact as turning it off systematically and significantly degrades the performances on every criterion. Moreover, using the partition-based vertex ordering also has a very significant impact for such a simple technique. Finally the mycielskian-based lower bound also clearly helps. However, its impact on the upper bound is limited. For instance, with respect

Table 4. Factor analysis: global results

Method	Optimal	ub		lb		gap (ub)	gap (lb)
	avg	gavg	avg	gavg	avg	avg	avg
<code>cdc1</code>	0.53398	15.247	30.107	10.790	18.689	0.0909	0.2254
<code>cdc1↑</code>	0.53398	16.248	33.427	11.846	19.427	0.4175	0.0740
<code>cdc1↑ \ M</code>	0.51456	16.219	33.466	11.738	19.262	0.4175	0.0925
<code>cdc1 \ M</code>	0.49515	15.308	30.126	10.364	18.272	0.0929	0.2764
<code>cdc1 \ M, O</code>	0.47573	15.469	30.534	10.234	18.282	0.1273	0.2890
<code>cdc1↑ \ M, O</code>	0.45631	16.370	33.476	11.701	19.369	0.4563	0.0988
<code>cdc1 \ M, O, L</code>	0.39806	15.521	30.524	10.034	18.184	0.1311	0.3602
<code>cdc1↑ \ M, O, L</code>	0.23301	17.861	34.476	10.724	18.738	0.6311	0.2808

to `cdc1 \ M`, it increases the proof ratio by 7.8% and the mean lower bound by 2.3%, but decreases the mean upper bound by only 0.3%.

4 Conclusions

We have presented a CP/SAT hybrid approach to graph coloring. The approach uses a new, sophisticated, lower bound that generalizes the clique bound and is inspired by Mycielskian graphs. We combined it with clause learning and effective primal heuristics for coloring to get a solver that in both its configurations outperforms the previous state of the art in satisfiability-based coloring, constraint programming based coloring, as well as a MIP model of the problem. The main disadvantage of the approach is that it requires one Boolean variable for each non-edge of the graph and hence cannot scale to large sparse graphs.

References

1. Aardal, K.I., Van Hoesel, S.P.M., Koster, A.M.C.A., Mannino, C., Sassano, A.: Models and solution techniques for frequency assignment problems. *Ann. Oper. Res.* **153**(1), 79–129 (2007)
2. Brélaž, D.: New methods to color the vertices of a graph. *Commun. ACM* **22**(4), 251–256 (1979)
3. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. *Comput. Lang.* **6**(1), 47–57 (1981)
4. De Cat, B., Denecker, M., Bruynooghe, M., Stuckey, P.: Lazy model expansion: interleaving grounding with search. *J. Artif. Intell. Res.* **52**, 235–286 (2015)
5. Eppstein, D., Löffler, M., Strash, D.: Listing all maximal cliques in large sparse real-world graphs. *ACM J. Exp. Algorithmics* **18**(3.1–3.21) (2013)
6. Frost, D., Dechter, R.: Look-ahead value ordering for constraint satisfaction problems. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-1995)*, pp. 572–578 (1995)

7. Gomes, C., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-1998), pp. 431–438 (1998)
8. Huang, J.: The effect of restarts on the efficiency of clause learning. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-2007) (2007)
9. Jiang, H., Li, C.-M., Manyà, F.: An exact algorithm for the maximum weight clique problem in large graphs. In: Proceedings of the 31st Conference on Artificial Intelligence (AAAI-2017), pp. 830–838 (2017)
10. Katsirelos, G., Bacchus, F.: Generalized nogoods in CSPs. In: Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-2005), pp. 390–396 (2005)
11. Lick, D.R., White, A.T.: k-degenerate graphs. *Can. J. Math.* **22**, 1082–1096 (1970)
12. Lin, J., Cai, S., Luo, C., Su, K.: A reduction based method for coloring very large graphs. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI-2017), pp. 517–523 (2017)
13. Marques-Silva, J.P., Sakallah, K.A.: GRASP—a search algorithm for propositional satisfiability. *IEEE Trans. Comput.* **48**(5), 506–521 (1999)
14. Mehrotra, A., Trick, M.A.: A column generation approach for graph coloring. *INFORMS J. Comput.* **8**(4), 344–354 (1996)
15. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 39th Design Automation Conference (DAC-2001), pp. 530–535 (2001)
16. Mycielski, J.: Sur le coloriage des graphes. *Colloq. Math* **3**, 161–162 (1955)
17. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation = lazy clause generation. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 544–558. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_39
18. Park, T., Lee, C.Y.: Application of the graph coloring algorithm to the frequency assignment problem. *J. Oper. Res. Soc. Jpn.* **39**(2), 258–265 (1996)
19. Schaafsma, B., Heule, M.J.H., van Maaren, H.: Dynamic symmetry breaking by simulating zykov contraction. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 223–236. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_22
20. Stergiou, K.: Heuristics for dynamically adapting propagation. In: Proceedings of the 18th European Conference on Artificial Intelligence (ECAI-2008), pp. 485–489 (2008)
21. Trick, M.A. (ed.): Computational Symposium on Graph Coloring and its Generalizations (COLOR-2002) (2002)
22. Van Gelder, A.: Another look at graph coloring via propositional satisfiability. *Discrete Appl. Math.* **156**(2), 230–243 (2008)
23. Van Hentenryck, P., Ågren, M., Flener, P., Pearson, J.: Tractable symmetry breaking for CSPs with interchangeable values. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-2003), pp. 277–282 (2003)
24. Walsh, T.: Breaking value symmetry. In: Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI-2008), pp. 880–887 (2008)
25. Zhou, Z., Li, C.-M., Huang, C., Ruchu, X.: An exact algorithm with learning for the graph coloring problem. *Comput. Oper. Res.* **51**, 282–301 (2014)
26. Zykov, A.A.: On some properties of linear complexes. *Mat. Sb. (N.S.)* **24(66)**(2), 163–188 (1949). <http://mi.mathnet.ru/eng/msb5974>



Portfolio-Based Algorithm Selection for Circuit QBFs

Holger H. Hoos¹, Tomáš Peitl²(✉), Friedrich Slivovsky², and Stefan Szeider²

¹ Leiden Institute of Advanced Computer Science,
Leiden University, Leiden, The Netherlands
hh@liacs.nl

² Algorithms and Complexity Group, TU Wien, Vienna, Austria
{[peitl](mailto:peitl@ac.tuwien.ac.at),[fslivovsky](mailto:fslivovsky@ac.tuwien.ac.at),[sz](mailto:sz@ac.tuwien.ac.at)}@ac.tuwien.ac.at

Abstract. Quantified Boolean Formulas (QBFs) are a generalization of propositional formulae that admits succinct encodings of verification and synthesis problems. Given that modern QBF solvers are based on different architectures with complementary performance characteristics, a portfolio-based approach to QBF solving is particularly promising.

While general QBFs can be converted to prenex conjunctive normal form (PCNF) with small overhead, this transformation has been known to adversely affect performance. This issue has prompted the development of several solvers for circuit QBFs in recent years.

We define a natural set of features of circuit QBFs and show that they can be used to construct portfolio-based algorithm selectors of state-of-the-art circuit QBF solvers that are close to the virtual best solver. We further demonstrate that most of this performance can be achieved using surprisingly small subsets of cheaply computable and intuitive features.

1 Introduction

The advent of modern satisfiability (SAT) solvers has established propositional logic as the low-level language of choice for encoding hard combinatorial problems arising in domains such as formal verification [4, 27] and AI planning [23]. However, since the computational complexity of these problems usually outstrips the complexity of SAT, propositional encodings of such problems can be exponentially larger than their original descriptions. This imposes a limit on the problem instances that can be feasibly solved even with extremely efficient SAT solvers, and has prompted research on decision procedures for more succinct logical formalisms such as Quantified Boolean Formulas (QBFs).

QBFs augment propositional formulas with existential and universal quantification over truth values and can be exponentially more succinct. The flip side of this conciseness is that the satisfiability problem of QBFs (QSAT) is PSPACE-complete [25], and in spite of substantial progress in solver technology, practically relevant instances remain hard to solve. The complexity of QSAT

This research was partially supported by FWF grants P27721 and W1255-N23.

is also reflected in the fact that there is currently no single best QBF solver—in fact, state-of-the-art solvers are based on fundamentally different paradigms whose underlying proof systems are known to be exponentially separated [3, 10]. In particular, it has been observed that expansion-based solvers work better than search-based solvers on formulas with few quantifier alternations, while search-based solvers tend to be better suited to formulas with many quantifier alternations [17].

Thus, even more so than in the case of SAT, portfolio-based approaches that leverage the complementary strength of multiple QBF solvers, such as per-instance algorithm selection, have the potential to achieve significant speedups over individual solvers, as demonstrated for QBF formulae in the prenex CNF (PCNF) format [20]. Although any QBF can be converted to PCNF with small overhead, this transformation is known to adversely affect solver performance [1]; moreover, it can obscure features of the original instance that might be strong predictors of solver performance. In light of the first issue, researchers have developed a new standard, QCIR, for representing *quantified circuits*, or *circuit QBFs* [12],¹ while the second issue is potentially relevant to per-instance algorithm selection.

In this work, we present the first per-instance algorithm selector for QCIR formulae, built from four state-of-the-art QBF solvers, and demonstrate that it achieves performance substantially better than any of the individual solvers and close to the theoretical upper bound given by the virtual best solver (VBS) both in terms of overall runtime and number of solved instances. Following common practice, we developed and used a large set of static and dynamic instance features for this purpose. To our surprise, we discovered that, different from the situation for SAT, probing features are not helpful, and a set of only three static instance features are sufficient to achieve 99% of the performance gain obtained using our full set of features. Interestingly, these features are simple, cheaply-computable and intuitively characterize the quantification and circuit structure of the instance. Therefore, our work provides evidence that, different from what might gather from the literature, to effectively leverage per-instance algorithm selection, at least in some cases, a small set of easily implemented and computed features is sufficient. This is a significant finding, since it further lowers the barrier for researchers to effectively apply algorithm selection.

2 Related Work

For many problems in AI, there is no single algorithm that is clearly superior to all other algorithms. This may be due to algorithms implementing heuristics that work well on some instance type but not on others. Per-instance algorithm selection (as originally introduced by Rice [22]) attempts to mitigate this issue by choosing the algorithm that is expected to solve a given instance most efficiently.

¹ We only consider “cleansed” QCIR instances in prenex normal form supported by the current generation of solvers.

In recent years, algorithm selection tools have been successfully applied to a variety of AI problems, such as SAT, CSP, ASP, and QBF [5, 18, 20, 29]. The most common approach to algorithm selection involves picking an algorithm from a set of algorithms called a portfolio. Since the relationship between properties of a problem instance and algorithm performance is typically opaque and hard to capture formally, the construction of a portfolio normally involves training a machine learning model to predict performance and choose an algorithm [14].

In the context of QBF, multinomial logistic regression has been used to switch between different branching heuristics in a search-based QBF solver based on instance features, even at runtime [24]. The (PCNF) portfolio solver AQME incorporates several models such as decision trees and nearest neighbor classification [20]. Moreover, it is “self-adaptive” in the sense that it can modify its performance prediction model to accommodate for instance types not seen during initial training. HORDEQBF is a massively parallel QBF solver [2] that implements a parallel portfolio by running multiple instances of the solver DEPQBF [16] with different parameter settings.

Automated parameter tuning is an area that is gaining popularity due to algorithms increasingly having a large number of parameters that are virtually impossible to tune by hand [6, 7]. Parameter tuning can be combined with portfolio construction in order to find algorithm configurations that complement each other well [28]. Algorithm selectors typically have many options themselves (such as the choice of machine learning model and its corresponding hyperparameters), and parameter tuning can also be used to configure the selector [15].

3 Preliminaries

3.1 Circuit QBF Solvers

Our portfolios comprise the QBF solvers that participated in the prenex non-CNF track of the 2017 QBF Evaluation² (with the exception of CQesto, which is not publicly available; for all solvers, the default configurations provided by their authors were used). Their performance on the corresponding benchmark set was fairly similar, with the number of solved instances ranging from 89 (GHOSTQ) to 117 (QFUN) out of a total 320.

1. QUABS [26] generalizes the concept of “clause selection” (as implemented in QESTO [11] and CAQE [21]) from clauses to subformulas. An abstraction is maintained for each quantifier block, and so-called interface literals communicate whether a subformula is satisfied or falsified at a lower (or higher) level.
2. QFUN [8] generalizes counterexample-guided abstraction refinement (CEGAR) solving [9] to circuit QBFs and uses decision tree learning to “guess” counter(models) based on recent truth assignments.

² See <http://www.qbffb.org>.

3. QUTE [19] is a search-based solver that implements a technique called dependency learning to ignore artificial syntactic dependencies induced by nested quantifiers.
4. GHOSTQ [13] is a search-based solver that utilizes so-called ghost literals for dual propagation.

3.2 AutoFolio

AUTOFOLIO is an algorithm selector that alleviates the burden of manually choosing the right machine learning model for a problem domain and hand-tuning hyperparameters by using algorithm configuration tools to automatically make design choices and find hyperparameter settings that work well for a particular scenario [15].

AUTOFOLIO allows us to construct a portfolio from the above solvers with little effort. In particular, it quickly lets us create portfolios that are tuned to particular subsets of features (see Sect. 6). Our main design choice consists in defining the set of features described in the next section.

4 QCIR Instance Features

We consider circuit Quantified Boolean Formulas (QBFs) in prenex normal form encoded according to the “cleansed” QCIR standard [12]. Each such formula is a pair $\mathcal{F} = \mathcal{Q}.\varphi$ consisting of a *quantifier prefix* \mathcal{Q} and a Boolean circuit φ called the *matrix* of \mathcal{F} . The quantifier prefix \mathcal{Q} is a sequence $Q_1X_1 \dots Q_kX_k$ where each $Q_i \in \{\forall, \exists\}$ is a *quantifier* for $1 \leq i \leq k$ such that $Q_i \neq Q_{i+1}$ for $1 \leq i < k$, and the X_i are pairwise disjoint sets of variables called *quantifier blocks*.

The matrix φ is a Boolean circuit encoded as a sequence of gate definitions of the form

$$g = \circ(l_1, \dots, l_r)$$

where $\circ \in \{\wedge, \vee\}$, each *gate literal* l_i is either an unnegated gate variable g' (a *positive* gate literal) or a negated gate variable $\neg g'$ (a *negative* gate literal), and g' is a previously defined gate or an input gate $g' \in \bigcup_{i=1}^k X_i$. We refer to r as the *size* of gate g . The *depth* of a gate g is 0 if g is an input gate, and otherwise the maximum depth of a gate occurring in the definition of g plus one. A unique gate literal is identified as the output of the circuit φ .

We consider the following *static features* of QCIR instances:

1. The number n_e of existential variables.
2. The number n_u of universal variables.
3. The balance $n_e/n_u + n_u/n_e$ of existential and universal variables.
4. The number k of quantifier blocks.
5. The minimum size \min_b of a quantifier block.
6. The maximum size \max_b of a quantifier block.
7. The average size μ_b of a quantifier block.
8. The standard deviation σ_b of the quantifier block size.

9. The relative standard deviation σ_b/μ_b of the quantifier block size.
10. The total number pos of positive gate literals.
11. The total number neg of negative gate literals.
12. The balance $pos/neg + neg/pos$ of positive and negative gate literals.
13. The number n_\wedge of AND gates.
14. The number n_\vee of OR gates.
15. The maximum gate size max_{gs} .
16. The average gate size μ_{gs} .
17. The standard deviation σ_{gs} of the gate size.
18. The relative standard deviation σ_{gs}/μ_{gs} of the gate size.
19. The maximum gate depth max_d .
20. The average gate depth μ_d .
21. The standard deviation σ_d of the gate depth.
22. The relative standard deviation σ_d/μ_d of the gate depth.
23. The number n_p of gates all of whose gate literals have the same polarity (all positive or all negative).

Features that only depend on the quantifier prefix can be computed just as well for PCNF instances, and indeed some of the features 1–9 were already used in constructing the portfolio solver AQME [20]. The main difference between PCNF and QCIR is in the representation of the matrix and accordingly, this is where new features are required. Some of the above features (such as the numbers of AND/OR gates) can be seen as generalizations of PCNF features (number of clauses). Others, such as the maximum gate depth, only make sense for circuits.

In addition to these static features, we use several *probing features* computed by a short run of QUTE (probing features are crucial for the performance of portfolios for SAT [29]):

1. The number of learned clauses.
2. The number of learned tautological clauses.
3. The number of learned terms.
4. The number of learned contradictory terms.
5. The fraction of variable assignments made by branching (the remaining assignments are due to propagation).
6. The total number of backtracks.
7. The number of backtracks due to dependency learning (a feature of QUTE).
8. The number of learned dependencies as a fraction of the trivial dependencies.

5 Per-instance Algorithm Selection for QCIR

The experiments were conducted on a cluster where each node is equipped with 2 Intel Xeon E5-2640 v4 processors (25M Cache, 2.40 GHz) and 160 GB of RAM. The machines are running 64-bit Ubuntu in version 16.04.3.

We work with the set of QCIR benchmark instances from the 2016 and 2017 QBF evaluations solved by at least one of the above solvers within 900s of CPU

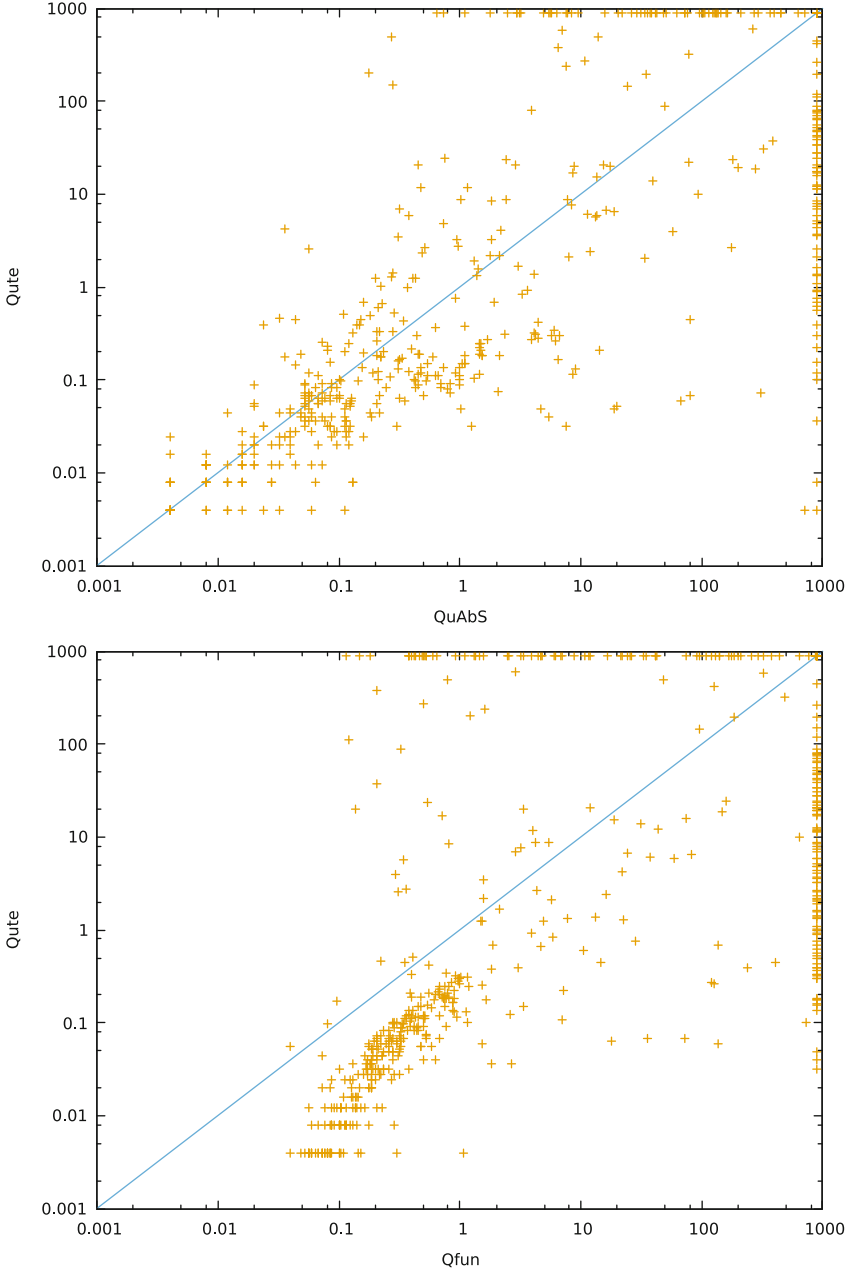


Fig. 1. Comparisons of high-performance QBF solvers on our instance set; performance is measured as PAR 10 (penalized running times with penalty factor 10) on our reference machines. This shows that there is quite a lot of complementarity between the solvers.

time and 4 GB of memory usage, a total of 731 instances. Figure 1 illustrates that there is a lot of complementarity between the component solvers. We split the 731 instances into a training set of 549 instances and a test set of 182 instances, uniformly at random. On the training set we fixed a cross-validation split into 10 folds of the same size. When we report performance of a selector on the *training* set, we in fact report cross-validation performance on this fixed split. This means that the selector was trained once on each subset of 9 folds and evaluated on the 10th one, and the results were combined. On the other hand, when we report performance on the test set, the respective selector is trained on the *entire* training set, disregarding the CV-split, and then evaluated on the entire test set. The reason why we use this setup for our evaluation is the following. The standard way to evaluate the performance of AUTOFOLIO is by using cross-validation. However, if AUTOFOLIO is tuned to the specific CV-split, the CV performance may be an overly optimistic estimate of how well the model will generalize. Even though cross validation should still protect us from overfitting, we decided to hold out a test set even on top of that, in order to perform a sanity check of the experiment afterwards.

Each of the selectors PF* mentioned in Table 1 was trained using AUTOFOLIO in self-tuning mode, with a budget of 42 000 wall-clock seconds and a bound of 50 000 runs for the algorithm configuration tool SMAC, and with a specific subset of features (see the next section and caption of Table 1 for details). For the SMAC-configuration phase we used the CV-split as mentioned earlier. The selectors PFA, PFS, and PF3 use an XGBoost classifier, while PF2 uses a random-forest regressor.

6 Which Features Matter?

It is common wisdom that high-performance per-instance algorithm selectors should have access to a large and rich set of features (see, e.g. [29]). While earlier selector designs based on ridge regression required feature selection to work well, state-of-the-art per-instance selectors make use of sophisticated machine learning techniques, such as random forests, that are less sensitive to uninformative or correlated features. However, defining and computing features requires substantial domain expertise and often involves significant amount of work, especially since feature computation must be efficient in order to achieve good selector performance. Furthermore, selectors based on large sets of complex features can be far more difficult to understand than ones based on few and simple features. Since our full feature set for QCIR formulae, as described previously, gave rise to excellent selector performance, we decided to investigate whether similarly good performance could be obtained with fewer features.

We first trained a selector using only our static features, using AUTOFOLIO, as described in the previous section. The resulting selector, denoted PFS in Table 1, performed slightly better than the selector trained using the full set of static and probing features (PFA). This was a great surprise to us in light of previous work on algorithm selection in which probing features were found to be

helpful (see, e.g. [14]). Since our full selector is already very close in performance to the VBS, it cannot be the case that we simply failed to come up with the right probing features, but rather that in the scenario we consider, static features are sufficient. Prompted by this finding, we decided to investigate the effect of further reducing our static features set.

In order to test what feature subsets might work well, we used the following setup. We configured AUTOFOLIO using the static features, and we saved the resulting configuration of hyperparameters. Then, with this configuration of AUTOFOLIO, we performed forward and backward selection on the set of static features. In forward selection, we started with the empty set of features, and at each step added a single feature, while in backward selection we started with the full set of static features, and at each step removed a feature. In both cases, the feature to be added/removed was chosen so that the resulting portfolio would have maximum performance. It is important to note here that we *did not* configure AUTOFOLIO for each of the subsets searched in this process—instead we used the configuration that we computed as described at the beginning of this paragraph. The reason for that was to avoid the huge computational cost of configuring AUTOFOLIO over and over again. In retrospect, this was indeed justified, as we obtained well-performing selectors for the feature subsets even this way, and we saved months of CPU time. However, note that once we found promising subsets of features by forward/backward selection, we configured AUTOFOLIO for these subsets again, and the results of those specifically configured selectors are reported in Table 1.

Table 1. Performance of component solvers and selectors on the training and test sets in terms of penalized average runtime (PAR10), the number of solved instances, and for selectors the extent to which they match the virtual best solver (VBS) measured as the percentage of the PAR10 gap between the single best solver (SBS) and the VBS that is closed by the selector. Training performance of selectors is CV-performance. Selectors were configured using AUTOFOLIO in self-tuning mode for each of the feature subsets reported. PF2 is the selector configured for the best subset of 2 features, similarly PF3, PFS uses static features only, and PFA uses all features.

Solver	Training set (549)			Test set (182)		
	PAR10	#solved	%closed	PAR10	#solved	%closed
GhostQ	2228.92	414	–	2492.61	132	–
Qfun	1922.07	433	–	2384.68	134	–
QuAbS	1641.90	450	–	1747.40	147	0%
Qute	1458.09	461	0%	1845.48	145	–
PFA	71.93	546	96.35%	171.03	179	91.01%
PF2	57.58	547	97.35%	217.16	178	88.34%
PF3	55.78	547	97.47%	165.97	179	91.30%
PFS	55.65	547	97.48%	167.53	179	91.21%
VBS	19.46	549	100%	15.35	182	100%

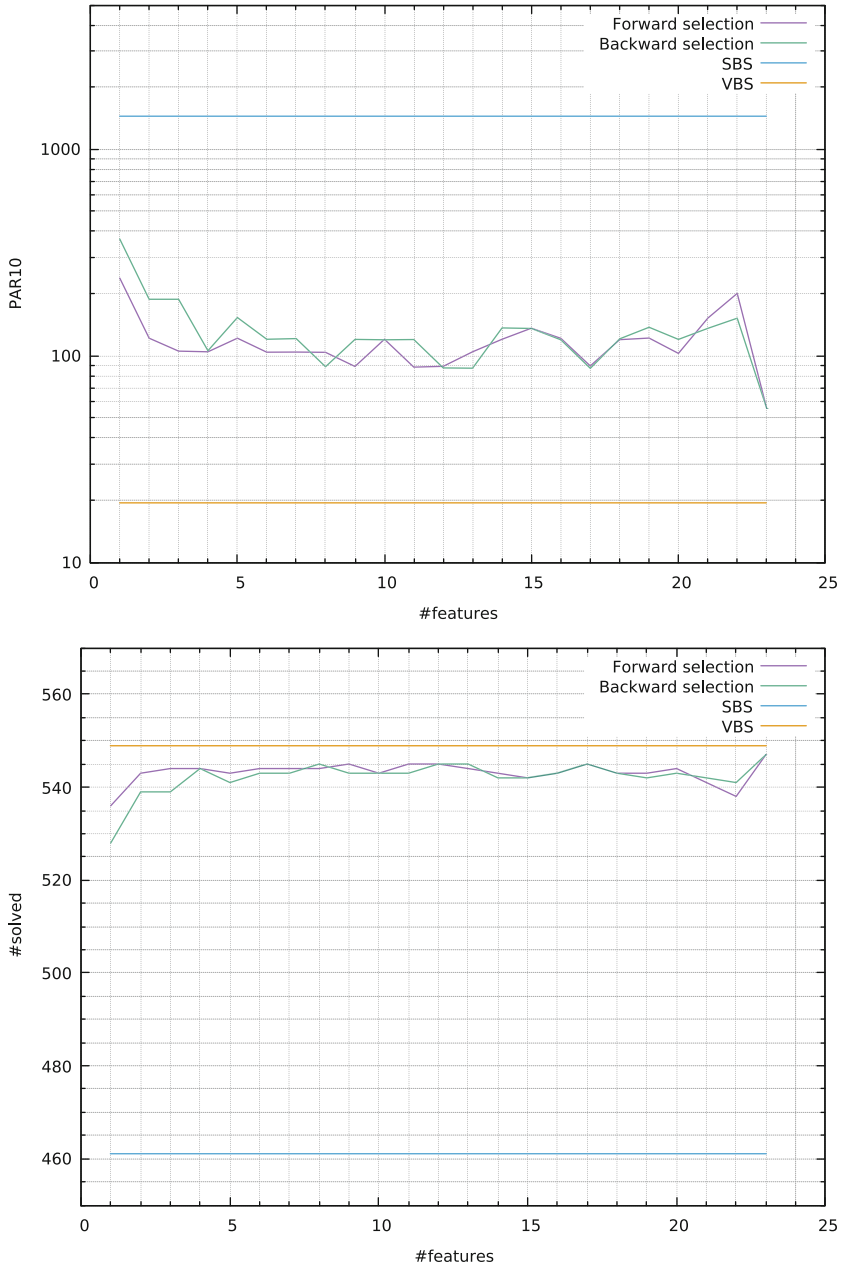


Fig. 2. Forward and backward selection on the static features; the plots show performance based on the number of features included. Note that for the performance evaluation during forward/backward selection, AUTOFOLIO was not automatically configured for each subset of features, but instead was once configured for the full set of static features at the beginning, and this configuration of hyperparameters was subsequently used for all feature subsets.

Figure 2 shows the performance curve along forward/backward selection. The values of PAR10 and the number of solved instances were obtained by performing cross validation on the fixed CV-split mentioned earlier. In particular, we can see that forward selection achieves very good performance with two or three features already. The first three features picked by forward selection are *circuit depth*, *number of quantifier blocks*, and *average block size*. Since so few features turned out to yield such good selectors, we performed a brute-force search of all subsets of size 2 or 3 (again, evaluating performance with the fixed AUTOFOLIO configuration used for forward/backward selection). This search confirmed that the size-2 subset found by forward selection was almost optimal (second best, equal number of solved instances as with the optimal set, difference of 1.2 in PAR10), while the size-3 subset was optimal. We decided to continue the experiment with the size-2 subset found by forward selection (instead of the “optimal” one), for two reasons. Firstly, it contains the feature *circuit depth*, which is the best single predictor, but which is replaced in the optimal subset by *relative standard deviation of gate depths*, a feature that is somewhat harder to interpret. Secondly, we need to keep in mind, that not even this exhaustive search was perfect, as we did not (and could not) configure AUTOFOLIO for each subset searched. Therefore, its results only served as a sanity check, to make sure that forward selection did not miss some great feature set, which turned out not to be the case. Hence, we went on to configure AUTOFOLIO for the subsets $\{\textit{circuit depth}, \textit{number of quantifier blocks}\}$, and $\{\textit{circuit depth}, \textit{number of quantifier blocks}, \textit{average block size}\}$, the results of which are shown in Table 1 (entries PF2 and PF3). As Table 1 shows, PF2 achieves virtually the same good performance as PFS, and closes almost all of the gap between SBS and VBS. This holds whether we look at the CV-evaluation on the training set, or the additional evaluation on the test set (Fig. 3).

As a final sanity check, we evaluated the performance of selectors trained using these small sets of features on the same set of instances, but using only 3 out the 4 participating solvers (for each subset of 3 solvers). We set this experiment up in the following way: for each subset of features corresponding to one of the selectors PF*, we saved the configuration of AUTOFOLIO that was optimized for the particular subset of features using all four solvers. We then evaluated the performance of selectors built using the saved configurations for each of the 4 size-3 solver subsets (a total of 16 selectors), in the same way as we did for Table 1. In order to get the theoretically best AUTOFOLIO performance, we would have had to reconfigure AUTOFOLIO for every pair of (solver subset, feature subset), but as before we simplified things to save computational resources. This experiment confirmed that even for different solver sets, the features *circuit depth*, and *number of quantifier blocks* are fairly robust predictors of solver performance. However, naturally, features must be tied to solvers whose performance they predict, so we cannot expect that a fixed set of features will be a universal predictor for all solver sets.

In a sense, these results are not surprising, as one would expect from complexity theory as well as from previous work that the number of quantifier blocks

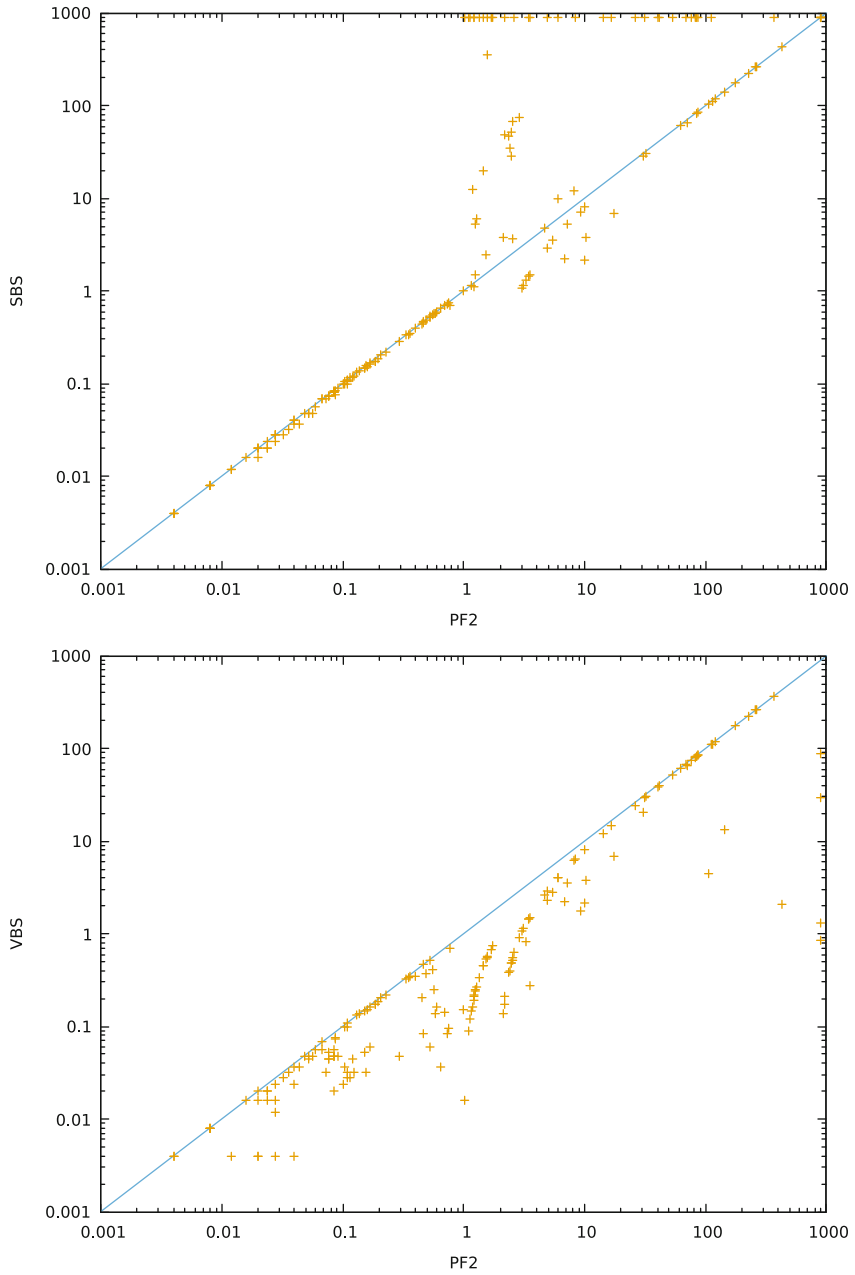


Fig. 3. Performance of PF2 with all four solvers vs SBS and VBS.

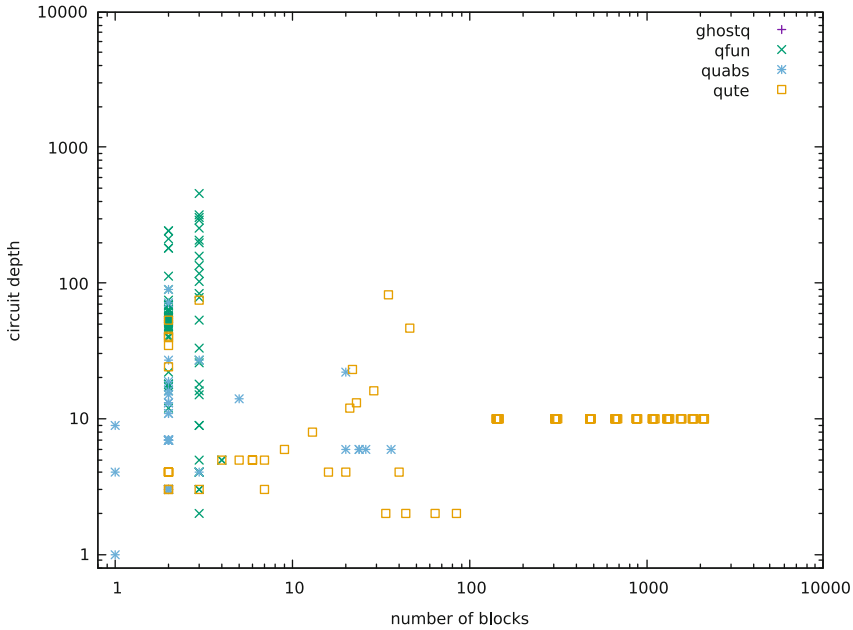


Fig. 4. Best solver choices based on instance features. Each point represents an instance/solver pair; the coordinates correspond to the number of quantifier blocks and circuit depth of the instance, the shape and color indicate the solver that is fastest on that instance. Only instances where the fastest solver is either the only one to solve the instance, or at least ten times faster than the second fastest, are shown. This is to ensure that the figure shows only solver choices that are crucial, and to avoid instances where the solver choice is unimportant, because all of them run in similar time.

indeed plays an important role. Similarly, circuit depth seems to be a prominent property of circuits. However, it is indeed striking that only two, and moreover the most straightforward features of circuit QBF suffice to build such robust portfolios. We believe that this opens up a new path of thinking for both solver users and developers. Users can classify their benchmarks and pick a suitable solver more easily, while developers can take advantage of this information to build portfolios within their solvers. Believing many features are necessary to learn anything meaningful about a given instance can be discouraging from even trying. With just two features, the options are much wider—they can be understood intuitively, or even plotted. In fact, to demonstrate how we can gain additional insight into the problem, we visualize the solver choices made both by the portfolio, as well as by the VBS. When plotting the VBS in Fig. 4, we ignore instances where the solvers perform too similarly, because they contain more noise than information. On the other hand, we plot the portfolio choices in Fig. 5 as a grid (of hypothetical instances), in order to discover the decision boundaries. These figures show very clearly which solvers are good for which

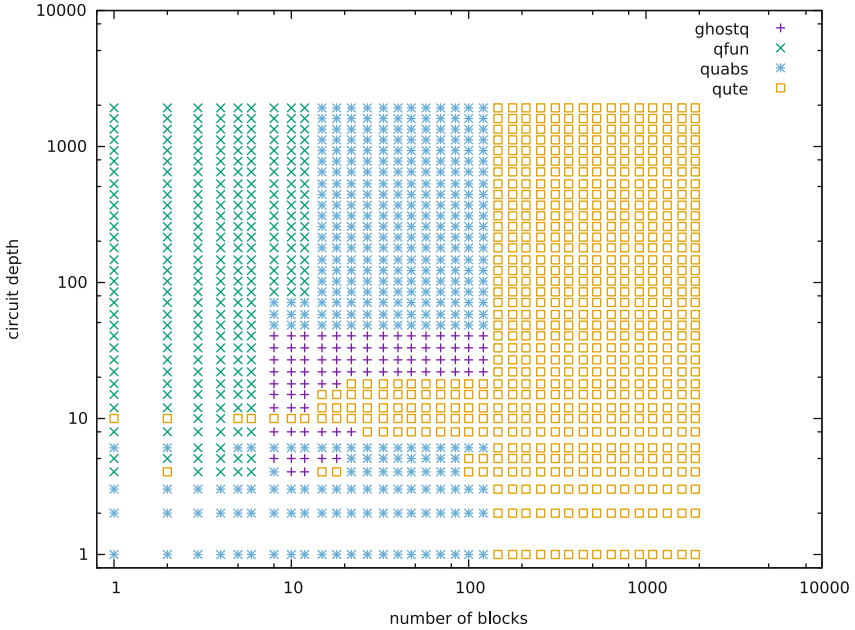


Fig. 5. Points indicate solver choices of PF2 based on feature values.

instances. Incidentally, Fig. 4 also reveals the fact that the QCIR instances that are available either have many quantifier blocks, or deep circuits, or neither, but not both (strictly speaking, to see that, we would need to plot all instances, but the picture has the same shape, only more noise). This should serve as a challenge to the QBF community to come up with a more complete distribution of benchmark instances.

7 Conclusions and Future Work

With the availability of tools such as AUTOFOLIO, the task of constructing effective per-instance algorithm selectors essentially boils down to designing and implementing features that (jointly) permit to effectively identify which solver to run on any given problem instance. This can still seem daunting in view of the fact that certain domains require rich sets of quickly computable features, with a combination of static and dynamic features, in order to achieve good selector performance [29]. Our results show that this need not be the case: for circuit QBFs, two or three cheaply computable instance features are sufficient to realize most of the performance potential of a (hypothetical) perfect selector. Moreover, these features include properties of QBFs such as the number of quantifier blocks that are known to affect solver performance. Apart from corroborating the notion that quantifier alternations matter, our results show that circuit depth seems to be important. This warrants further investigation.

Our finding that simple feature sets can be effective likely applies to other problems and encourages an incremental design philosophy: start with a few simple features and add features as needed. As part of future work we hope to find other domains where this approach works well and, more generally, identify the circumstances under which this is the case.

References

1. Ansótegui, C., Gomes, C.P., Selman, B.: The Achilles' heel of QBF. In: Veloso, M.M., Kambhampati, S. (eds.) *The Twentieth National Conference on Artificial Intelligence - AAAI 2005*, pp. 275–281. AAAI Press/The MIT Press (2005)
2. Balyo, T., Lonsing, F.: HordeQBF: a modular and massively parallel QBF solver. In: Creignou, N., Le Berre, D. (eds.) *SAT 2016*. LNCS, vol. 9710, pp. 531–538. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_33
3. Beyersdorff, O., Chew, L., Janota, M.: Proof complexity of resolution-based QBF calculi. In: Mayr, E.W., Ollinger, N. (eds.) *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, 4–7 March 2015, Garching, Germany*. LIPIcs, vol. 30, pp. 76–89. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)
4. Biere, A.: Bounded model checking. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, vol. 185, pp. 457–481. IOS Press (2009)
5. Gebser, M., et al.: A portfolio solver for answer set programming: preliminary report. In: Delgrande, J.P., Faber, W. (eds.) *LPNMR 2011*. LNCS (LNAI), vol. 6645, pp. 352–357. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20895-9_40
6. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Coello, C.A.C. (ed.) *LION 2011*. LNCS, vol. 6683, pp. 507–523. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25566-3_40
7. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: Paramils: an automatic algorithm configuration framework. *J. Artif. Intell. Res.* **36**, 267–306 (2009)
8. Janota, M.: Towards generalization in QBF solving via machine learning. In: McIlraith, S.A., Weinberger, K.Q. (eds.) *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence - AAAI 2018*. AAAI Press (2018)
9. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with counterexample guided refinement. In: Cimatti, A., Sebastiani, R. (eds.) *SAT 2012*. LNCS, vol. 7317, pp. 114–128. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_10
10. Janota, M., Marques-Silva, J.: Expansion-based QBF solving versus Q-resolution. *Theor. Comput. Sci.* **577**, 25–42 (2015)
11. Janota, M., Marques-Silva, J.: Solving QBF by clause selection. In: Yang, Q., Wooldridge, M. (eds.) *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015*. pp. 325–331. AAAI Press (2015)
12. Jordan, C., Klieber, W., Seidl, M.: Non-CNF QBF solving with QCIR. In: Darwiche, A. (ed.) *Beyond NP, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016*. AAAI Workshops, vol. WS-16-05. AAAI Press (2016)

13. Klieber, W., Sapra, S., Gao, S., Clarke, E.: A non-prenex, non-clausal QBF solver with game-state learning. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 128–142. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14186-7_12
14. Kotthoff, L.: Algorithm selection for combinatorial search problems: a survey. In: Bessiere, C., et al. (eds.) Data Mining and Constraint Programming. LNCS (LNAI), vol. 10101, pp. 149–190. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50137-6_7
15. Lindauer, M.T., Hoos, H.H., Hutter, F., Schaub, T.: Autofolio: an automatically configured algorithm selector. *J. Artif. Intell. Res.* **53**, 745–778 (2015)
16. Lonsing, F., Egly, U.: DepQBF 6.0: a search-based QBF solver beyond traditional QCDCL. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 371–384. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_23
17. Lonsing, F., Egly, U.: Evaluating QBF solvers: quantifier alternations matter. CoRR abs/1701.06612 (2017). <http://arxiv.org/abs/1701.06612>
18. O’Mahony, E., Hebrard, E., Holland, A., Nugent, C., O’Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Irish Conference on Artificial Intelligence and Cognitive Science, pp. 210–216 (2008)
19. Peitl, T., Slivovsky, F., Szeider, S.: Dependency learning for QBF. In: Gaspers, S., Walsh, T. (eds.) SAT 2017. LNCS, vol. 10491, pp. 298–313. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_19
20. Pulina, L., Tacchella, A.: A self-adaptive multi-engine solver for quantified Boolean formulas. *Constraints* **14**(1), 80–116 (2009)
21. Rabe, M.N., Tentrup, L.: CAQE: a certifying QBF solver. In: Kaivola, R., Wahl, T. (eds.) Formal Methods in Computer-Aided Design - FMCAD 2015, pp. 136–143. IEEE Computer Society (2015)
22. Rice, J.R.: The algorithm selection problem. *Adv. Comput.* **15**, 65–118 (1976)
23. Rintanen, J.: Planning and SAT. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 483–504. IOS Press (2009)
24. Samulowitz, H., Memisevic, R.: Learning to solve QBF. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22–26, 2007, Vancouver, British Columbia, Canada, pp. 255–260. AAAI Press (2007)
25. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time: preliminary report. In: Aho, A.V., et al. (eds.) Proceedings of the 5th Annual ACM Symposium on Theory of Computing, 30 April–2 May 1973, Austin, Texas, USA, pp. 1–9. Association for Computing Machinery, New York (1973)
26. Tentrup, L.: Non-prenex QBF solving using abstraction. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 393–401. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_24
27. Vize!, Y., Weissenbacher, G., Malik, S.: Boolean satisfiability solvers and their applications in model checking. *Proc. IEEE* **103**(11), 2021–2035 (2015)
28. Xu, L., Hoos, H., Leyton-Brown, K.: Hydra: automatically configuring algorithms for portfolio-based selection. In: Fox, M., Poole, D. (eds.) Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, 11–15 July 2010. AAAI Press (2010)
29. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Satzilla: portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.* **32**, 565–606 (2008)



Making Compact-Table Compact

Linnea Ingmar¹ and Christian Schulte²(✉)

¹ Uppsala University, Uppsala, Sweden

`linnea.ingmar.3244@student.uu.se`

² KTH Royal Institute of Technology, Stockholm, Sweden

`cschulte@kth.se`

Abstract. The compact-table propagator for table constraints appears to be a strong candidate for inclusion into any constraint solver due to its efficiency and simplicity. However, successful integration into a constraint solver based on *copying* rather than *trailing* is not obvious: while the underlying bit-set data structure is *sparse* for efficiency it is not *compact* for memory, which is essential for a copying solver.

The paper introduces techniques to make compact-table an excellent fit for a copying solver. The key is to make sparse bit-sets *dynamically compact* (only their essential parts occupy memory and their implementation is dynamically adapted during search) and tables *shared* (their read-only parts are shared among copies). Dynamically compact bit-sets reduce peak memory by 7.2% and runtime by 13.6% on average and by up to 66.3% and 33.2%. Shared tables even further reduce runtime and memory usage. The reduction in runtime exceeds the reduction in memory and a cache analysis indicates that our techniques might also be beneficial for trailing solvers. The proposed implementation has replaced Gecode's original implementations as it runs on average almost an order of magnitude faster while using half the memory.

1 Introduction

The compact-table propagator [5] implements table constraints, where an explicit table of tuples defines the solutions to the constraint. Its basic idea is to assign a number to each tuple in the table and maintain a sparse bit-set where bit number i is set iff the tuple with number i is still considered a possible solution. The sparse bit-set is represented as an array of words (typically, words of 64 bits) and is *sparse*: operations performed on it by the propagator only consider words that have at least one bit set (that is, non-zero or non-empty words), where the emptiness information is tracked by an *index* structure. Compact-table and its extensions have already shown great potential [5, 14, 15] and sparse bit-sets have also been successfully used for itemset mining constraints [12].

The above-mentioned papers use constraint solvers that are based on *trailing* where changes during propagation and search are recorded and undone when backtracking occurs. Solvers based on *copying* create copies of the solver's state to which they can return during backtracking [10, 13]. For a copying solver it is

crucial that the state to be copied be small, so any of its propagators should require as little memory as possible to be copied. This paper contributes how this can be achieved for the compact-table propagator.

Operations on sparse bit-sets save time as they safely ignore empty words. However, empty words might still occupy memory as they are interleaved with non-empty words in memory. This does not matter for a trailing solver as the bit-set exists in one copy, however it poses a problem for a copying solver where the bit-set needs to be copied for each node of the search tree. We contribute how to make bit-sets *sparse as well as compact*: non-empty words move to the beginning of the word array and hence only its non-empty prefix needs copying. Additionally, compact bit-sets are more cache-friendly and require fewer indirections during update, which might be also beneficial for trailing solvers.

We make the propagator parametric with respect to its sparse bit-set implementation, so that we can get variants specialized for small tables. Here we take advantage by compressing the index structure or dropping it altogether. This optimization is *dynamic*: when the propagator is copied, the current size of its sparse bit-set decides which implementation is best for the copy. The rationale is that most copies are created close to the leaves of the search tree and hence many words of the bit-sets might be empty.

It is important that as much information as possible of the table that is read-only to the propagator be *shared* among its copies and among propagators using the same table for different constraints. We introduce a design where tables can be shared and only requires two mutable pointers per propagator variable.

The paper evaluates the various design decisions showing that the implementation of table constraints based on compact-table outperforms the original implementations in Gecode. We demonstrate that compactness is important, identify a promising hybrid candidate, and demonstrate that sharing is beneficial while residues (discussed below) are not beneficial in the context of Gecode.

Plan of the Paper. The next section reviews the compact-table propagator. Section 3 shows how tables can be shared between several propagators. Section 4 introduces techniques for dynamic compact sparse bit-sets which are evaluated in Sect. 5 and Sect. 6 concludes the paper.

2 Compact-Table

Throughout the paper we assume: the n tuples in the table t are numbered from 0 to $n - 1$; the i -th tuple is denoted as t_i ; the constraint (and hence t) has arity a ; the value at position k ($1 \leq k \leq a$) of tuple t_i is denoted as $t_{i,k}$; the variables are x_1, \dots, x_a where the domain of variable x_k is $\text{dom}(x_k)$; a tuple t_i is a *support* for a *variable-value* pair $\langle x_k, v \rangle$ and for a *position-value* pair $\langle k, v \rangle$ if $t_{i,k} = v$.

Sparse Bit-Sets. The n tuples are maintained in a sparse bit-set per propagator where bit number i is set iff the tuple t_i is still considered a possible solution. The sparse bit-set is an array `words` of words of 64 bits and is *sparse*: its operations

only consider non-empty words, where the emptiness information is tracked by an *index* structure. The index structure is an array `index` of 32-bit words that maintain a permutation of the indices of `words` and a counter `limit` for the current number of non-empty words. The first `limit` entries of `index` store the indices of `words` that are currently non-empty. If `limit` reaches zero the entire bit-set is empty. The index structure is key to sparseness: bit-set operations only need to consider words with indices in the first `limit` entries of `index`.

Modifications to the sparse bit-set are performed by intersections (word by word bit-wise *and*, denoted as $\&$) with a temporary mask. If the word at `index[i]` in `words` becomes empty, then the index structure records this by swapping `index[i]` with `index[limit - 1]` in `index` and decrementing `limit` by one. By doing so, non-empty indices move to the front of `index` while their order in `words` remains unchanged. The following invariant is maintained, where w is the number of `words`: $\forall i \in \{0, \dots, w - 1\} : i < \text{limit} \Leftrightarrow \text{words}[\text{index}[i]] \neq 0$.

Support Bit-Sets. For each variable x_k ($1 \leq k \leq a$) and value $v \in \text{dom}(x_k)$ a *support* bit-set is constructed when the propagator is created, denoted by `supports` $_{\langle x_k, v \rangle}$. It captures the tuples in the table t that are supports for $\langle x_k, v \rangle$: bit i in the support bit-set is set iff $t_{i,k} = v$. The support bit-sets are used to update the sparse bit-set during the filtering phase of the algorithm (discussed below). Note that the support bit-sets are created for each propagator using the same table t with respect to the initial variable domains according to [5], a design that we are going to improve on in Sect. 3.

Update and Filtering. The sparse bit-set and the support bit-sets encode the information necessary to perform the two phases of the algorithm. The *update phase* zeroes the bits in the sparse bit-set that correspond to tuples that have lost support. The *filtering phase* removes values from variable domains that are no longer supported by any tuple.

An optimization in the filtering phase of the algorithm are *residual supports*: for each variable x_k and value $v \in \text{dom}(x_k)$ the word index in the sparse bit-set for which a support for $\langle x_k, v \rangle$ was found is cached.

3 Sharing Tables

Sharing tables among propagators has two aspects: copies of a propagator created during search share tables and propagators using the same table for different constraints (that is, for different variables) share it. As mentioned in Sect. 2, the latter case is not exploited in [5]. Sharing saves memory and increases spatial locality and is likely to improve cache performance. When the table is created, *admissible domains* and *supports* are computed, which are shared among all propagators and their copies using the table.

Admissible Domains. For each position k ($1 \leq k \leq a$) the *admissible domain* is computed as the set of values $d_k = \{t_{i,k} \mid 0 \leq i < n\}$ that occur in a tuple.

The admissible domains only depend on the table and hence can be shared. When a propagator with variables $x_1 \dots, x_a$ and table t is created, the variable domains are constrained to $d_k \cap \text{dom}(x_k)$.

Supports. The table data structure provides shared access to the support bit-sets $\text{supports}_{\langle k,v \rangle}$ for $v \in d_k$. Note the difference from the notation $\text{supports}_{\langle x_k,v \rangle}$ used in Sect. 2, as support bit-sets are based on domains of variables x_k in [5]. All support bit-sets for a position k are stored contiguously in a bit-set array such that $\text{supports}_{\langle k,v_2 \rangle}$ is stored directly after $\text{supports}_{\langle k,v_1 \rangle}$ if value v_2 is the next larger value than v_1 in the admissible domain d_k .

The table should provide constant-time operations to find $\text{supports}_{\langle k,v \rangle}$ for $v \in \text{dom}(x_k)$ or $v \in \Delta_{x_k}$. Here Δ_{x_k} is the *delta* of x_k as the set of values that are removed from $\text{dom}(x_k)$. This is important if deltas are accurate, as for [5], even though it is not discussed there. In [5] the domain implementation relies on sparse bit-sets, which provide cheap access to deltas [11]. Gecode provides only accurate delta information in case the lower or upper bound of a variable changes [7]. Hence, the operations required skip entire ranges of values. Our propagator maintains per variable x_k pointers to $\text{supports}_{\langle k, \min \text{dom}(x_k) \rangle}$ and $\text{supports}_{\langle k, \max \text{dom}(x_k) \rangle}$ which are adjusted by binary search when $\min \text{dom}(x_k)$ or $\max \text{dom}(x_k)$ change. In case no delta information is available, the corresponding support information is computed by simultaneously iterating over variable domains and the supports between the two pointers.

4 Dynamically Compact Sparse Bit-Sets

This section introduces techniques to make sparse bit-sets compact and index structures small that can be adapted dynamically during copying. As in [5], our implementation is a data structure that could also be used in other contexts.

Compact Bit-Sets. Our implementation makes sparse bit-sets compact such that their non-empty words form a contiguous block in memory.

Let us consider part of the algorithm’s update phase. The sparse bit-set with `limit=4` is about to be updated with the shown mask (computed from the support information). For simplicity, we use 4-bit rather than 64-bit words. The update with the mask is performed by word-wise in-place intersection. After the update, `limit` is 2 and the words w_1 and w_3 are empty.

mask	1010	0010	0111	0010
	w_0	w_1	w_2	w_3
words	1101	1000	1011	1001
index	0	1	2	3

The original implementation, called ORIGINAL, executes the following instructions (simplified), where $x \leftarrow \& y$ abbreviates $x \leftarrow x \& y$:

```

for i ← limit - 1 downto 0 do
  words[index[i]] ← & mask[index[i]]
  if words[index[i]] = 0 then
    index[i] ← index[limit - 1]
    limit ← limit - 1
  end
end
end

```

	w_0	w_1	w_2	
words	1000	0000	0011	
index	0	2		

The result is shown to the right, where dead entries (not to be copied) are marked gray. When copying the resulting bit-set the word w_1 would be copied even though it is empty, as it is interleaved with w_0 and w_2 .

Our compact implementation, called COMPACT, updates the data structures by executing the following instructions (simplified) leading to the result shown to the right of the instructions. Note that only w_0 and w_2 need copying.

```

for  $i \leftarrow \text{limit} - 1$  downto 0 do
   $\text{words}[i] \leftarrow \& \text{mask}[i]$ 
  if  $\text{words}[i] = 0$  then
     $\text{index}[i] \leftarrow \text{index}[\text{limit} - 1]$ 
     $\text{words}[i] \leftarrow \text{words}[\text{limit} - 1]$ 
     $\text{limit} \leftarrow \text{limit} - 1$ 
  end
end

```

	w_0	w_2		
words	1000	0011		
index	0	2		

The key benefit of COMPACT is that non-empty words are contiguous in memory as captured by the following invariant:

$$\forall i \in \{0, \dots, \text{limit} - 1\} : \text{words}[i] \neq 0$$

As only non-empty words need copying, both memory usage and time for copying is reduced. The data structure is also more cache-friendly as it is contiguous. COMPACT uses less indirection and hence might be more efficient than ORIGINAL as the words are accessed directly and not through `index`. Even though the removal of an empty word now also requires an update to `words`, these updates are infrequent. Additional changes to the implementation are required, they are analogous. In particular, masks are constructed to be compact to match `words` directly without indirection.

Note that for a trailing solver, rather than *overwriting* `index[i]` and `words[i]` with `index[limit - 1]` and `words[limit - 1]`, these entries would be swapped. Hence the idea of COMPACT is also compatible with a trailing solver.

During the filtering phase of the algorithm, the bit-set is intersected with support bit-sets which are not compact. The original implementation executes instructions of the following form:

$$\text{words}[\text{index}[i]] \& \text{supports}_{\langle k,v \rangle}[\text{index}[i]]$$

while our compact implementation uses less indirection:

$$\text{words}[i] \& \text{supports}_{\langle k,v \rangle}[\text{index}[i]]$$

for position-value pairs $\langle k, v \rangle$ and $0 \leq i < \text{limit}$.

Compressing the Index Structure. We save additional memory for the index structure. When possible, we use 16- or 8-bit data types for the entries in `index` instead of 32-bit; this optimization we refer to as COMPACT++. Consider a table with 16384 tuples, which results in `words` and `index` with 256 entries each. Assuming two pointers for `words` and `index` with 64 bits each, COMPACT++ can

use 8-bit entries and reduce memory usage from 3 092 to 2 321 bytes, a reduction by $\approx 25\%$ (ignoring memory layout requirements).

Two specialized implementations for sufficiently small tables with w words are as follows. `SMALLw` uses 4-bit index entries, that is up to 15 entries and the `limit` field are packed into a single 64-bit word. `DENSEw` drops the entire index structure and considers all words in the bit-set.

Dynamic Data Structures. Making the propagator parametric with respect to its sparse bit-set implementation gives opportunity for further optimization. The decision which implementation to use is made *statically* when the propagator is created or *dynamically* when the propagator is copied. For implementations where only static decisions are made, we use “S” as subscript; for implementations also making dynamic decisions we use “D”. For example, `COMPACT++S` may decide to use 16-bit integers initially and all of its copies also use 16-bit integers, while `COMPACT++D` might create copies using 8-bit integers if possible.

5 Evaluation

We evaluate our implementation of compact-table and the various optimizations on top of Gecode (Version 6.0.0), on the same benchmark set as in [5], involving 1 621 table instances. Being originally in format XCSP 2.1, which is not supported by Gecode, the benchmarks are translated into *MiniZinc* [9] using the tool `xcsp2mzn`¹. Time measurements are run on a Windows 7 (64 bit) computer with two four-core Intel Xeon E5462 of 2.80 GHz and 8 GB RAM. Measurements of memory usage are run on a server cluster. A time out of 1 000 s is used on each instance. We skip instances that (i) cannot be translated to *MiniZinc* due to parse errors (117 instances); (ii) require more than 8 GB of RAM (43 instances); (iii) cannot be solved within the time out for the ORIGINAL configuration (170 instances); or (iv) are solved in less than 1 s for the ORIGINAL configuration (1014 instances). In total, 277 instances are evaluated.

Table 1 shows the relative performance of various implementations, using ORIGINAL as baseline. No implementation except RESIDUES uses residual supports. We report the minimum and maximum relative performance, the geometric mean of the relative performance, as well as the geometric standard deviation of the relative performance. Solvetime means wall time excluding the time for parsing *FlatZinc* and peak heap memory usage also excludes the peak memory for parsing. For presenting the relation of implementation a to a baseline implementation b (ORIGINAL in Table 1), we use the relative measures $100 \cdot \frac{a_i^T}{b_i^T}$ and $100 \cdot \frac{a_i^M}{b_i^M}$, where c_i^T and c_i^M are solvetime respectively peak memory usage for implementation c on instance i . Detailed results are available from the authors.

As shown in Table 1, compressing the bit-set (COMPACT) improves runtime by 14.4% and peak memory usage by 4.5% on average compared to the original implementation (ORIGINAL). The decrease in runtime is most likely achieved

¹ Available at <https://github.com/CP-Unibo/mzn2feat>, last accessed April 17, 2018.

Table 1. Solvetime and peak memory relative to ORIGINAL.

Solvetime	COMPACT	COMPACT++ _S	COMPACT++ _D	BEST _S	BEST _D	RESIDUES
Min	-67.1%	-66.8%	-66.4%	-66.7%	-66.3%	-8.5%
Mean	-14.4%	-14.4%	-13.7%	-14.6%	-13.6%	13.1%
Max	0.4%	1.1%	0.7%	2.2%	0.9%	32.4%
Deviation	±30.8%	±31.1%	±29.7%	±31.0%	±29.6%	±8.3%
Peak memory	COMPACT	COMPACT++ _S	COMPACT++ _D	BEST _S	BEST _D	RESIDUES
Min	-27.2%	-33.4%	-33.4%	-33.4%	-33.2%	-0.0%
Mean	-4.5%	-6.8%	-6.8%	-7.2%	-7.2%	10.0%
Max	0.2%	0.0%	0.0%	-0.3%	-0.3%	71.2%
Deviation	±8.5%	±11.4%	±11.4%	±11.2%	±11.2%	±13.1%

by a combination of (i) fewer operations for copying, (ii) less indirection as argued in Sect. 4, and (iii) better cache performance due to the more compact representation. Additional analysis with the cache profiling tool *Cachegrind* [8] on instances with solvetime of more than 10 seconds (using a time out of one hour), indicates that COMPACT reduces the miss rate of the first-level data cache by $\approx 3\%$ on average compared to ORIGINAL.

Compressing the bit-set as well as the index structure (COMPACT++), further reduces peak memory usage to -6.8% on average. Comparing COMPACT++_S and COMPACT++_D, there is no visible difference in peak memory usage, while the runtime is slightly higher for COMPACT++_D due to extra overhead during copying.

Our proposed winning strategy, BEST, is the combination of COMPACT++ and DENSE₄. This strategy has the best memory usage among all evaluated implementations by a very modest $\approx 0.5\%$ in average while being only marginally slower in average. The winning strategy is chosen from evaluating the specialized implementations SMALL_w for $w \in \{1, 2, 4, 8, 15\}$ and DENSE_w for $w \in \{1, 2, 4, 8, 16\}$ respectively, whose results we omit for lack of space. Overall, these variants perform similarly to each other for all w , though DENSE tends to be slightly faster than SMALL. For simplicity, we use only one of DENSE and SMALL, with DENSE being the winner of the two. The threshold value $w = 4$ is chosen as larger values for w yield slightly higher maximum memory usage.

In Table 1, RESIDUES denotes an implementation that is like ORIGINAL except that it uses residues, discussed in Sect. 2. Clearly, computation saved during propagation by residues do not compensate for the memory and copying overhead they incur, as both runtime and memory usage is increased.

To evaluate the impact of sharing tables between propagators, as discussed in Sect. 3, we extend Gecode’s *FlatZinc* interpreter so that propagators that use the same set of tuples can share the corresponding tables. Measurements are made with and without sharing for BEST_D, and on average runtime is reduced by 4.6% and memory usage by 56.5% when using sharing.

Note that the comparison is slightly approximate in the sense that it does not only reflect sharing of the admissible domains and the support bit-sets, but also the actual tuples, as the entire tables are duplicated in the version without sharing (even though the actual tuples are never used during propagation). That being said, as the runtime measurements do not include *FlatZinc* overhead, during which the tables are created, and as tables are not copied, the runtime measurements are likely to reflect actual difference from sharing admissible domains and support bit-sets. Analysis with *Cachegrind* indicates that sharing tables reduces the miss rate of the first-level data cache by $\approx 18\%$ on average on sufficiently time-consuming instances.

We compare the performance of BEST_D with the two original implementations of table constraints in Gecode, based on [3] and [4]. The comparison is without sharing tables in *FlatZinc* as this is not available for the two implementations. Our implementation reduces runtime on average by 85.7% and up to 99.6% and reduces peak memory on average by 45.4% and up to 67.7% compared to the best of Gecode’s two implementations. Note that these numbers are based on fewer instances than the other studied configurations, as the best old implementation timed out on 43 additional instances.

6 Conclusions and Future Work

This paper shows that compact-table is an excellent fit for a copying solver; the algorithm runs on average almost an order of magnitude faster than Gecode’s implementations while using only half the memory. Our proposed implementation of compact-table is more compact in memory than the original implementation, reducing peak memory usage by 7.2% and solvetime by 13.6% on average on our benchmark set. We introduce how to share tables among propagators and demonstrate that sharing moderately reduces solvetime by 4.6% and considerably reduces memory usage by 56.5%. A trailing solver can most likely benefit from our optimizations as well: support bit-sets can be shared; and it might be beneficial to make sparse bit-sets compact as it is better for cache performance and uses less indirection.

Future Work. Our implementation only uses approximate information about variable deltas; more accurate information maintained by the propagator might speed up propagation. Heuristics for re-ordering the tuples in the table have not been explored. Re-ordering can have an impact on how much the sparse bit-sets can be compressed, as the order in which the tuples appear in the table decides which bits become zero first during propagation.

Acknowledgments. We are grateful for numerous comments and assistance from Mats Carlsson and Roberto Castañeda Lozano. Part of the work has been carried out in the first author’s bachelor thesis [6] and during her student internship at KTH. We are grateful for the helpful comments from the anonymous reviewers.

References

1. Beck, J.C. (ed.): CP 2017. LNCS, vol. 10416. Springer, Cham (2017). <https://doi.org/10.1007/978-3-319-66158-2>
2. Bessière, C. (ed.): CP 2007. LNCS, vol. 4741. Springer, Heidelberg (2007). <https://doi.org/10.1007/978-3-540-74970-7>
3. Bessière, C., Régin, J.C.: Arc consistency for general constraint networks: preliminary results. In: International Joint Conference on Artificial Intelligence (IJCAI), vol. 1, Nagoya, Japan, pp. 398–404, August 1997
4. Bessière, C., Régin, J.C., Yap, R.H.C., Zhang, Y.: An optimal coarse-grained arc consistency algorithm. *Artif. Intell.* **165**(2), 165–185 (2005)
5. Demeulenaere, J., et al.: Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In: Rueher, M. (ed.) CP 2016. LNCS, vol. 9892, pp. 207–223. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44953-1_14
6. Ingmar, L.: Implementation and evaluation of a compact-table propagator in Gecode. Bachelor thesis, Department of Information Technology, Uppsala University, Sweden, August 2017. <http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-328679>
7. Lagerkvist, M.Z., Schulte, C.: Advisors for incremental propagation. In: Bessière [2], pp. 409–422
8. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Ferrante, J., McKinley, K.S. (eds.) Conference on Programming Language Design and Implementation (PLDI), pp. 89–100. ACM, San Diego (2007)
9. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessière [2], pp. 529–543
10. Reischuk, R.M., Schulte, C., Stuckey, P.J., Tack, G.: Maintaining state in propagation solvers. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 692–706. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_54
11. de Saint-Marcq, V.L.C., Schaus, P., Solnon, C., Lecoutre, C.: Sparse-sets for domain implementation. In: CP Workshop on Techniques for Implementing Constraint Programming Systems (TRICS), pp. 1–10, Uppsala, Sweden, September 2013
12. Schaus, P., Aoga, J.O.R., Guns, T.: CoverSize: a global constraint for frequency-based itemset mining. In: Beck [1], pp. 529–546
13. Schulte, C.: Comparing trailing and copying for constraint programming. In: De Schreye, D. (ed.) International Conference on Logic Programming, pp. 275–289. The MIT Press, Las Cruces (1999)
14. Verhaeghe, H., Lecoutre, C., Deville, Y., Schaus, P.: Extending compact-table to basic smart tables. In: Beck [1], pp. 297–307
15. Verhaeghe, H., Lecoutre, C., Schaus, P.: Extending compact-table to negative and short tables. In: Singh, S.P., Markovitch, S. (eds.) AAAI Conference on Artificial Intelligence, San Francisco, CA, USA, pp. 3951–3957, February 2017



Approximation Strategies for Incomplete MaxSAT

Saurabh Joshi¹(✉), Prateek Kumar¹, Ruben Martins², and Sukrut Rao¹

¹ Indian Institute of Technology Hyderabad, Sangareddy, India
{sbjoshi, cs15btech11031, cs15btech11036}@iith.ac.in

² Carnegie Mellon University, Pittsburgh, USA
rubenm@andrew.cmu.edu

Abstract. Incomplete MaxSAT solving aims to quickly find a solution that attempts to minimize the sum of the weights of the unsatisfied soft clauses without providing any optimality guarantees. In this paper, we propose two approximation strategies for improving incomplete MaxSAT solving. In one of the strategies, we cluster the weights and approximate them with a representative weight. In another strategy, we break up the problem of minimizing the sum of weights of unsatisfiable clauses into multiple minimization subproblems. Experimental results show that approximation strategies can be used to find better solutions than the best incomplete solvers in the MaxSAT Evaluation 2017.

Keywords: MaxSAT · Incomplete · Approximation

1 Introduction

Given a set of Boolean constraints in a conjunctive normal form (CNF), the problem of Maximum Satisfiability (MaxSAT) asks to provide valuation of variables so that maximum number of constraints are satisfied. These constraints can be assigned weights to prioritize some set of constraints over others, which would give rise to a weighted MaxSAT problem where the goal is to find a valuation which maximizes the sum of the weights of the satisfied constraints. Any improvements in MaxSAT solving have a huge impact because many real world problems can be encoded as MaxSAT problems (e.g., [5, 14, 16]).

Often, the application may be able to tolerate a suboptimal solution but requires this solution to be computed in a very short amount of time. For such cases, it tremendously helps if there are techniques and tools that can very quickly find a solution which is close enough to an optimal solution. Incomplete MaxSAT solvers [4, 9, 10, 19, 20, 26] strive to find a good solution in a limited time frame. The solution, thus provided, need not be an optimal one. Therefore, for improvement, we need to develop tools and techniques that can find better solutions (closer to an optimal solution) in the same time frame.

As part of this paper, we contribute the following:

- An approximation strategy based on weight relaxation (Sect. 3), which modifies the weights of the clauses in a manner so that it is easier for the solver to find a solution quickly.
- An approximation strategy which breaks up the problem of minimizing the sum of weights of unsatisfied clauses into multiple minimization subproblems and attempts to minimize these subproblems in a greedy order (Sect. 3). This strategy can also be combined with the weight relaxation strategy.
- Empirical results on how the accuracy of the solver gets affected as we vary the weight relaxation parameter (Sect. 5).
- An implementation of these strategies using the Open-WBO framework. We also demonstrate the advantage of these approximation strategies by showing its prowess against state-of-the-art incomplete MaxSAT solvers (Sect. 5).

2 Preliminaries

Let x be a Boolean variable which can take values *true* or *false*. A literal l is a variable x or its negation $\neg x$. A clause ω is a disjunction of literals and a formula φ is a conjunction of clauses. Notationally, we will treat a clause ω and a formula φ as sets containing literals and clauses respectively.

An assignment ν maps variables to either *true* or *false*. An assignment is said to satisfy a positive literal x (resp. a negative literal $\neg x$) if $\nu(x) = \textit{true}$ (resp. $\nu(x) = \textit{false}$). A clause is said to be satisfied if at least one of its literals is satisfied. A formula is said to be satisfied by an assignment if all of its clauses are satisfied by the assignment. A formula is called *satisfiable* if there exists a satisfying assignment for that formula, otherwise it is called *unsatisfiable*. Boolean satisfiability problem (SAT) asks to find a satisfying assignment (i.e., model) to a formula. Maximum satisfiability (MaxSAT) problem is an optimization version where the goal is to find an assignment which satisfies the maximum number of clauses of a formula. In a partial MaxSAT problem, a partition of φ is given as two mutually exclusive sets φ_h (*hard* clauses) and φ_s (*soft* clauses), where the goal is to satisfy all the clauses in φ_h ¹ while maximizing the number of clauses satisfied in φ_s . Let *weight* : *Clauses* $\rightarrow \mathbb{N}^+$ be a map from a set of clauses to positive integers. In a partial weighted MaxSAT problem, the goal is to find an assignment that maximizes the sum of weights of the satisfied soft clauses. From now on, we will refer to a weighted partial MaxSAT problem as MaxSAT.

A clause ω can be relaxed by adding a relaxation variable r so that the relaxed clause becomes $\omega \cup \{r\}$. The relaxed clause can be satisfied by either satisfying the original clause or its relaxation variable. For a formula φ , when all of its soft clauses are relaxed, we will denote it as φ^r . We define the cost of a relaxation variable r to be the weight of the clause that it relaxed, $\textit{cost}(r) = \textit{weight}(\omega)$. The cost of an assignment ν is defined as $\textit{cost}(\nu) = \sum_{r_i: \nu(r_i)=1} \textit{cost}(r_i)$. The goal of MaxSAT is to find a satisfying assignment with the minimum cost.

¹ For simplicity, we will assume that φ_h is always satisfiable.

Input : Formula φ_s , Map *weight*, partitioning parameter m
Output: Partition $P(m)$, new weight map *weight_m*

```

1  $n \leftarrow |\varphi_s|$ 
2 sort clauses of  $\varphi_s$  in the ascending order of weights
3 for  $i \leftarrow 1$  to  $n - 1$  do
4    $diff_i \leftarrow weight(\omega_{i+1}) - weight(\omega_i)$ 
5  $\langle i_1, \dots, i_{m-1} \rangle \leftarrow$  sorted indices where top  $(m - 1)$  difference  $diff_i$  occurs
6  $c_1 \leftarrow \{\omega_1, \dots, \omega_{i_1}\}$ 
7 for  $j \leftarrow 2$  to  $m - 1$  do
8    $c_j \leftarrow \{\omega_{i_{j-1}+1}, \dots, \omega_{i_j}\}$ 
9  $c_m \leftarrow \{\omega_{i_{m-1}+1}, \dots, \omega_n\}$ 
10  $P(m) \leftarrow \{c_1, \dots, c_m\}$ 
11 foreach  $c_i \in P(m)$  do
12   foreach  $\omega_j \in c_i$  do
13      $weight_m[\omega_j] \leftarrow RepresentativeWeight(c_i)$ 
14 return  $\langle P(m), weight_m \rangle$ 
    
```

Algorithm 1. Partitioning and weight approximation

Input: Formula φ^r , weight maps *weight_m*, and *weight*
Output: model to φ

```

1 (model,  $\mu$ ,  $\varphi_W$ )  $\leftarrow$  ( $\emptyset$ ,  $+\infty$ ,  $\varphi^r$ )
2 status = SAT
3 while status = SAT do
4   (status,  $\nu$ )  $\leftarrow$  SAT( $\varphi_W$ )
5   if status = SAT then
6     if  $cost(\nu) < cost(model)$  then
7       model  $\leftarrow$   $\nu$ 
8        $\mu \leftarrow cost_m(\nu)$ 
9        $\varphi_W \leftarrow \varphi_W \cup \{CNF((\sum_{r \in V_R} (cost_m(r) \cdot r) \leq \mu - 1))\}$ 
10 return model
    
```

Algorithm 2. Linear search Sat-Unsat algorithm for MaxSAT

3 Approximation Strategies

In this section, we describe two approximation strategies that can allow MaxSAT algorithms to converge faster to lower cost solutions. Note that the best model found by approximation strategies is not guaranteed to be an optimal solution of the original MaxSAT formula.

Weight-Based Approximation. Let $P_m(\varphi_s) = \{c_1, \dots, c_m\}$ be a partition of φ_s into m mutually exclusive sets c_1, \dots, c_m such that $\bigcup_{1 \leq i \leq m} c_i = \varphi_s$ and $\forall_{i \neq j} : c_i \cap c_j = \emptyset$. We will call sets c_1, \dots, c_m as clusters of the partition.

Given a formula φ_s , Algorithm 1 partitions the clauses into clusters as follows. All soft clauses are sorted by their weights (Line 2). Then, differences in weights between two consecutive clauses are calculated (Line 4). $m - 1$ indices are picked where the weight differences are amongst the top $m - 1$ weight differences (Line 5). These indices are used as boundaries to create clusters (Lines 6–9). This way of clustering is similar to single-link agglomerative clustering [15]. Finally, a new weight map *weight_m* is created, where all the clauses in the same cluster get the same weight (Lines 11–13). *RepresentativeWeight* (Line 13) indicates any representative weight for the cluster. In this paper, we use the *arithmetic mean* of the weights of the clauses in a cluster as the representative weight. In principle, other representative weights can also be chosen which may have different effect on

Input: $\varphi = \varphi_h \cup \varphi_s$, weight maps $weight$ and $weight_m$, Partition $P(m)$
Output: model to φ

```

1 (model,  $\mu$ ,  $\varphi_W$ ,  $\mathcal{C}$ )  $\leftarrow$   $(\emptyset, +\infty, \varphi^r, P_m(\varphi_s))$ 
2 foreach  $c_i \in \mathcal{C}$  in the descending order of  $weight_m(c_i)$  do
3    $V_i \leftarrow V_R \cap c_i$ 
4   status = SAT
5   while status = SAT do
6     (status,  $\nu$ )  $\leftarrow$  SAT( $\varphi_W$ )
7     if status = SAT then
8       if  $cost(\nu) < cost(model)$  then
9         model  $\leftarrow$   $\nu$ 
10         $\mu_i \leftarrow |\{r \in V_i \mid \nu(r) = 1\}|$ 
11         $\varphi_W \leftarrow \varphi_W \cup \{\text{CNF}(\sum_{r \in V_i} r \leq \mu_i - 1)\}$ 
12      else
13         $\varphi_W \leftarrow \varphi^r$ 
14        foreach  $c_j \in \mathcal{C}$  such that  $weight_m(c_j) \geq weight_m(c_i)$  do
15           $\varphi_W \leftarrow \varphi_W \cup \{\text{CNF}(\sum_{r \in V_j} r \leq \mu_j)\}$ 
16 return model

```

Algorithm 3. Clustering-based algorithm for MaxSAT

how much an algorithm can deviate from finding the minimum cost assignment. It is redundant to have $m > \#weights$, where $\#weights$ are different number of weights, because for $m \geq \#weights$, $weight = weight_m$. Algorithm 1 can be combined with any search algorithm and as m increases the deviation of the search algorithm from an optimal solution decreases. If $m = 0$ it is assumed that no partitioning is done.

There are encodings which perform better when $\#weights$ is small [12, 17]. Such encodings can benefit from approximation of weights because it results in a smaller size formula when converted to CNF. This can be used with a cost minimization algorithm for MaxSAT such as the linear search Sat-Unsat algorithm [8, 18] shown in Algorithm 2. In this algorithm, all the clauses in φ_s are initially relaxed, and the set of corresponding relaxation variables is denoted as V_R . A working formula φ_W is initialized with the relaxed formula φ^r . The cost of an empty model is assumed to be $+\infty$. Our primary goal is to find a satisfying assignment ν to φ with the minimum $cost(\nu)$. Algorithm 2 iteratively asks a SAT solver if there is a satisfying assignment, with its cost at most $\mu - 1$ (Line 9). The approximation comes from Algorithm 2 using $cost_m$ instead of $cost$ to encode a pseudo-Boolean (PB) constraint that restricts the cost of relaxation variables being set to *true* (Line 9). Since $cost_m$ is an approximation of $cost$, minimizing $cost_m$ does not necessarily translate to minimization w.r.t. $cost$. Therefore, we update model, only when a satisfying assignment indeed reduces the previous value of $cost(model)$ (Lines 6–7).

Approximation via Subproblem Minimization. Algorithm 3 proceeds in a greedy manner by processing each cluster in the descending order of its representative weight (Line 2). V_i indicates a set of relaxation variables corresponding to the clauses in c_i (Line 3). Minimization of the cost of a satisfying assignment is divided in subproblems by minimizing the number of unsatisfied clauses in clusters, starting from highest representative weight to the lowest (Line 2). For each cluster, the number of unsatisfied clauses are minimized by iteratively

reducing the upper bound μ_i on the number of relaxation variables in V_i that can be set to *true* (Lines 10–11). In the process, the minimum cost assignment seen so far is recorded (Lines 8–9). Since within any cluster, all the clauses have the same *weight_m*, only cardinality constraints are used to restrict the number of unsatisfied clauses within μ_i (Line 11). Once μ_i can not be reduced further, it is frozen by adding upper bound μ_i for all the cluster seen so far (Lines 12–15). Since the minimization is done locally as a minimization subproblem at a cluster level, rather than looking at the whole formula, this procedure is not guaranteed to converge to a globally optimum solution.

4 Related Work

Approaches for incomplete MaxSAT solving can primarily be divided into two categories: (i) stochastic MaxSAT solvers [9, 10, 13, 19, 20] and (ii) complete MaxSAT solvers that can find intermediate solutions [4, 8, 11, 18, 23, 25].

Incomplete MaxSAT. Stochastic solvers start by finding a random assignment ν for φ . Since this assignment is unlikely to satisfy all clauses in φ , they choose a clause ω_i that is unsatisfied by ν and flip the assignment of a variable in ω_i such that ω_i becomes satisfied. When compared to local SAT solvers, stochastic MaxSAT solvers have additional challenges since they must find an assignment ν that satisfies φ_h while attempting to minimize the cost of the unsatisfied soft clauses. Stochastic MaxSAT solvers are particularly effective for random benchmarks but their performance tends to deteriorate for industrial benchmarks. Since the MaxSAT Evaluation 2017 (MSE2017) [2] did not contain any random instances, there were no stochastic MaxSAT solvers in the MSE2017.

Complete MaxSAT. Complete solvers can often find intermediate solutions to φ before finding an optimal assignment ν . MaxSAT solvers based on linear search algorithms [8, 18, 23] can find a sequence of intermediate solutions that converge to an optimal solution. These solvers use PB constraints to enforce convergence. While SAT4J [8] uses specialized data structures for PB constraints to avoid their conversion to CNF, other solvers such as QMaxSAT [18] convert the PB constraint into clauses using PB encodings [12, 17, 27]. Some MaxSAT solvers which are based on the implicit hitting set approach [11, 25] maintain a lower and an upper bound on the values of the solution. These solvers can also be used for incomplete MaxSAT since they are also able to find intermediate solutions. Another approach for complete MaxSAT solving is to use unsatisfiability-based algorithms [1, 4, 24]. These algorithms use unsatisfiable subformulas to increase a lower bound on the cost of a solution until they find an optimal solution. For weighted MaxSAT, these algorithms employ a stratified approach [3] where they start by considering only a subset of the soft clauses with the largest weights and iteratively add more soft clauses when the subformula becomes satisfiable. An intermediate solution is found at each iteration. WPM3 [4] is an example of an unsatisfiability-based solver that can be used for incomplete MaxSAT and was the best incomplete MaxSAT solver in the MSE2016 [6]. maxroster [26] was the

winner of the incomplete track for Weighted MaxSAT in the MSE2017. It is a hybrid solver that combines an initial short phase of a stochastic algorithm [13] with complete MaxSAT algorithms [18, 24].

Boolean Multilevel Optimization. The clustering-based algorithm presented in Algorithm 3 is closely related to Boolean Multilevel Optimization (BMO) [21]. BMO is a technique for identifying lexicographic optimization conditions, i.e. the existence of an ordered sequence of objective functions. Let M_i be the minimum weight of soft clauses in a cluster c_i . Consider a sequence of clusters c_1, \dots, c_m arranged in a descending order of M_i . A MaxSAT formula is an instance of BMO if for every cluster c_i , M_i is larger than the sum of the weights of all soft clauses in clusters c_{i+1}, \dots, c_m . If this condition holds then the result of Algorithm 3 is equivalent to solving a BMO formula. However, when using the proposed clustering-based algorithm on partitions that do not preserve the BMO condition, it is not guaranteed that the solution found by Algorithm 3 is an optimal solution for φ . Our approach differs from previous complete approaches in using approximation strategies that do not preserve optimality but are more likely to converge faster to a better solution.

5 Experimental Results

To evaluate incomplete MaxSAT solvers we used the scoring mechanism from MaxSAT Evaluations 2017 (MSE2017) [2]. Given a formula φ , the score for a solver \mathcal{S} is computed by the ratio of the cost (sum of weights of unsatisfied clauses) of the best solution known for φ , denoted as $best(\varphi)$,² to the best cost found by \mathcal{S} , denoted as $cost^{\mathcal{S}}(\varphi)$.³ The score for \mathcal{S} for a set of n benchmarks is given by the average score $([0, 1])$ as follows:

$$\text{score}(\mathcal{S}) = \frac{\sum_{i=1}^n \frac{best(\varphi_i)}{cost^{\mathcal{S}}(\varphi_i)}}{n} \quad (1)$$

$\text{score}(\mathcal{S})$ shows how close on average is a solver \mathcal{S} to the best known solution.

All the experiments were conducted on Intel[®] Xeon[®] E5-2620 v4 processors with a memory limit of 32 GB and time limits of 10, 60 and 300 s. We have used a non-standard timeout of 10 s to show that approximation strategies can find good solutions very quickly. We used the 156 benchmarks for incomplete MaxSAT from MSE2017 [2]. Note that most of these benchmarks are challenging for complete solvers and have unknown optimal solutions.

We have implemented all the algorithms presented in this paper in OpenWBO-Inc. OpenWBO-Inc is built on top of OpenWBO [23] which uses Glucose [7] as the underlying SAT solver. We used Generalized Totalizer Encoding (GTE) [17] and incremental Totalizer encoding [22] to translate PB constraints and cardinality constraints into CNF, respectively.

² $best(\varphi)$ is the cost of the best solution found by any solver in this evaluation.

³ We consider a score of 0 if \mathcal{S} did not find any solution to φ .

We evaluated **Open-WBO-Inc** by conducting experiments that are designed to answer the following questions: (1) What is the impact of the number of clusters in the quality of the solution found by our approximation strategies? (2) How does **Open-WBO-Inc** compare against state-of-the-art incomplete MaxSAT solvers?

Impact of the Number of Clusters. We measure the impact of partitioning parameter m on the accuracy of the results. Figure 1a shows the score of Algorithm 2 with GTE encoding (henceforth called **apx-weight**). Figure 1a shows that **apx-weight** performs the worst when no partitioning is done. This is attributed to the fact that in the absence of any partitioning, the size of the underlying encoding is dictated by $\#weights$, where $\#weights$ are the number of different weights in the weight map. Figure 1c shows a measure of increase in formula size as m varies. The Y-axis shows the ratio of the formula size after the PB encoding to the size of the original input formula. Because of the weight-based approximation, the reduction on the $\#weights$ leads to a smaller encoding, thus making it easier for the underlying SAT solver. As m increases, the possible deviation from an optimal cost also decreases, thereby resulting in increased scores. The degradation for larger m is attributed to larger size of the formula. As the timeout is increased, the score increases because Algorithm 2 has more time and can do more iterations to reduce $cost_m(\text{model})$.

Figure 1b shows that similar scoring trends are witnessed for Algorithm 3 (henceforth called **apx-subprob**). As **apx-subprob** uses only cardinality constraints, the formula size is not much sensitive to m . As m increases, the scores also improve, with the best scores achieved when $m = \#weights$. **apx-subprob** is guaranteed to find optimal solution only if BMO condition holds and $m = \#weights$. Only 3 out of 156 benchmarks have the BMO condition and **apx-subprob** with $m = \#weights$ does not terminate for any of them. However, **apx-subprob** using a 300s time limit terminates for 94 out of 156 benchmarks which shows that **apx-subprob** quickly finds a good solution.

Comparison Against State-of-the-Art MaxSAT Solvers. We compared the best version of **Open-WBO-Inc** for weight-based approximation, **apx-weight** with $m = 2$, and subproblem minimization approximation, **apx-subprob** with $m = \#weights$, with **maxroster** [26], **WPM3** [4] and **QMaxSAT** [18]. **maxroster** and **WPM3** were the winners of the incomplete weighted category of the MSE2017 and MSE2016, respectively. **QMaxSAT** was placed second on the complete category of the MSE2017 and uses the algorithm described in Algorithm 2.⁴

As shown in Fig. 1d, for a 10s timeout, both **apx-weight** and **apx-subprob** perform better than all the other solvers with **apx-subprob** performing the best. This demonstrates that approximation strategies are quite effective when we want to quickly find a solution which is closer to an optimal solution. For 60 and 300s timeout, **apx-subprob** performs the best with **maxroster** being second and **apx-weight** outperforming **WPM3** and **QMaxSAT**. Even though **apx-weight**

⁴ Even though **MaxHS** [11] placed first in the complete weighted category of the MSE2017, its incomplete version is not as competitive as the other solvers [2].

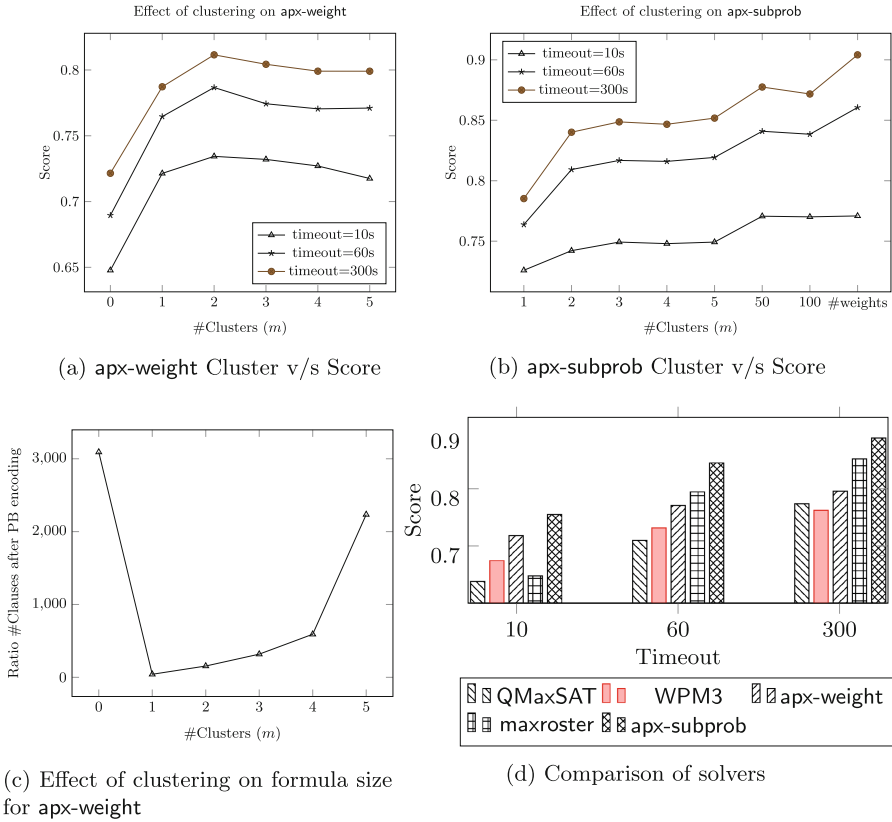


Fig. 1. Impact of clustering and comparison against state-of-the-art

with $m = 0$ performs worse than QMaxSAT, it outperforms QMaxSAT when clustering is used. **apx-subprob** outperforms all other solvers and these results prove the efficacy of approximation strategies with respect to the state-of-the-art in incomplete MaxSAT solving.

6 Conclusion and Future Work

Approximation strategies, be it weight-based relaxation or subproblem minimization, are not guaranteed to find an optimal solution even when unlimited time is given. However, they serve the purpose of quickly finding a good solution. Our experiments have successfully demonstrated that with the right parameters, these strategies can outperform the best incomplete solvers. In future, we would like to explore the application of approximation strategies to complete algorithms. In particular, progressively increasing the number of clusters and using approximation strategies to find good initial upper bounds that can later be exploited by complete MaxSAT algorithms.

Acknowledgements. This work is partially funded by ECR 2017 grant from SERB, DST, India, NSF award #1762363 and CMU/AIR/0022/2017 grant. Authors would like to thank the anonymous reviewers for their helpful comments, and Saketha Nath for lending his servers for the experiments.

References

1. Alviano, M., Dodaro, C., Ricca, F.: A MaxSAT algorithm using cardinality constraints of bounded size. In: Proceedings of International Joint Conference on Artificial Intelligence, pp. 2677–2683. AAAI Press (2015)
2. Ansótegui, C., Bacchus, F., Järvisalo, M., Martins, R.: MaxSAT Evaluation 2017 (2017). <http://mse17.cs.helsinki.fi/>. Accessed 18 Apr 2017
3. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving SAT-based weighted MaxSAT solvers. In: Milano, M. (ed.) CP 2012. LNCS, pp. 86–101. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33558-7_9
4. Ansótegui, C., Gabàs, J.: WPM3: an (in)complete algorithm for weighted partial MaxSAT. *Artif. Intell.* **250**, 37–57 (2017)
5. Argelich, J., Le Berre, D., Lynce, I., Marques-Silva, J., Rapicault, P.: Solving linux upgradeability problems using Boolean optimization. In: Proceedings of Workshop on Logics for Component Configuration, pp. 11–22. EPTCS (2010)
6. Argelich, J., Li, C.M., Manyà, F., Planes, J.: MaxSAT Evaluation 2016 (2016). <http://maxsat.ia.udl.cat/>. Accessed 18 Apr 2016
7. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Proceedings of International Joint Conference on Artificial Intelligence, pp. 399–404. AAAI Press (2009)
8. Le Berre, D., Parrain, A.: The SAT4J library, release 2.2. *JSAT* **7**(2–3), 59–64 (2010)
9. Cai, S., Luo, C., Thornton, J., Su, K.: Tailoring local search for partial MaxSat. In: Proceedings of AAAI Conference on Artificial Intelligence, pp. 2623–2629. AAAI Press (2014)
10. Cai, S., Luo, C., Zhang, H., From decimation to local search and back: a new approach to MaxSAT. In: Proceedings of AAAI Conference on Artificial Intelligence, pp. 571–577. AAAI Press (2017)
11. Davies, J., Bacchus, F.: Solving MAXSAT by solving a sequence of simpler SAT instances. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 225–239. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23786-7_19
12. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. *JSAT* **2**(1–4), 1–26 (2006)
13. Fan, Y., Ma, Z., Kaile, S., Sattar, A., Li, C.: Ramp: a local search solver based on make-positive variables. In: Proceedings of MaxSAT Evaluation (2016)
14. Feng, Y., Bastani, O., Martins, R., Dillig, I., Anand, S.: Automated synthesis of semantic malware signatures using maximum satisfiability. In: Proceedings of Network and Distributed System Security Symposium (2017)
15. Johnson, S.C.: Hierarchical clustering schemes. *Psychometrika* **32**(3), 241–254 (1967)
16. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Proceedings of Conference on Programming Language Design and Implementation, pp. 437–446. ACM (2011)

17. Joshi, S., Martins, R., Manquinho, V.: Generalized totalizer encoding for pseudo-Boolean constraints. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 200–209. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_15
18. Koshimura, M., Zhang, T., Fujita, H., Hasegawa, R.: QMaxSAT: a partial MaxSAT solver. JSAT **8**(1/2), 95–100 (2012)
19. Luo, C., Cai, S., Kaile, S., Huang, W.: CCEHC: an efficient local search algorithm for weighted partial maximum satisfiability. Artif. Intell. **243**, 26–44 (2017)
20. Luo, C., Cai, S., Wei, W., Jie, Z., Kaile, S.: CCLS: an efficient local search algorithm for weighted maximum satisfiability. IEEE Trans. Comput. **64**(7), 1830–1843 (2015)
21. Marques-Silva, J., Argelich, J., Graça, A., Lynce, I.: Boolean lexicographic optimization: algorithms & applications. Ann. Math. Artif. Intell. **62**(3–4), 317–343 (2011)
22. Martins, R., Joshi, S., Manquinho, V., Lynce, I.: Incremental Cardinality Constraints for MaxSAT. In: O’Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 531–548. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_39
23. Martins, R., Manquinho, V., Lynce, I.: Open-WBO: a modular MaxSAT solver’. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 438–445. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_33
24. Morgado, A., Dodaro, C., Marques-Silva, J.: Core-guided MaxSAT with soft cardinality constraints. In: O’Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 564–573. Springer, Cham (2014)
25. Saikko, P., Berg, J., Järvisalo, M.: LMHS: a SAT-IP hybrid MaxSAT solver. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 539–546. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_34
26. Sugawara, T.: MaxRoster: solver description. In: Proceedings MaxSAT Evaluation 2017: Solver and Benchmark Descriptions, vol. B-2017-2, p. 12. University of Helsinki, Department of Computer Science (2017)
27. Warners, J.P.: A linear-time transformation of linear inequalities into conjunctive normal form. Inf. Process. Lett. **68**(2), 63–69 (1998)



A Novel Graph-Based Heuristic Approach for Solving Sport Scheduling Problem

Meriem Khelifa^{1(✉)}, Dalila Boughaci^{1(✉)}, and Esmâ Aïmeur²

¹ Department of Computer Science-Laboratory of Research in Artificial Intelligence
LRIA, USTHB, Bab Ezzouar, Algeria

khalifa.merieme.lmd@gmail.com, dboughaci@usthb.dz

² Department of Computer Science and Operations Research HERON Laboratory,
Montreal, Canada

aimeur@iro.unmontreal.ca

Abstract. This paper proposes an original and effective heuristic approach for solving the unconstrained traveling tournament problem (denoted by UTTP) in sport scheduling. UTTP is an interesting variant of the well-known NP-hard traveling tournament problem (TTP) where the main objective is to find a tournament schedule that minimizes the total distances traveled by the teams. The proposed graph-based heuristic method starts with a set of n teams ($n < 10$). The method models the problem by representing the home locations of the teams as vertices and each arc corresponds to the matching between two teams. Each round corresponds to a 1-factor of the generated graph. We use the Bron-Kerbosch clique detection algorithm to enumerate all the possible $2(n-1)$ cliques from the 1-factors. Then, the vertices of each $2(n-1)$ cliques are sorted to create double round robin tournament (DRRT) schedules. The schedule with lowest cost travel is selected to be the solution of the problem. The proposed method is evaluated on several instances and compared with the state-of-the-art. The numerical results are promising and show the benefits of our method. The proposed method significantly improves the current best solutions for the US National Baseball League (NL) instances and produces new good solutions for the Rugby League (SUPER) instances.

Keywords: Sport scheduling · Traveling tournament problem
Heuristic · Perfect matching · Graph
Unconstrained traveling tournament problem

1 Introduction

Sports scheduling is an active and important research area that can be useful to real sports leagues [11, 17, 18]. This work focus on the traveling tournament problem (TTP) [8, 12, 19]. The traveling tournament problem (TTP) is a core problem in sports scheduling where the aim is to find a double round robin tournament

(DRRT) schedule for a finite number of teams located at various geographical locations with a finite distance between them [8].

Most association sport leagues in the world can be organized in a double round robin. In a DRRT schedule, all teams $T = (t_1, t_2 \dots t_n)$ meet all other teams twice where n is a positive even integer. Every team plays against each other team exactly once at home and once away, and all teams must play only one match every round. Consequently, a DRRT has $n(n - 1)$ games, and $n/2$ games are played in every round. Thus exactly $2(n - 1)$ rounds are required to schedule a DRRT.

TTP is a difficult problem to solve, and it is known to be NP-hard [22]. In this work, we are interested in the unconstrained traveling tournament problem (UTTP) which is a variant of TTP. UTTP is also an NP-hard problem [3].

In this paper, we propose an effective heuristic for UTTP. We propose a graph-based heuristic able to yield promising results for a number of teams $n < 10$. We use as a model a complete graph $G_T(H, E)$ where the set $H = \{h_i\}_{i=1}^n$ of the vertices represents the home of the team t_i , and the set of edges E , are weighted by the distance $dis(h_i, h_j)$ between the home of teams t_i and t_j . Our method is based on the perfect matching of graphs of rounds where the input is the number of the teams ($n < 10$) and the output is a DRRT schedule. The proposed method produces new good solutions on some unsolved UTTP instances. A significant improvement is noted over previous approaches for National League United States (NL) instances.

The rest of this paper is organized as follows. Section 2 gives a background on the TTP problem. Section 3 presents some related works. Section 4 details the proposed approach. Section 5 gives the numerical results and the comparison of our results with the best ones existing in the literature. Finally, Sect. 6 concludes and gives some perspectives.

2 Problem Definition

TTP is the problem of scheduling a double round-robin tournament while satisfying a set of related constraints and minimizing the total distance traveled by all the teams. The TTP can be stated as follows:

- **The TTP inputs:** include a set of n teams $T = (t_1, t_2 \dots t_n)$ and a symmetric n by n integer distance matrix that represents the distance between team sites. We denote this matrix by dis .
- **The output:** is a DRRT of $2(n - 1)$ rounds that minimizes the total traveled distance of the teams with the following constraints:

1. No team should play more than 3 consecutive home or away games.
2. A team cannot play against the same opponent in two consecutive games.

The objective of the TTP is to find a schedule with the minimum cost satisfying the two above constraints. We note that the cost of a schedule is the sum of

the cost of every team where the cost of team t_i is equal to the total distance traveled by this team.

UTTP is a variant of TTP in which constraints 1 and 2 are eliminated. The objective of the UTTP is the same of TTP; find a schedule with minimum cost where the constraints 1 and 2 are not necessarily required. UTTP can be a suitable model for some practical scheduling problems.

Moreover, eliminating both constraints decreases the budget constraint. Thus a solution for UTTP is an appropriate model for managing a sports league with low budget.

Example: Table 1 provides an example of a TTP schedule for $n = 6$.

Table 1. Double round robin tournament (DRRT) ($n = 6$)

R_1	(t_6, t_4)	(t_2, t_3)	$(\mathbf{t}_5, \mathbf{t}_1)$	R_6	(t_2, t_6)	$(\mathbf{t}_4, \mathbf{t}_1)$	(t_3, t_5)
R_2	(t_5, t_6)	$(\mathbf{t}_3, \mathbf{t}_1)$	(t_2, t_4)	R_7	(t_3, t_6)	(t_5, t_4)	$(\mathbf{t}_2, \mathbf{t}_1)$
R_3	$(\mathbf{t}_1, \mathbf{t}_6)$	(t_5, t_2)	(t_3, t_4)	R_8	(t_6, t_2)	$(\mathbf{t}_1, \mathbf{t}_4)$	(t_5, t_3)
R_4	(t_6, t_3)	(t_4, t_5)	$(\mathbf{t}_1, \mathbf{t}_2)$	R_9	(t_6, t_5)	$(\mathbf{t}_1, \mathbf{t}_3)$	(t_4, t_2)
R_5	$(\mathbf{t}_6, \mathbf{t}_1)$	(t_2, t_5)	(t_4, t_3)	R_{10}	(t_4, t_6)	(t_3, t_2)	$(\mathbf{t}_1, \mathbf{t}_5)$

The schedule in Table 1 specifies that team t_1 has the following schedule: it plays against teams: t_5 away, t_3 away, t_6 at home, t_2 at home, t_6 away, t_4 away, t_2 away, t_4 at home, t_3 at home and t_5 at home. The t_1 tour π^{t_1} is: $(h_1, h_5, h_3, h_1, h_6, h_4, h_2, h_1)$.

The travel cost or distance value of team t_1 is:

$$dis(h_1, h_5) + dis(h_5, h_3) + dis(h_3, h_1) + dis(h_1, h_6) + dis(h_6, h_4) + dis(h_4, h_2) + dis(h_2, h_1)$$

More formally, the distance of a team tour π^t is the summation of distances between consecutive venues $dis(\pi_i^t, \pi_{i+1}^t)$ found in the tour π^t of team t . Locations of π_0^t and $\pi_{2(n-1)+1}^t$ must be at home. Thus, a tour distance is:

$$\pi^t = \sum_{i=1}^{2(n-1)+1} dis(\pi_{i-1}^t, \pi_i^t) \tag{1}$$

The travel cost of the schedule (*travel_cost*):

$$travel_cost = \sum_{t=1}^n \pi^t \tag{2}$$

We note that a good optimization in TTP may lead to great savings in travel costs due to reduced travel distances, while poor schedules might lead to heavy budget losses to the sport league management. Furthermore, reducing the traveling distances creates more opportunities for players to train along a season.

3 Related Works

TTP is a core problem in sports scheduling. Several works in different contexts tackled TTP. Among them, we mention the integer programming, constraint programming, and their hybridization which is a useful tool to model and solve small TTP instances: In [6], de Carvalho and Lorena, presented an integer programming formulation to the mirrored version of TTP (mTTP) and the Max-MinTTP variant, in which the problem of minimizing the longest traveled distance is addressed. Another exact method based on branch-and-price-based solution for the TTP is developed in [10]. They optimally solved NL8 and CIRC6 instances.

Rasmussen and Trick, in [19] proposed an exact method based on constraint programming and branch-and-price for a special case of TTP so-called TCDMP (Constrained distance minimization problem). The TCDMP problem consists of finding an optimal home-away assignment when the opponents of each team in each round are given.

Approximation algorithms were also proposed for the TTP: in [24] Westphal and Noparlik proposed an algorithm that approximates the optimal solution by a factor of 5.875.

For the large instances, the methods based on meta-heuristics are the most successful approaches for TTP. In [15], a hybrid approach combining a simulated annealing (SA) and hill-climbing components is proposed for TTP. Variable neighborhood search [12] and a clustering search (CSA) method for mTTP [4] are also proposed to solve TTP and they have got results comparable with the best-known solution for NL and CON instances.

Further, in [20], a hybridization of GRASP (Greedy Randomized Adaptive Search Procedure) and ILS (Iterated Local Search) are proposed for solving mTTP problem. Their experiments were performed on CIRC and NL instances.

A new enhanced harmony search combined with a variable neighborhood search (V-HS) was developed for mTTP in [13]. This approach was evaluated on instances of up to size 16 for NL instances and 20 for CON instances. The approach matches the optimal solution for NL 6, CON 4 to CON 12 instances and the general deviation from optimality is equal to 4.45%. In [1], an enhanced simulated annealing for TTP (TTSA) is proposed for TTP. This method was successful for solving TTP and it improved the best known solution for NL instances.

Other approximation algorithms were conducted for the UTTP in [9]. Their idea is based on the circle method (the Kirkman schedule) and the shortest Hamiltonian cycle passing by all venues teams.

4 Contributions

We propose an effective and fast heuristic method for UTTP. The proposed method is structured into three main successive steps where the input is the number of the teams ($n < 10$) and the output is a set of DRRT schedules for sport leagues with low-cost travel. The different steps of the proposed method for UTTP are given as follows:

1. **Step 1. Generation of all the possible rounds:** In this step, we create all the possible rounds by enumerating all the perfect matching of the initial directed graph of the teams $G_T(H, E)$ (Sect. 4.1). We note that, a perfect matching of a graph is a matching in which every vertex of the graph is incident to exactly one edge of the matching.
2. **Step 2. Enumeration of the $2(n - 1) - cliques$ of the graph of the rounds** (Sect. 4.2): In this step first, we use the resulting rounds from the previous step to create the undirected graph of rounds $G_R(V, U)$. Then, we enumerate all the $2(n - 1) - cliques$ of the graph $G_R(V, U)$.
3. **Step 3. Creation of the DRRT schedules:** In this last step, we sort the vertices of each $2(n - 1) - clique$ (the cliques of the second step) to form DRRT schedules with low-cost travel (Sect. 4.3).

Figure 1 illustrates the different steps of our proposed approach for UTTP.

As shown in Fig. 1, the method starts with a set of n teams with a distance matrix dis . We create the complete directed graph of teams. Then, from this graph we enumerate all the 1-factors of the initial graph where 1-factor represents a round. For this purpose, we use a simple backtracking and recursion algorithm. In the next step, we generate the undirected graph of rounds and enumerate all the $2(n - 1) - cliques$ of the graph. After that, we sort the vertices of each $2(n - 1) - clique$ in order to form the DRRT schedules. The schedule with lowest cost travel is selected in the last step to be the solution for our problem. More details are given in the next subsections.

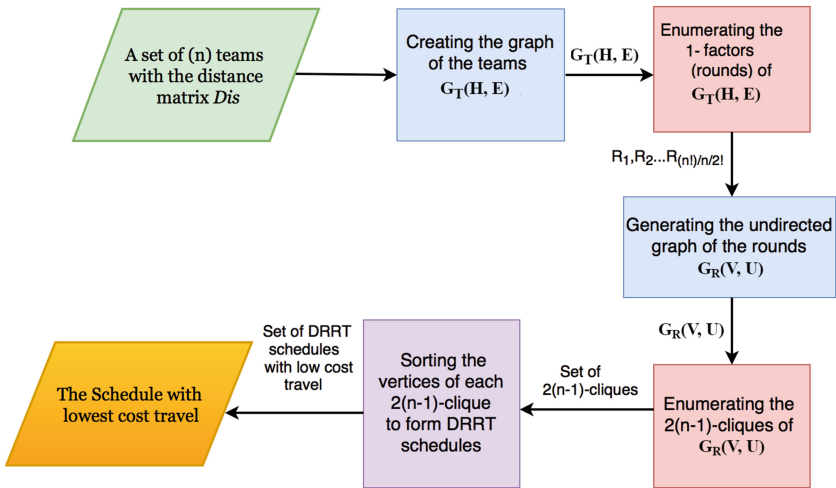


Fig. 1. The flowchart of the proposed approach

4.1 Step1. Generation of All the Possible Rounds

We start with a complete directed graph $G_T(H, E)$ of the teams. As already said, the vertices $H = \{h_i\}_{i=1}^n$ of the graph $G_T(H, E)$ are the home locations of the teams $T = (t_1, \dots, t_n)$. The arc $e_{i,j}^{\rightarrow}$ corresponds to the matching between team t_i and team t_j , the direction of the arc indicates that the game takes place in the home city of team t_i .

Figure 2 shows an example of a complete directed graph $G_T(H, E)$, where $H = \{t_1, t_2, t_3, t_4\}$.

We use here the concept of 1-factor. The 1-factor of a graph $G_T(H, E)$ is a sub-graph $G_T(H, E')$, with $E' \subset E$, all edges of E' are independent (they have no common end vertex) and all the vertices have their degrees equal to one. Since each team plays only one game per round, $n/2$ different games are played in every round.

Accordingly, each round corresponds to a 1-factor in $G_T(H, E)$. More precisely, the oriented 1-factor may model the $n/2$ games scheduled for a given round [7].

In the 1-factor, each game is modeled by an arc (t_i, t_j) oriented from team t_i to t_j which means that the game is played in the home location of team t_i . Figure 2 depicts examples of 1-factors that represent the schedules of given rounds.

For example, the first 1-factor (R_a) may schedule one round $R_a : (t_1, t_2)(t_4, t_3)$ (as shown in Fig. 2).

We remind that the aim of the first step is to derive all possible rounds by getting the oriented 1-factors of $G_T(H, E)$: $|H| = n$ (n teams).

Note: the number of 1-factors of a complete directed graph G_n is equal to $\frac{(n)!}{(n/2)!}$

Proof: In the complete directed graph with n vertices, to form a 1-factor (perfect matching), each vertex must be assigned to a single arc, so for the first vertex there are $(n - 1) \times 2$ choices ($\times 2$ to orient the edge) to choose its neighbor. For the second vertex, we pick an unused vertex from the sub-graph on the rest $n - 2$ vertices, so it will result in $2(n - 3)$ ways to choose its neighbor, for the third vertex we will have $2(n - 5)$ ways to choose its neighbor and so on with others vertices. Thus the overall number of 1-factors is equal to:

$$(n - 1) \times 2 \times (n - 3) \times 2 \times (n - 5) \times 2 \dots 3 \times 2 \times 1 \times 2 = 2^{n/2} \times (n - 1)!! = \frac{(n)!}{(n/2)!}$$

Figure 2 represents the 1-factors of $G_T(H, E)$: ($R_a : ((t_1, t_2), (t_4, t_3))$), ($R_b : ((t_1, t_2), (t_3, t_4))$), ($R_c : ((t_1, t_2), (t_3, t_4))$), ..., ($R_l : ((t_4, t_1), (t_2, t_3))$).

4.2 Step2. Enumeration of All the $2(n - 1) - cliques$ of the Graph of the Rounds

In this section, we explain the creation of the graph of the rounds. Then we show how we enumerate all the $2(n - 1) - cliques$ of the graph $G_R(V, U)$.

Creation of the Graph of the Rounds. The resulting rounds of the previous step $V := \{R_i\}_{i=1}^{\frac{(n)!}{(n/2)!}}$ are used to form an undirected graph $G_R(V, U)$ where V are the nodes and U are the edges:

$$U := \left\{ \{R_i, R_j\} : \text{Independent_Rounds}(R_i, R_j), i, j = 1, \dots, \frac{(n)!}{(n/2)!} \right\}.$$

We define independent_Rounds (R_i, R_j) where there is no common game between rounds R_i and R_j . This is illustrated in Fig. 2, there is no common game between $R_a : ((t_1, t_2), (t_4, t_3))$ and $R_c : ((t_3, t_1), (t_2, t_4))$, hence the corresponding vertices R_a and R_c in $G_R(V, U)$ must be linked by edges. On the other hand there is a common game (t_1, t_2) between rounds $R_a : ((t_1, t_2), (t_4, t_3))$ and $R_b : ((t_1, t_2), (t_3, t_4))$ hence both rounds(vertices) are not connected by an edge.

In Fig. 3, we give the graph $G_R(V, U)$ formed by using the rounds depicted in Fig. 2.

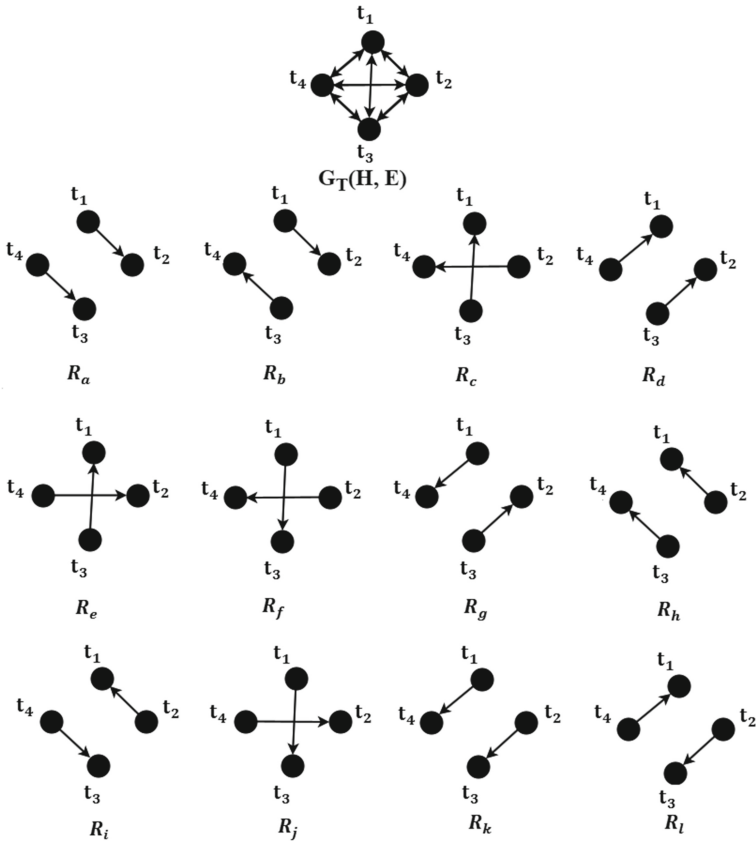


Fig. 2. The 1-factors of the complete graph $G_T(H, E)$ $n = 4$

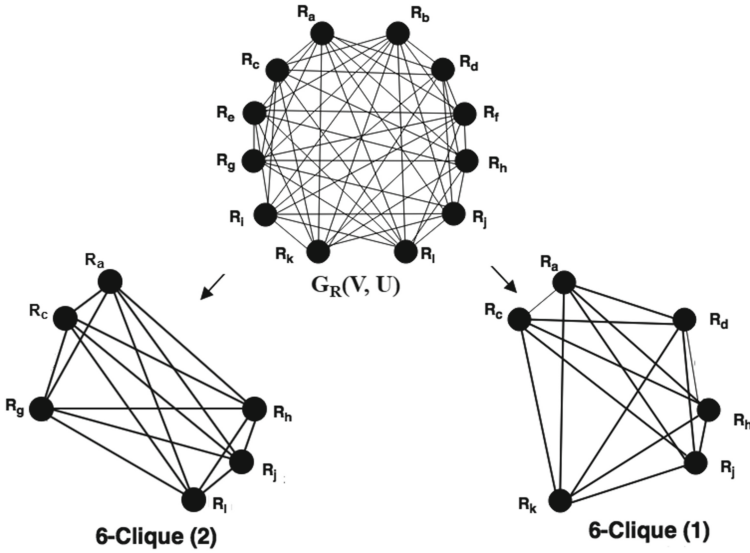


Fig. 3. The graph of the rounds $G_R(V, U)$ with two cliques getting from this graph

Enumeration of All $2(n - 1) - cliques$ of the Graph $G_R(V, U)$. A clique is a complete sub-graph that represents a subset of vertices, all adjacent to each other, and a k -clique is a clique with k vertices.

As already mentioned, a TTP solution is a DRRT schedule. DRRT has $n(n - 1)$ games which are organized in $2(n - 1)$ Independent_Rounds.

In the graph of the rounds $G_R(V, U)$ (see Fig. 3):

1. The adjacent rounds are Independent.
2. $2(n - 1) - clique$ in the graph $G_R(V, U)$ is a set of $2(n - 1)$ Independent_Rounds all adjacent to each other (see Fig. 3).

From (1) and (2), we can say that a $2(n - 1) - clique$ in $G_R(V, U)$ may schedule or form a DRRT schedule (see Fig. 4).

To enumerate all possible DRRT schedules, we enumerate the $2(n - 1) - cliques$ of the graph $G_R(V, U)$. Thus, the purpose of this step is to enumerate the $2(n - 1) - cliques$ of the graph of rounds $G_R(V, U)$. This is done by using Bron-Kerbosch clique detection algorithm [5, 23] with complexity of $O(3^{V/3})$ (V -vertex graph: for $n = 4$, V is equal to 12, for $n = 6$, V is equal to 120 and for $n = 8$, V is equal to 1680).

Figure 3 outlines an example of two $2(n - 1) - cliques$ of the graph of rounds, the 6 - clique (1) is formed by the independent rounds $(R_a, R_d, R_c, R_k, R_j, R_h)$, and the 6 - clique (2) formed by independent rounds $(R_a, R_c, R_g, R_l, R_j, R_h)$. Both 6 - cliques (1) and (2) may form two different DRRT schedules.

4.3 Step3. Creation of DRRT Schedules

To reorder efficiently the rounds of each clique and so create the corresponding schedules in the best possible way (with low- cost travel), we model the problem of sorting the rounds of each $2(n - 1) - clique$ as a traveling salesman problem (TSP) [14].

We sort the rounds (the vertices) of each $2(n - 1) - clique$ according to the shortest route passing all its vertices (the rounds of the schedule).

The process can be done as follows: for each $2(n - 1) - clique$:

1. We associate a weight to each edge of the clique $w(u_{i,j}) = dR(R_i; R_j)$ where $dR(R_i; R_j)$ indicates the distance between the rounds. It represents the distance traveled by the teams from round R_i to round R_j in the schedule:

$$dR(R_i; R_j) = \sum_{t=1}^n dis(\pi_{R_i}^t, \pi_{R_j}^t). \tag{3}$$

Figure 4 gives an example of weighting each edge in the clique 6-clique(1): $dR(R_a; R_d) = dis(h_1, h_4) + dis(h_1, h_3) + dis(h_4, h_3)$.

2. Find the shortest route (in the $2(n - 1) - clique$) that visits each vertex exactly once (to ensure each round will appear once in the schedule). Then returns to the origin vertex; accordingly we are looking for solving TSP [14] in the $2(n - 1) - clique$ in $G_R(V, U)$. We use the nearest neighbor algorithm (O(log n)-approximation algorithm) [2,14,21]). This is done in order to sort

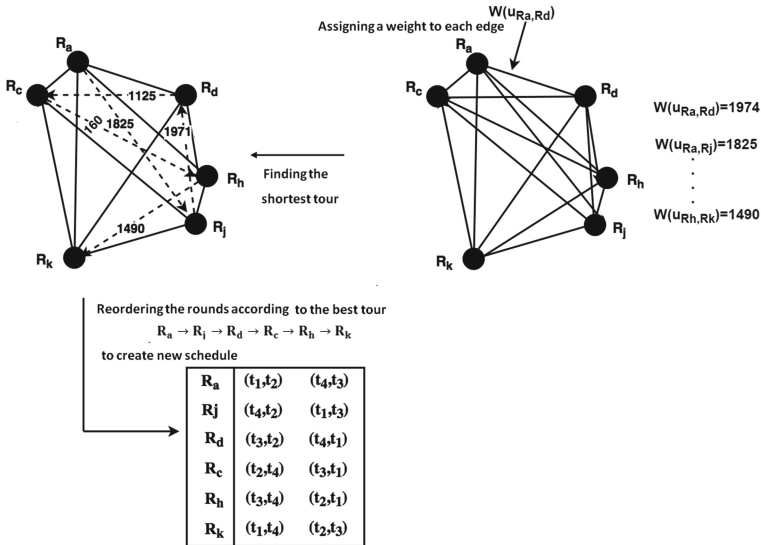


Fig. 4. Reordering the vertices of 6 - clique(1) to form schedule with low-cost travel

the rounds of the each $2(n - 1) - clique$ in the best way which forms the corresponding schedule with low-cost travel.

As shown in Fig. 4, the shortest route in $6 - clique(1)$ is $R_a \rightarrow R_j \rightarrow R_d \rightarrow R_c \rightarrow R_h \rightarrow R_k$.

3. We sort the rounds of the clique according to the TSP solution to create a new schedule S with low-cost travel.

Figure 4 illustrates the process of sorting the vertices of $6 - clique(1)$ (depicted in Fig. 3), with the aim to build a DRRT schedule with low-cost travel.

5 Experiments

The proposed method is evaluated on some well-known datasets for TTP. The source code is written in Java and run on an Intel(R) Core(TM) i5 4270UCPU @ (3.20 GHz) with 8 GB of RAM.

We consider the well-known benchmarks instances [16] which are: the so-called NLx, CONx, CIRCx, SUPERx (x stands for the dimension of the instance).

- **NLx** instances are based on real data of the US National Baseball League, where x is an even number of teams. The NLx family of instances is probably the most researched TTP benchmark- family and virtually all researches studying the TTP publish their computational results with NLx instances [11, 19, 24].
- **CONx** The constant distance instances are characterized by a distance of one (1) between all teams.
- **SUPERx** is based on Rugby League, a league with 14 teams from South Africa, New Zealand and Australia.
- **CIRCx** instances: all teams are placed on a circle, with unit distances (distance of 1 between all adjacent nodes). The distance between two teams i and j with $i > j$ is then equal to the length of the shortest path between i and j which is the minimum of $i - j$ and $j - i + n$.

5.1 The Numerical Results

Table 2 gives the numerical results obtained by our approach on the different NL, CON, CIRC, and SUPER instances. The first column gives the name of the instance. The column (best-known) is the best known solutions for the considered instance (UTTP) [9, 16]. The column *ourapproach* provides the travel cost achieved by the proposed algorithm. The column $Gap_{(ourapproach/best-known)}$ provides the gap between the best solution of our method and the best-known solutions.

$$Gap_{(ourapproach/best-known)}\% = \left(\frac{travel_cost_{ourapproach} - best - known}{travel_cost_{ourapproach}} \right) \cdot 100 \quad (4)$$

Table 2. Comparative study with the best-known solutions

Instance	Our approach	Best-known	$Gap_{best-known/ourapproach}$
NL4	8276 ^{Opt}	8276	0%
NL6	19900*	20547	-3.25%
NL8	30700 *	33190	-8.11%
CON4	17^{Opt}	17	0%
CON6	38^{Opt}	38	0%
CON8	67^{Opt}	67	0%
CIRC4	20^{Opt}	20	0%
CIRC6	54^{Opt}	54	0%
CIRC8	102^{Opt}	102	0%
Super4	63405	-	-
Super6	99825	-	-
Super8	134980	-	-

* New best solutions on UTTP

Opt Optimal solution

The numerical results show that our proposed approach generates schedules of high quality. In order to quantify this improvement, we compute the arithmetic mean of the percentage (Gap), which we called the performance ratio (PR):

$$PR = \sum_{i=1}^{NBins} Gap_{ourapproach/best-knowni} / NBins \quad (5)$$

The $Gap_{ourapproach/best-knowni}$ (formula 4) is the gap between the best solution of our method and the best known solution (best-known) of instance i , and $NBins$ is the number of the tested instances for each benchmark.

As reported in Table 2, our approach succeeds in improving the current best solutions [9] for the tested NL instances. The performance ratio is equal to $PR = 3.78\%$. Thus our approach improves the best known solutions for the NL considered instances ($n < 10$) by 3.78% in average.

Further, the results show that the new proposed approach is able to achieve the optimal solution [16] for CON4, CON6 and CON8 instances. The proposed approach matches the best known solutions on CIRC4, CIRC6 and CIRC8 instances. For the SUPER instances we can say that the proposed approach is the first research effort that handles them, thus for the moment our results are the best solutions. Additionally, our heuristic can be used for solving the 4, 6 and 8 teams TTP version by selecting the best schedule that satisfies both constraints. It can reach the optimal solution for NL4, NL 6 and NL8 instances on TTP.

We note that the obtained schedules are validated by using the validator of Luca Di Gaspero and Andrea Schaerf from the Challenge Traveling Tournament Instances web site [16].

6 Conclusion

This paper proposed a novel and effective heuristic for the well-known UTTP which is the problem of scheduling a double round robin tournament, by minimizing the total distances traveled by the teams. The proposed method is a graph-based heuristic where the input is the number of the teams ($n < 10$) with asymmetric n by n integer distance matrix that represents the distance between team sites and the output is a set of DRRT schedules for sports leagues with low-cost travel. The proposed approach is evaluated on several instances and compared with the state-of-the-art. The numerical results show the performance of the proposed approach. The latter significantly improves the current best solutions for the considered National League (NL) instances. Further, the proposed approach is able to produce new good solutions to unsolved Rugby League composed of teams from New Zealand, Australia, and South Africa (SUPER) instances. It is important to note that our contribution is not only to achieve a good algorithm to solve UTTP but also to propose a new strategy to generate appropriate schedules for sport leagues with low-travel cost. As future work, we plan to combine our approach with integer programming based methods to solve large TTP instances.

References

1. Anagnostopoulos, A., Michel, L., Van Hentenryck, P., Vergados, Y.: A simulated annealing approach to the traveling tournament problem. *J. Sched.* **9**(2), 177–193 (2006)
2. Bellmore, M., Nemhauser, G.L.: The traveling salesman problem: a survey. *Oper. Res.* **16**(3), 538–558 (1968)
3. Bhattacharyya, R.: Complexity of the unconstrained traveling tournament problem. *Oper. Res. Lett.* **44**(5), 649–654 (2016)
4. Biajoli, F.L., Lorena, L.A.N.: Clustering search approach for the traveling tournament problem. In: Gelbukh, A., Kuri Morales, Á.F. (eds.) MICAI 2007. LNCS (LNAI), vol. 4827, pp. 83–93. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76631-5_9
5. Bron, C., Kerbosch, J.: Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM* **16**(9), 575–577 (1973)
6. de Carvalho, M.A.M., Lorena, L.A.N.: New models for the mirrored traveling tournament problem. *Comput. Ind. Eng.* **63**(4), 1089–1095 (2012)
7. De Werra, D.: Scheduling in sports. *Stud. Graphs Discrete Program.* **11**, 381–395 (1981)
8. Easton, K., Nemhauser, G., Trick, M.: The traveling tournament problem description and benchmarks. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 580–584. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45578-7_43
9. Imahori, S., Matsui, T., Miyashiro, R.: A 2.75-approximation algorithm for the unconstrained traveling tournament problem. *Ann. Oper. Res.* **218**(1), 237–247 (2014)
10. Irnich, S.: A new branch-and-price algorithm for the traveling tournament problem. *Eur. J. Oper. Res.* **204**(2), 218–228 (2010)

11. Kendall, G., Knust, S., Ribeiro, C.C., Urrutia, S.: Scheduling in sports: an annotated bibliography. *Comput. Oper. Res.* **37**(1), 1–19 (2010)
12. Khelifa, M., Boughaci, D.: A variable neighborhood search method for solving the traveling tournaments problem. *Electr. Notes Discrete Math.* **47**, 157–164 (2015)
13. Khelifa, M., Boughaci, D.: Hybrid harmony search combined with variable neighborhood search for the traveling tournament problem. In: Nguyen, N.-T., Manolopoulos, Y., Iliadis, L., Trawiński, B. (eds.) ICCCI 2016, Part I. LNCS (LNAI), vol. 9875, pp. 520–530. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45243-2_48
14. Laporte, G.: The traveling salesman problem: an overview of exact and approximate algorithms. *Eur. J. Oper. Res.* **59**(2), 231–247 (1992)
15. Lim, A., Rodrigues, B., Zhang, X.: A simulated annealing and hill-climbing algorithm for the traveling tournament problem. *Eur. J. Oper. Res.* **174**(3), 1459–1478 (2006)
16. Michael, T.: Challenge traveling tournament instances. <http://mat.tepper.cmu.edu/TOURN/>. Accessed Jan 2018
17. Rasmussen, R.V., Trick, M.A.: A Benders approach for the constrained minimum break problem. *Eur. J. Oper. Res.* **177**(1), 198–213 (2007)
18. Rasmussen, R.V., Trick, M.A.: Round robin scheduling—a survey. *Eur. J. Oper. Res.* **188**(3), 617–636 (2008)
19. Rasmussen, R.V., Trick, M.A.: The timetable constrained distance minimization problem. *Ann. Oper. Res.* **171**(1), 45 (2009)
20. Ribeiro, C.C., Urrutia, S.: Heuristics for the mirrored traveling tournament problem. *Eur. J. Oper. Res.* **179**(3), 775–787 (2007)
21. Rosenkrantz, D.J., Stearns, R.E., Lewis, P.M.: An analysis of several heuristics for the traveling salesman problem. In: Ravi, S.S., Shukla, S.K. (eds.) *Fundamental Problems in Computing*, pp. 45–69. Springer, Dordrecht (2009). https://doi.org/10.1007/978-1-4020-9688-4_3
22. Thielen, C., Westphal, S.: Complexity of the traveling tournament problem. *Theor. Comput. Sci.* **412**(4–5), 345–351 (2011)
23. Tomita, E., Tanaka, A., Takahashi, H.: The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.* **363**(1), 28–42 (2006)
24. Westphal, S., Noparlik, K.: A 5.875-approximation for the traveling tournament problem. *Ann. Oper. Res.* **218**(1), 347–360 (2014)



Augmenting Stream Constraint Programming with Eventuality Conditions

Jasper C. H. Lee¹, Jimmy H. M. Lee^{2(✉)}, and Allen Z. Zhong²

¹ Department of Computer Science, Brown University,
Providence, RI 02912, USA
jasperchlee@brown.edu

² Department of Computer Science and Engineering,
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
{jlee, azhong}@cuhk.edu.hk

Abstract. Stream constraint programming is a recent addition to the family of constraint programming frameworks, where variable domains are sets of infinite streams over finite alphabets. Previous works showed promising results for its applicability to real-world planning and control problems. In this paper, motivated by the modelling of planning applications, we improve the expressiveness of the framework by introducing (1) the “until” constraint, a new construct that is adapted from Linear Temporal Logic and (2) the @ operator on streams, a syntactic sugar for which we provide a more efficient solving algorithm over simple desugaring. For both constructs, we propose corresponding novel solving algorithms and prove their correctness. We present competitive experimental results on the Missionaries and Cannibals logic puzzle and a standard path planning application on the grid, by comparing with Apt and Brand’s method for verifying eventuality conditions using a CP approach.

1 Introduction

Stream constraint programming [11, 12] is a recent addition to the family of constraint programming frameworks. Instead of reasoning about finite strings [7], the domain of the constraint variables in a *Stream Constraint Satisfaction Problem* (St-CSP) consists of *infinite streams* over finite alphabets. A St-CSP solver computes not only one but *all* stream solutions to a given St-CSP, succinctly represented as a deterministic Büchi automaton. Because of the infinite stream domains, and the fact we can find all solutions, the framework is particularly suitable for modelling problems involving time series, for example in control and planning, using one variable for each stream as opposed to using one variable per stream per time point in traditional finite domain constraint programming [1]. Lallouet et al. [11] first demonstrated such capabilities by implementing the game controller of Digi Invaders¹, a popular game on vintage Casio calculator

¹ See <https://www.youtube.com/watch?v=1YafgAcmov4> for a video of the game as implemented by Casio.

models, using the St-CSP framework. Lee and Lee [12] further applied the framework to synthesise PID controllers for simple robotic systems².

In addition to using St-CSPs for control, Lee and Lee also proposed a framework for modelling planning problems as St-CSPs, adapting that of Ghallab et al. [6] for finite domain constraint programming. Even though the St-CSP framework can express the entirety of what finite domain CSPs could, there are still natural constraints on plans that we expect to be able to express but are unable to. For example, we cannot express the constraint that the generated plan must *eventually* satisfy a certain condition, without imposing a hard upper bound on the number of steps before the plan must satisfy the condition.

This paper focuses on enhancing the expressiveness of the St-CSP framework, using planning problems as a motivation. We introduce the “until” constraint (Sect. 3), adapted from Linear Temporal Logic (LTL) [14], which includes as a special case the “eventually” constraint. In addition, in the case where we do wish to concretely bound the number of steps before a condition is satisfied, we introduce the \mathbb{C} operator (Sect. 4) to simplify the modelling from the approach of Lee and Lee. There are two advantages to using the new operator in constraints: (1) we can better leverage known structure to accelerate solving, and (2) the notation is significantly less cumbersome, as measured in the length of the constraint expressions. We give experimental evidence (Sect. 5) of the competitiveness of our new solving algorithms.

For space reasons, we give only proof sketches of some of the results. The full paper is available at <https://arxiv.org/abs/1806.04325>, which includes all proofs, constraint models of our experiments and also more detailed exposition.

2 Background

We review the basics of stream constraint programming.

Existing Stream Expressions and Constraints. A *stream* a over a (finite) alphabet Σ is a function $\mathbb{N}_0 \rightarrow \Sigma$. For example, the function $a(n) = n \bmod 2$ is a stream over *any* alphabet containing $\{0, 1\}$. The set of all streams with alphabet Σ is denoted by Σ^ω . The notation $a(i, \infty)$ is used for the stream suffix a' where $a'(j) = a(j + i)$. For a language L , we similarly define $L(i, \infty) = \{a(i, \infty) \mid a \in L\}$. In this paper, we are only concerned with St-CSPs whose variables take alphabets that are integer intervals, i.e. $[m..n]^\omega$ for some $m \leq n \in \mathbb{Z}$. However, the framework generalises naturally to any other finite alphabets.

To specify expressions, there are primitives such as variable streams, which are the variables in the St-CSP, and constant streams. For example, the stream 2 denotes the stream s where $s(i) = 2$ for all $i \geq 0$.

Pointwise operators, such as integer arithmetic operators $\{+, -, *, /, \%\}$, combine two streams at each index using the corresponding operator. Integer arithmetic relational operators are $\{\text{lt}, \text{le}, \text{eq}, \text{ge}, \text{gt}, \text{ne}\}$. They compare two

² See <http://www.youtube.com/watch?v=dT56qAZt8hI> and <http://www.youtube.com/watch?v=5GvbG3pN0vY> for video demonstrations.

streams pointwisely and return a *pseudo-Boolean stream*, that is a stream in $[0..1]^\omega$. Pointwise Boolean operators `{and, or}` act on any two pseudo-Boolean streams a and b . The final pointwise operator supported is `if-then-else`. Suppose c is pseudo-Boolean, and a, b are streams in general, then `(if c then a else b)` (i) is $a(i)$ if $c(i) = 1$ and $b(i)$ otherwise. There are also three *temporal operators*, in the style of the Lucid programming language [16]: `first`, `next` and `fbv`. Suppose a and b are streams. We have `first a` being the constant stream of $a(0)$, and `next a` being the “tail” of a , that is `next a` $= a(1, \infty)$. In addition, a `fbv` $b = c$ is the concatenation of the head of a with b (*a followed by b*), that is $c(0) = a(0)$ and $c(i) = b(i - 1)$ for $i \geq 1$. Note that stream expressions can involve stream variables. For example, `(first y) + (next x)` is an expression.

Given stream expressions, we can now use the following relations to express stream constraints. For integer arithmetic comparisons $R \in \{<, <=, ==, >=, >, !=\}$, the constraint $a R b$ is *satisfied* if and only if the arithmetic comparison R is true at every point in the streams. Therefore, a constraint is *violated* if and only if there exists a time point at which the arithmetic comparison is false. For example, `next x != y + 1` is a constraint enforcing that the stream expression $y + 1$ is not equal to the stream `next x` at all time points. Similarly, we define the constraint $a \rightarrow b$ to hold if and only if for all $i \geq 0$, $a(i) \neq 0$ implies $b(i) \neq 0$. Here we use the C language convention for interpreting integers as Booleans.

Care should be taken to distinguish between constraints and relational expressions. Relational operators take two streams and output a pseudo-Boolean stream. Constraints, however, are relations on streams. Two simple examples illustrate the difference: $x \text{le } 4$ is a pseudo-Boolean stream, whereas $x \leq 4$ is a constraint that enforces x to be less than or equal to 4 at every time point.

Stream Constraint Satisfaction Problems

Definition 1 [11,12]. A stream constraint satisfaction problem (*St-CSP*) is a triple $P = (X, D, C)$, where X is the set of variables and $D(x) = (\Sigma(x))^\omega$ is the domain of $x \in X$, the set of all streams with alphabet $\Sigma(x)$. A constraint $c \in C$ is defined on an ordered subset $\text{Scope}(c)$ of variables, and every constraint must be formed as specified previously (though it is the aim of this paper to extend the class of specifiable constraints).

Figure 1 gives an example St-CSP. An *assignment* $A : X \rightarrow \bigcup_{x \in X} D(x)$ is a function mapping a variable $x_i \in X$ to an element in its domain $D(x_i)$. A constraint c is satisfied by an assignment A if and only if it is satisfied by the streams $\{A(x)\}_{x \in \text{Scope}(c)}$, and a St-CSP P is *satisfied* by A if and only if all constraints $c \in C$ are satisfied by A . We call the assignment A a *solution* of the St-CSP P . We denote the *solution set* of P , namely the set of all solutions A to P , by $\text{sol}(P)$. The St-CSP P is *satisfiable* if $\text{sol}(P)$ is non-empty, and *unsatisfiable* otherwise. We also say that two St-CSPs P and P' are *equivalent* (denoted $P \equiv P'$) when $\text{sol}(P) = \text{sol}(P')$.

Given a set of constraints C and an integer i , the *shifted view* of C is defined as $C(i, \infty) = \{c_k(i, \infty) \mid c_k \in C\}$ by interpreting constraints as languages. Similarly,

given an St-CSP $P = (X, D, C)$ and a point i , the *shifted view* of P is defined as $\hat{P}(i) = (X, D, C(i, \infty))$.

Solving St-CSPs. Lallouet et al. [11] showed that the solution set $sol(P)$ of a St-CSP P is a *deterministic ω -regular language*, accepted by some *deterministic Büchi automaton* \mathcal{A} , which is a deterministic finite automaton for languages of streams [3]. A stream s is accepted by \mathcal{A} if the execution of \mathcal{A} on input s visits accepting states of \mathcal{A} infinitely many times. When given a St-CSP P , the goal of a St-CSP solver, then, is to produce a *deterministic Büchi automaton* \mathcal{A} , called a *solution automaton* of P , that accepts the language $sol(P)$. We note that the work of Golden and Pang [7] for finite string constraint reasoning also finds all solutions as a single regular expression.

A St-CSP can be solved by a two-step approach [11, 12]. First, a given St-CSP P is *normalised* into some normal form P' where auxiliary variables may be introduced, but P' is equivalent to P modulo the auxiliary variables. Afterwards, the *search tree* (as defined below) is explored and “morphed” into a deterministic Büchi automaton via a *dominance detection procedure*, which is then output as the solution automaton. In the rest of the paper, when we augment the language for specifying stream expressions and constraints, we shall also follow the above two-step approach to solve these new classes of St-CSPs. As such, we only have to (a) specify our new normal forms, (b) give a corresponding normalisation procedure, and (c) detail the new dominance detection procedures.

We now define the notion of search trees for St-CSPs, adapted from that for traditional finite-domain CSPs [4]. We also describe how they are explored and how dominance detection allows us to compute solution automata from search trees. A *search tree* for a St-CSP P is a tree with potentially infinite height. Its nodes are St-CSPs with the root node being P itself. The *level* of a node N is defined as 0 for the root node and recursively for descendants. A child node $Q' = (X, D, C \cup \{c'\})$ at level $k + 1$ is constructed from a parent node $P' = (X, D, C)$ at level k and an *instantaneous assignment* $\tau(x) \in \Sigma(x)$, where τ takes a stream variable x and returns a value in $\Sigma(x)$. In other words, τ assigns a value to each variable at time point k . The constraint c' specifies that for all $x \in X$, $x(k) = \tau(x)$ and for all $i \neq k$, $x(i)$ is unconstrained. We write $P' \xrightarrow{\tau} Q'$ for such a parent to child construction, and label the edge on the tree between the two nodes with τ . During search in practice, we shall *not* consider every possible instantaneous assignment, but instead consider only the ones remaining after applying *prefix- k consistency* [11].

We can identify a search node Q at level k with the shifted view $\hat{Q}(k)$. Taking this view, if $\hat{P}(k) = (X, D, C)$ is the parent node of $\hat{Q}(k + 1)$, then $\hat{Q}(k + 1) = (X, D, C \cup \{c'\})(1) = (X, D, (C \cup \{c'\})(1, \infty))$ where c' is the same constraint as defined above.

Recall that a constraint violation requires only a single time point at which the constraint is false. Therefore, we can generalise the definition of constraint violation such that a finite prefix of an assignment can violate a constraint. A sequence of instantaneous assignments from the root to a node is isomorphic to a finite prefix of an assignment, and so the definition again generalises. Suppose

$F = (X, D, C)$ is a node at level k such that $\{\tau_i\}_{i \in [1..k]}$ is the sequence of instantaneous assignments that constructs F from the root node, i.e. $P \xrightarrow{\tau_1} \dots \xrightarrow{\tau_k} F$. We say node F is a *failure* if and only if $\{\tau_i\}_{i \in [1..k]}$ violates a constraint $c \in C$.

Given a normalised St-CSP P , its search tree is then explored using depth first search. Backtracking happens when the current search node is a failure. A search node M at level k is said to *dominate* another search node N at level k' , written $N \prec M$, if and only if their shifted views are equivalent ($\hat{M}(k) \equiv \hat{N}(k')$) and M is visited before N during the search [11, 12]. When the algorithm visits a search node N that is dominated by a previously visited node M , the edge pointing to N is redirected to M instead. If the algorithm terminates, then the resulting (finite) structure is a deterministic Büchi automaton (subject to accepting states being specified). If dominance detection were perfect, then the search algorithm terminates, because every branch either ends in a failure or contains two nodes with the same shifted views [11]. The crucial missing detail from this high-level algorithm, then, is *exactly* how dominance is *detected* in practice. Search node dominance is an inherently semantic notion, implying that it is often inefficient to detect precisely. Thus, previous works identify efficient *syntactic approximations* to detecting dominance such that the overall search algorithm terminates [11, 12]. We shall also give a new dominance detection procedure in light of the new ways of forming stream expressions and constraints. As for specifying the set of accepting states, previous work take *all* states as accepting states, whereas we shall give a more nuanced criterion.

3 The “Until” Constraint

In this section, we introduce the “until” constraint to the St-CSP framework. Recall that all the stream constraints introduced in Sect. 2 are pointwise predicates. That is, the constraint is satisfied if its corresponding predicate holds for every single time point of its input streams. The “until” constraint, as we shall later see, is *not* a pointwise constraint.

Let us consider the following path planning problem on the standard $n \times n$ grid world domain [8, 15]. Between any two neighbouring vertices on the grid, there could be 0, 1 or 2 *directed* edges. We ask for all paths on the directed graph from a given start point that eventually visit a given end point.

Our method finds more than a shortest path. Modelling this problem as a St-CSP allows us to find a succinct description of *all* the paths, and moreover allows for additional side constraints. Well-studied side constraints in the literature include precedence constraints [10] and time window constraints [13].

We can formulate as a St-CSP the condition that the path starts at (i_s, j_s) , has to respect the graph, and furthermore in the St-CSP model check whether the goal of visiting the end point (i_g, j_g) is attained. This St-CSP is shown in Fig. 1. We use variables x, y to represent the x and y coordinates of the current position. In addition, a variable *goal* denotes if we have visited the end point. The second to last constraint is such that if *goal* is true in one time point, it

```

var  $x, y$  with alphabet  $[1..n]$ 
var  $goal$  with alphabet  $[0..1]$ 

first  $x == i_s$ 
first  $y == j_s$ 

For each vertex  $(i, j)$ ,
 $((next\ x\ eq\ i)\ and\ (next\ y\ eq\ j)) \rightarrow ((x\ eq\ i\ and\ y\ eq\ j)\ or$ 
 $(x\ eq\ i_1\ and\ y\ eq\ j_1)\ or\ \dots\ or\ (x\ eq\ i_{d(i,j)}\ and\ y\ eq\ j_{d(i,j)})$ 
where  $(i_1, j_1), \dots, (i_{d(i,j)}, j_{d(i,j)})$  have edges into  $(i, j)$ 
and  $d(i, j)$  is the in-degree of  $(i, j)$ 

 $goal == (x\ eq\ i_g\ and\ y\ eq\ j_g)\ or\ (0\ fby\ goal)$ 
 $goal\ eq\ 1 \rightarrow ((x\ eq\ next\ x)\ and\ (y\ eq\ next\ y))$ 

```

Fig. 1. St-CSP model for the path planning problem

stays true in the next one as well. The last constraint says that if the path has reached the end point, then it stays there indefinitely.

In this current model, we have not enforced that the goal is indeed *eventually* attained at some point. An undesirable solution to the St-CSP would be, for example, to stay in one location forever. However, variants of the “eventually” constraint is not expressible in the St-CSP framework prior to this work, since all constraints are inherently *pointwise*. Temporal operators are not expressive enough for our purpose, since these operators shift streams by a constant number of time points only. The “eventually” constraint, on the other hand, can be satisfied at an unbounded number of time points away into the future.

We thus introduce the “until” constraint, adapted from Linear Temporal Logic (LTL) [14] and essentially equivalent to “eventually” [5].

Definition 2 (The “Until” Constraint). *Given two streams a, b , the constraint a until b is satisfied if and only if there exists a time point $i \geq 0$ such that (1) for all $j < i$, $a(j) \neq 0$ and (2) $b(i) \neq 0$. We say that the constraint is finally satisfied at time point i if $b(i) \neq 0$. Note that we are again adapting the C language convention for interpreting integers as Booleans.*

The “eventually” constraint is expressible in terms of the “until” constraint. Suppose we want to express the constraint that a predicate G on stream elements eventually holds, for example if G is “ $goal\ eq\ 1$ ”. Then, we can express the constraint as “ 1 until G ”, or in our particular example, “ 1 until ($goal\ eq\ 1$)”. Conversely, “ a until b ” is equivalent to “ $c == b\ fby\ (next\ b\ or\ c); (not\ c) \rightarrow (a\ ne\ 0);$ eventually b ”.

3.1 Normalising “Until” Constraints

In light of the “until” constraint, we give a new constraint normal form. A St-CSP is in *normal form* if it contains only constraints of the following forms:

- Primitive next constraints: $x_i == \text{next } x_j$
- Primitive until constraints: $x_i \text{ until } x_j$
- Primitive pointwise constraints with no `next`, `fbv` or `until` (but can contain `first` operators).

Any St-CSP can be transformed into this normal form by applying the rewriting system below. We adopt notations from programming language semantics theory [17], writing $c[_]$ for *constraint contexts*, i.e. constraints with placeholders for syntactic substitution. For example, if $c[_] = [_ + 3 >= 4]$, then $c[\text{first } \alpha] = [(\text{first } \alpha) + 3 >= 4]$. We also write a constraint rewriting transition as $(C_0, C_1) \rightsquigarrow (C'_0, C'_1)$, where C_0, C_1, C'_0 and C'_1 are sets of constraints. C_0 is the set of constraints that potentially could be further normalised, and C_1 is the set that is already in normal form. Hence, the initial constraint pair for the St-CSP (X, D, C) is $(C, \{\})$. Rules are applied *in arbitrary order* until none are applicable.

- $(C_0 \cup \{c[\text{next } \text{expr}]\}, C_1) \rightsquigarrow (C_0 \cup \{c[x_1], x_2 == \text{expr}\}, C_1 \cup \{x_1 == \text{next } x_2\})$, where x_1 and x_2 are fresh auxiliary stream variables.
- $(C_0 \cup \{c[\text{expr}_1 \text{ fby } \text{expr}_2]\}, C_1) \rightsquigarrow (C_0 \cup \{c[x_1], x_2 == \text{expr}_1, x_3 == \text{expr}_2\}, C_1 \cup \{\text{first } x_1 == \text{first } x_2, x_3 == \text{next } x_1\})$, where x_1, x_2 and x_3 are fresh auxiliary stream variables.
- $(C_0 \cup \{\text{expr}_1 \text{ until } \text{expr}_2\}, C_1) \rightsquigarrow (C_0 \cup \{x_1 == \text{expr}_1, x_2 == \text{expr}_2\}, C_1 \cup \{x_1 \text{ until } x_2\})$, where x_1 and x_2 are fresh auxiliary stream variables.

We can check easily the following properties of the new rewriting system.

Proposition 1. *The new rewriting system always terminates, regardless of the order in which the rules are applied.*

Proposition 2. *The rewriting system has the Church-Rosser property (up to auxiliary variable renaming).*

Proposition 3. *The rewriting system is sound, in the sense that it preserves the projection of the solution set of the resulting St-CSP into the original variables.*

3.2 Search Algorithm and Dominance Detection

In the following, we assume that all given St-CSPs are in normal form.

Recalling the high-level solving algorithm in Sect. 2, we give in this section a concrete instantiation of the syntactic dominance detection procedure. Our syntactic procedure should possess two key properties. First, the procedure should be *sound*: if two search nodes are claimed to have equivalent shifted views by the procedure, then they do indeed have equivalent shifted views. Second, the approximation should be sufficiently close to the semantic notion, such that the overall search algorithm terminates and produces a *finite* structure. Otherwise, in the extreme scenario where the dominance detection procedure never reports any dominance, the search algorithm will simply search the entire (usually infinite) search tree, resulting in non-termination.

Algorithm 1. Dominance Detection with Until Constraints

```

1: function CONSTRUCT(Search Node  $\hat{P}(k) = (C, h)$ , Instantaneous Assignment  $\tau$ )
2:   Historic values  $h' \leftarrow \emptyset$ 
3:   for all primitive next constraints  $x_i == \text{next } x_j$  do
4:      $h'(x_j) \leftarrow \tau(x_i)$ 
5:   Constraint Set  $C' \leftarrow \emptyset$ 
6:   for all primitive until constraints  $x_i$  until  $x_j$  do
7:     if  $\tau(x_j) = 0$  then
8:        $C' \leftarrow C' \cup \{x_i \text{ until } x_j\}$ 
9:   for all primitive pointwise, next constraints  $c$  do
10:    Constraint  $c' \leftarrow c$  evaluated with  $\tau$ 
11:    if  $c'$  is not a zeroth order tautology then
12:       $C' \leftarrow C' \cup \{c'\}$ 
13:   return  $\hat{Q}(k+1) = (C', h')$ 
14: function AREEQUAL(Search Nodes  $\hat{P}(k) = (C_P, h_P)$ ,  $\hat{Q}(k') = (C_Q, h_Q)$ )
15:   return  $(C_P = C_Q) \wedge (h_P = h_Q)$ 

```

Our dominance detection procedure, as with previous works [11, 12], involves keeping track of a *syntactic* representation of the shifted view of each search node, and detects dominance by checking *syntactic equivalence* between the two representations. Hereafter, we refer to search nodes and their syntactic representations interchangeably for narratory simplicity. Each search node, then, is represented by two components: (1) a set C of St-CSP constraints and (2) a table h , called *historic values*, storing for each variable x_j in a primitive next constraint “ $x_i == \text{next } x_j$ ” the value assigned to x_i *at the previous time point*. The historic values are used to enforce primitive next constraints. If a value v is assigned to x_i at the previous time point, then **first** $x_j == v$ holds in the shifted view of the current search node. We thus store v in the table entry for x_j .

Algorithm 1 gives pseudocode for two functions, CONSTRUCT and AREEQUAL, both adapted from the algorithm of Lee and Lee [12] with *minimal* changes (lines 6–8) to accommodate “until” constraints. The function CONSTRUCT takes a parent search node $\hat{P}(k)$ and an instantaneous assignment τ , and outputs the corresponding child search node $\hat{Q}(k+1)$ (the new constraint set C' and new historic values h'). The function AREEQUAL, on the input of two search nodes, just checks whether their components are syntactically equal.

We describe the function CONSTRUCT in more detail. The new set of historic values h' is conceptually simple to compute. For each primitive next constraint “ $x_i == \text{next } x_j$ ”, we store $h'(x_j) = \tau(x_i)$ where τ is the instantaneous assignment given for the construction of the child search node. The new constraint set C' is computed from C by processing each constraint individually: (1) For primitive next constraints, we keep them as is and put them into C' . (2) For primitive pointwise constraints, we follow Lee and Lee [12] in *evaluating* them using the instantaneous assignment τ . That is, we substitute every variable stream x appearing in an expression whose outermost operator is the **first** operator,

using the value $\tau(x)$. This process produces expressions that consist entirely of constant streams, pointwise operators and **first** operators, and thus can be evaluated into a single constant stream. If, as a result, a primitive pointwise constraint becomes a numerical tautology (e.g. $1 == 1$), we discard such a constraint. (3) For primitive until constraints “ x_i **until** x_j ” (lines 6–8), we simply check whether $\tau(x_j)$ is 1, namely if the constraint is satisfied by the instantaneous assignment τ . If so, we discard the constraint; otherwise we keep it in C' .

When the search algorithm terminates, which provably happens as we shall state later, we have a finite automaton whose states have to be labelled as accepting or non-accepting. We choose the set of accepting states as those whose constraint set C contains *no* primitive until constraints. As a special case, when the given St-CSP has no “until” constraints, then all the states are accepting.

We stress again that our algorithm requires minimal changes from previous work to support the use of “until” constraints in St-CSPs. The only changes we have are lines 6–8 for the treatment of primitive until constraints, as well as how we pick the set of accepting states.

We first show that the dominance detection procedure is sound. To do so, we show that from a parent search node (C, h) and an instantaneous assignment τ , CONSTRUCT computes a child node (C', h') representing the correct shifted view. Thus, if two search nodes are syntactically equivalent, the corresponding shifted views must also be equivalent.

Theorem 1 (Soundness of dominance detection). *Suppose the constraint set C' of the shifted view of child node $\hat{Q}(k+1)$ is output by CONSTRUCT from the constraint set C of the parent node $\hat{P}(k)$ and the instantaneous assignment τ_k . Then $\text{sol}(C' \cup \{c_2\}) = \text{sol}(\{c \cap \pi_{\text{Scope}(c)}(c_1) \mid c \in C\}(1, \infty))$ where c_1 is the constraint stating $x(0) = \tau_k(x)$ for all streams x , and c_2 is the constraint stating $x_j(0) = \tau_k(x_i)$ for all constraints $x_i == \text{next } x_j$ in C (and hence C'). Note that c_2 is enforced by the set of historic values h' produced by CONSTRUCT.*

Proof (Sketch). The two solution sets share the same primitive pointwise constraints. Primitive next constraints in C (and C') are respected in both solution sets because of the constraint c_2 . Primitive until constraints in C are either preserved in C' or removed by CONSTRUCT depending on τ_k . Hence the constraint sets are either both constrained by an until constraint or both are not.

Having analysed the dominance detection algorithm, we can leverage the results to prove termination and soundness of the overall search algorithm.

Theorem 2 (Termination). *Using this new dominance detection procedure, the search algorithm always terminates.*

Proof (Sketch). Overall, the search algorithm can produce only finitely many syntactically distinct search nodes, and thus always terminates.

Theorem 3 (Soundness and Completeness). *The resulting solution automaton \mathcal{A} accepts the same language $L(\mathcal{A})$ as the solution set $\text{sol}(P)$ of the input St-CSP P .*

Proof (Sketch). $L(\mathcal{A}) \subseteq \text{sol}(P)$: The search algorithm ensures that primitive pointwise and next constraints are satisfied. Primitive until constraints are also satisfied by streams in the language by our choice of accepting states.

$\text{sol}(P) \subseteq L(\mathcal{A})$: Follows from Theorem 1 and induction on time points.

3.3 Automaton Pruning

As a post-processing step, we prune all states that cannot reach any accepting states via a flood-fill algorithm taking time linear in the size of the automaton (before pruning), which retains the accepted language by the following lemma.

Lemma 1. *Given a solution automaton \mathcal{A} , let \mathcal{A}' be obtained from \mathcal{A} by removing all states not reaching any accepting states. Then $L(\mathcal{A}) = L(\mathcal{A}')$.*

Furthermore, the pruning gives us the following guarantee about finite runs of the resulting automaton.

Theorem 4. *For any finite-length run of the generated and pruned solution automaton \mathcal{A} , corresponding to a finite string (stream prefix) p , there exists a solution stream $s \in L(\mathcal{A})$ such that p is the prefix of s of length $|p|$.*

Proof (Sketch). Every finite run can be extended, inductively by the fact that each state can reach an accepting state.

Intuitively, the theorem says that, no matter how we run the automaton, we can always extend the (finite) stream prefix *generated so far* into an infinite-length solution stream. This therefore also guarantees that it is *sound* to generate solution streams by running the automaton.

We emphasise that this pruning is for soundness, not solving efficiency.

4 The @ Operator

With the introduced “until” constraint along with a new solving algorithm, we can now model in St-CSPs conditions that need to be *eventually* satisfied. However, eventuality constraints might not be suitable for all application scenarios. It could be vital to be able to impose a strict upper bound on when a condition is satisfied, whilst with an eventuality constraint, the time at which a specified condition is satisfied could be arbitrarily far into the future.

Lee and Lee [12] propose using a constraint of the form “`first next ... next goal == 1`” to model this bound, reflected by the number of `next` operators in the constraint as the time bound. There are, however, two disadvantages to this approach. First, such a constraint has its own structure that we could not exploit to improve solving if we were to simply use the above syntax and current solving algorithms. Second, the notation is cumbersome, with the length of the constraint scaling linearly with the upper bound we wish to impose. To remedy these two issues, we propose a new temporal operator “@” that acts as syntactic

sugar, and further give another modification to the solving algorithm (more concretely, the dominance detection algorithm) to solve constraints involving the $\textcircled{}$ operator efficiently. We note however that, since the $\textcircled{}$ operator is simply a sugar, it does not enhance the expressiveness of the St-CSP framework.

Definition 3 (The $\textcircled{}$ operator). *Given a stream x (where x is instantiated or is some expression even involving stream variables) and a number $t \geq 1$, the stream $x\textcircled{t}$ is defined as the constant stream $(x\textcircled{t})(i) = x(t)$ for all $i \geq 0$. Equivalently, it is defined as `first next ... next x` , where there are t many `next` operators.*

We require that, for the purpose of this paper, the $\textcircled{}$ operator to take only a concrete number, instead of a variable, for its second parameter t . Our solving algorithm relies crucially on this assumption.

4.1 Modified Constraint Normalisation

We first augment the constraint normal form to allow for primitive $\textcircled{}$ constraints: $x_i == x_j\textcircled{t}$, where $t \geq 1$.

Accordingly, we add the following rewriting rule to the constraint rewriting system presented in Sect. 3.

- $(C_0 \cup \{c[\text{expr}\textcircled{t}]\}, C_1) \rightsquigarrow (C_0 \cup \{c[x_1], x_2 == \text{expr}\}, C_1 \cup \{x_1 == x_2\textcircled{t}\})$, where x_1 and x_2 are fresh auxiliary stream variables.

This new rewriting system is also terminating, Church-Rosser and sound. The proofs are essentially identical to those in Sect. 3.

4.2 Changes to Dominance Detection

Having introduced the $\textcircled{}$ operator, we adapt the function `CONSTRUCT` by describing how primitive $\textcircled{}$ constraints are modified when we construct a child search node from its parent. Given a primitive $\textcircled{}$ constraint “ $x_i == x_j\textcircled{t}$ ” from a parent node, we consider two cases.

- If $t > 1$, then we include “ $x_i == x_j\textcircled{(t-1)}$ ” in the new constraint set.
- If $t = 1$, then we include “ $x_i == \text{first } x_j$ ” instead.

This modification is orthogonal to those for the “until” constraint. This new dominance detection procedure (namely `CONSTRUCT` and `AREEQUAL`) is again sound, and induces a terminating, sound and complete overall search algorithm. The proofs are again essentially same as those in Sect. 3.

5 Experimental Results

We performed experiments in two settings to demonstrate the competitiveness of our approaches: (1) solving the Missionaries and Cannibals logic puzzle and

(2) solving a standard path planning problem on grid instances. For each setting, we solve for plans that eventually attain the goal using the “until” constraint in the model, as well as for bounded-length plans using the \textcircled{C} operator.

For the “until” experiments, we compare our approach to a standard CP approach proposed by Apt and Brand [1]. Their approach creates a *series* of finite domain CSPs, each corresponding to a finite horizon into the future, asking if the eventuality condition is satisfiable within the horizon. The time bound is incremented until the resulting CSP becomes satisfiable. (The idea was also used by van Beek and Chen [2], who credit Kautz and Selman [9].) As a result, if there is no upper bound a-priori on the minimum length of successful plans, this approach may not terminate. However, in the two settings we consider, such upper bounds do exist, and so we also experimented on using a CP solver to solve for a plan of exactly that length at the upper bound.

For the bounded-length plans scenario, we compare the use of the \textcircled{C} operator to the use of the `first next ... next` operator phrase, as well as to using a standard CP approach of solving the corresponding finite domain CSP.

All our experiments were run on an Intel Xeon CPU E5-2630 v2 (2.60 GHz) machine with 256 GB of RAM, with a timeout of 600 s. We used Gecode v6.0.0 as our finite domain CP solver. We also configured both the St-CSP solver and Gecode to *not* output the solutions to the file system, so as to minimise the impact of file I/O on time. The Gecode solver selects variables using the input order and according to the time point, which is the same as how the St-CSP solver label stream variables. Values are assigned the min value first. We tried fail-first for Gecode, but the results are less competitive.

5.1 Missionaries and Cannibals

In the Missionaries and Cannibals problem, there are n missionaries and n cannibals trying to cross a river from one bank to another, using a boat of capacity b people. There are three constraints in this problem: (1) at any time, there could be at most b people on the boat, (2) there must be at least one person on the boat on every trip and (3) for each bank, if there are any missionaries, then the cannibals cannot outnumber the missionaries; otherwise the missionaries will perish. The success condition is when everyone ends up on the other bank.

Table 1 shows the experimental results, when we solve using the St-CSP solver for *all* valid transportation plans that *eventually* attains the goal. Rows and columns in the table give different values of n and b respectively. Each entry in the table denotes the solving time in seconds for the test case. The results show that our solver is able to solve the problem for reasonably large instances without suffering from exponential increases in runtime.

We also performed experiments using the Apt and Brand framework [1] that uses traditional finite domain CP solvers. **Such CP approach timed out on all these instances.** On the other hand, for this particular problem there is, in fact, an upper bound on the number of steps of $n(b + 1)$ if a feasible plan exists. We used a CP solver to solve for plans of such length, and because of the simple

Table 1. Missionaries and Cannibals: “until”

	$b = 4$	$b = 5$	$b = 6$	$b = 7$	$b = 8$
$n = 40$	1.456	1.939	2.307	2.537	2.959
$n = 60$	4.459	5.831	7.417	9.081	10.698
$n = 80$	9.979	13.45	17.324	21.356	26.229
$n = 100$	19.053	26.044	33.747	42.16	53.112
$n = 120$	33.56	44.782	59.113	73.335	91.351
$n = 140$	51.623	70.666	92.744	118.407	146.325
$n = 160$	76.532	105.341	139.212	175.149	219.134
$n = 180$	110.122	149.741	196.743	250.56	313.35
$n = 200$	150.137	207.466	274.537	348.243	436.469
$n = 220$	201.308	277.219	363.592	463.509	–
$n = 240$	259.773	360.413	474.005	–	–

structure in the constraints, the solver was able to terminate under 15s in all these instances, outperforming our approach.

The next set of experiments replaces the “until” constraint that *eventually* everyone is on the other bank with the condition that the goal must be satisfied at time t , which is a value we vary between test cases. Because the St-CSP model is modified, requiring different solving times, the range of parameters (n, b) we experimented on is also different.

Table 2(a) shows the experimental results comparing the @ operator against **first next ... next**. Each table entry again shows the solving times using the new and old approaches respectively, separated by a “/”, with “–” denoting a timeout. The results demonstrate our implementation significantly outperforming the previous approach, with up to 2 orders of magnitude speedup.

Table 2. Missionaries and Cannibals: Time bounded

(a) @ vs **first next ... next**

(n, b)	$t = 10$	$t = 40$	$t = 70$	$t = 100$
(20, 5)	0.64/49.68	4.04/–	9.21/–	14.84/–
(30, 6)	1.71/178.68	16.33/–	36.23/–	56.76/–
(40, 7)	4.01/454.98	38.55/–	95.19/–	152.79/–
(50, 8)	9.07/–	100.34/–	236.58/–	374.07/–
(60, 9)	17.31/–	183.89/–	461.51/–	–/–
(70, 10)	32.25/–	371.57/–	–/–	–/–

(b) CP approach

(n, b)	$t = 10$	$t = 40$	$t = 70$	$t = 100$
(20, 5)	0.663	0.435	0.562	1.075
(30, 6)	0.435	0.560	0.780	1.011
(40, 7)	0.562	0.519	0.799	1.139
(50, 8)	0.762	0.521	0.767	1.102
(60, 9)	1.002	0.501	0.835	0.975
(70, 10)	1.425	0.526	0.873	0.1109

For the reader’s reference, we also include Table 2(b), that is the solving time of Gecode finding a single solution/plan for the time-bounded scenario. Since St-CSP solvers find *all* solutions, it is reasonable to not be competitive with a traditional CP approach. However, when we asked for *all* solutions instead, **Gecode timed out for all but the $t = 10$ instances**, since the St-CSP search

algorithm is able to avoid repeating equivalent search, via dominance detection. Asking a St-CSP solver to decide only the *existence of some* solution, instead of solving for all solutions, is scope for future work.

5.2 Path Planning in Grid World

The second set of experiments uses the path finding problem defined by the St-CSP model presented in Fig. 1. We generate random grid worlds of size $n \times n$ by independently sampling each directed edge between adjacent cells with probability p , as well as uniformly sampling the start and end points on the grid. Similarly, we performed two sets of experiments, solving for plans that eventually reach the goal (using the “until” constraint), and plans that have to reach the goal within a certain number of steps (using the @ operator).

For the “until” experiments, we varied both n and p , sampling 50 random instances for each setting of n and p . Figure 2(a) shows the average solving time of the test instances, where instances that timed out count as 600 s. The solving times in this setting increase in n polynomially, and become concave for larger n and p when a substantial number of instances start timing out.

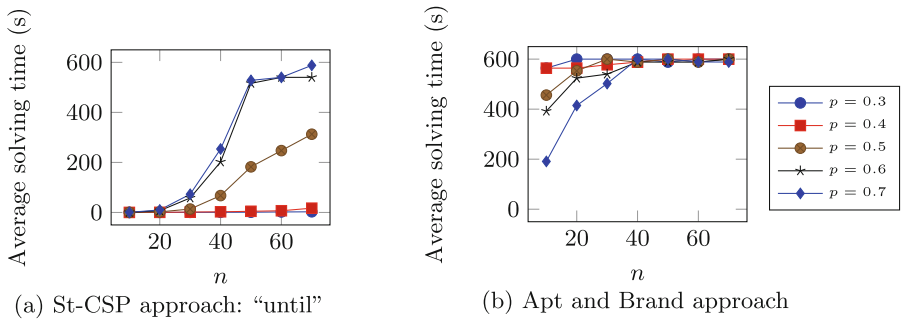


Fig. 2. Path planning: Eventuality condition

For comparison, Fig. 2(b) shows the solving time using the Apt and Brand [1] framework. The figures show that most of the instances timed out, demonstrating that the St-CSP approach is far more efficient. Since any simple path on the grid has an upper bound of n^2 in length, similarly to the previous setting we also used a CP solver to solve for a plan of length n^2 . However, Gecode runs into memory issues around $n = 40$, exceeding the 256 GB memory available. Even before so, for $n = 10$ a significant proportion of the instances already timed out, even though the St-CSP solves them almost instantaneously (as in Fig. 2(a)). Because of the memory issues that Gecode ran into, we decided to not give corresponding runtime plots since runtime is ill-defined.

For our last set of experiments, we again replace the “until” constraint with the constraint that the path must have visited the end point by t steps, a parameter that we vary across test cases. We generated 50 random instances for a

selected set of n values, however fixing $p = 0.8$ to make sure that a sizeable portion of the instances are satisfiable. We further varied t on these instances.

Figure 3(a) shows the average solving times by the old and new St-CSP approaches. We observe a 2 orders of magnitude improvement in solving time for large t . The plots for the @ operator are also in general better behaved. We further found that the reason for the essentially horizontal plots for the “**first next ... next**” operator phrase is due to it only being able to solve the trivially unsatisfiable instances in under 1s, where the reachable component from the start point is small. All the other cases timed out, giving the plateau we observe in solving time for the operator phrase.

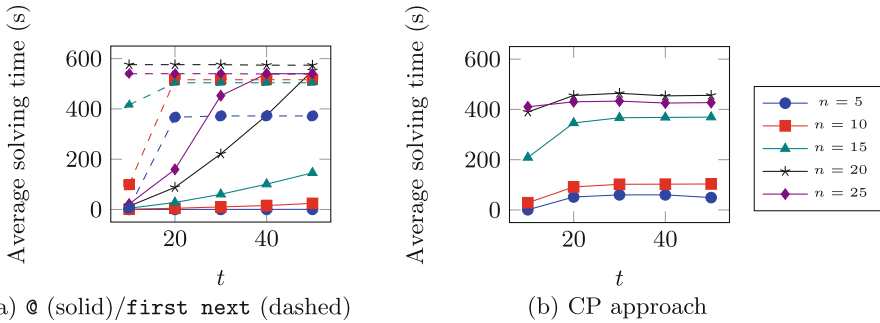


Fig. 3. Path planning: Time bounded

Figure 3(b) shows the solving time using Gecode. The plots display similar plateauing behaviour as our old approach, only starting earlier at $t = 20$. In comparison, the St-CSP approach is competitive with Gecode, despite the St-CSP solver being a prototype. We believe that it is due to the inherent specification complexity of the path planning problem on the grid. The entire graph structure has to be encoded for each time point, meaning that for the CP approach, the program is of size $O(tn^2)$, whereas the St-CSP is only of size $O(n^2)$.

6 Concluding Remarks

Our work improves the expressiveness of the St-CSP framework by augmenting it with (1) the new “until” constraint construct, adapted from the corresponding LTL operator, and (2) the @ operator, which is a syntactic sugar for **first next ... next** that further allows for faster solving by exploiting the special structure of the expression. We give corresponding new St-CSP solving algorithms, and also experimental evidence for their competitiveness with the corresponding CP approaches using Gecode. In our opinion the @ operator and the “until” constraint are for different purposes. The former is for time bounded scenario, while the latter is useful, for example, from a security perspective: we wish to know our adversary can never achieve sinister goal regardless of time budget.

By introducing the “until” constraint, we altered the structure of the generated solution automata and the guarantee we give regarding the execution of the automata (Sect. 3.3). From the statement that every run of the automaton is an accepting run, we weaken the guarantee (*whilst maintaining practical relevance*) to such that every finite run of the automaton could be extended to an infinite length solution stream. A natural direction for further investigation is to consider, under this weaker guarantee, how much more expressive can the St-CSP framework become. Are there other practical and natural constraints or temporal operators that, despite being currently inexpressible in the St-CSP framework, can be introduced with a solving algorithm that provides the above guarantee? Can we identify even weaker, yet still practically relevant guarantees that allows for even more expressiveness in the framework? We leave the answering of these questions for future work.

References

1. Apt, K.R., Brand, S.: Infinite qualitative simulations by means of constraint programming. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 29–43. Springer, Heidelberg (2006). https://doi.org/10.1007/11889205_5
2. van Beek, P., Chen, X.: CPlan: a constraint programming approach to planning. In: Proceedings of AAAI 1999/IAAI 1999, pp. 585–590 (1999)
3. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Mac Lane, S., Siefkes, D. (eds.) The Collected Works of J. Richard Büchi, pp. 425–435. Springer, New York (1990). https://doi.org/10.1007/978-1-4613-8928-6_23
4. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers Inc., San Francisco (2003)
5. Emerson, E.A.: Temporal and modal logic. In: Handbook of Theoretical Computer Science (vol. B), pp. 995–1072. MIT Press, Cambridge (1990)
6. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., San Francisco (2004)
7. Golden, K., Pang, W.: Constraint reasoning over strings. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 377–391. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45193-8_26
8. Harabor, D., Grastien, A.: Online graph pruning for pathfinding on grid maps. In: Proceedings of AAAI 2011, pp. 1114–1119 (2011)
9. Kautz, H., Selman, B.: Planning as satisfiability. In: Proceedings of ECAI 1992, pp. 359–363 (1992)
10. Kilby, P., Prosser, P., Shaw, P.: A comparison of traditional and constraint-based heuristic methods on vehicle routing problems with side constraints. *Constraints* 5(4), 389–414 (2000)
11. Lallouet, A., Law, Y.C., Lee, J.H.M., Siu, C.F.K.: Constraint programming on infinite data streams. In: Proceedings of IJCAI 2011, pp. 597–604 (2011)
12. Lee, J.C.H., Lee, J.H.M.: Towards practical infinite stream constraint programming: applications and implementation. In: O’Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 449–464. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_34
13. Pesant, G., Gendreau, M., Potvin, J.Y., Rousseau, J.M.: An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transp. Sci.* 32(1), 12–29 (1998)

14. Pnueli, A.: The temporal logic of programs. In: Proceedings of FOCS 1977, pp. 46–57 (1977)
15. Sturtevant, N.R.: Benchmarks for grid-based pathfinding. *IEEE Trans. Comput. Intell. AI Games* **4**(2), 144–148 (2012)
16. Wadge, W.W., Ashcroft, E.A.: LUCID, the Dataflow Programming Language. Academic Press Professional Inc., San Diego (1985)
17. Winskel, G.: *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge (1993)



A Complete Tolerant Algebraic Side-Channel Attack for AES with CP

Fanghui Liu, Waldemar Cruz, and Laurent Michel^(✉)

Computer Science and Engineering Department, School of Engineering,
University of Connecticut, Storrs, CT 06269-4155, USA
{fanghui.liu,waldemar.cruz,laurent.michel}@uconn.edu

Abstract. Tolerant Algebraic Side-Channel Attack (TASCA) is a combination of algebraic and side-channel analysis with error tolerance. Oren et al., used mathematical programming to implement TASCA over a round-limited version of AES. In [7], Liu et al. revisited their results and introduced a TASCA-CP model that delivers solutions to this 1-round relaxation with orders of magnitude improvement in both solving time and memory consumption.

This paper extends the result and considers TASCA for the full 10-rounds AES algorithm. Two approaches are introduced: staged and integrated. The staged approach uses TASCA-CP as a spring board to enumerate and check its candidate solutions against the requirements of subsequent rounds. The integrated model formulates all the rounds of AES together with side-channel constraints on all rounds within a single unified optimization model. Empirical results shows both approaches are suitable to find the correct key of AES while the integrated model dominates the staged both in simplicity and solving time.

Keywords: Algebraic side-channel attack · AES
Cryptography · Block cipher · Constraint programming · Optimization

1 Introduction

Side-Channel Analysis (SCA) is a type of attack that exploits the physical properties of a device performing a cryptographic operation, the goal of which is to obtain secret information (e.g., a secret key) from a cryptographic system. A side-channel attack typically needs hundreds to thousands of power traces to reduce the sensitivity to noise from measurement or decoding. *Algebraic* Side-Channel Attack (ASCA) was introduced in [19] to combine algebraic cryptanalysis with side-channel attack. It was applied to AES in [20]. Compared to SCA, ASCA requires much less power traces ([20] showed that as little as a single power trace is enough to recover the secret key of AES). However, the success of ASCA heavily depends on the accuracy of the side-channel information because it does not tolerate power trace measurement errors. Improved ASCA (IASCA) [12] improved the performance of ASCA on AES by optimizing AES and algebraic

representation, it also introduced a method for error handling. Tolerant Algebraic Side-Channel Attack (TASCA) [14] transforms a side-channel analysis problem to a pseudo-boolean optimization problem where the objective is the minimization of the total deviation from the measured side-channel signal.

The TASCA for AES, as presented by Oren et al. [16], models the encryption algorithm as a series of pseudo-boolean equations and side-channel constraints (i.e., the Hamming weights representing the side-channel signal). The formulation allows for providing either the plaintext alone, or both the plaintext and the ciphertext. The key was recovered with up to 20% of error rate, which refers to the probability that the measured Hamming weights are incorrect. The key is counted as correct if four or less bytes of the sixteen bytes of the key are incorrect.

In 2017, Liu et al. [7] revisited the TASCA attack on AES and offered a CP formulation based on bit-vectors [8]. Their model, like that of Oren et al. [16], focused on the first round of AES and provided an approach that attempts to recover the key with low complexity. By using CP over bit-vectors with a customized search, it was possible to recover candidate keys for the one round problem with orders of magnitude improvements both in runtime and memory usage over the pseudo-boolean optimization and Integer Programming (IP) approaches. Yet, relying on only one round of AES and side-channel information is a relaxation of the true problem that mandates a brute-force post-processing to zero-in on the true key and rule out candidate solutions that are not conforming to the requirements of the full AES algorithm and side-channel information.

Several approaches were proposed to apply Constraint Programming to crypt-analysis. [18] used a CSP framework to design substitution functions for substitution permutation (SP) networks. [4] introduced a chosen key differential crypt-analysis using Constraint Programming models, and it was performed against AES-128 in [5], AES-192 and AES-256 in [3]. Based on [4,21] apply CP to search for differential/linear characteristics, integral distinguishers and performed the analysis on AES, PRESENT [2] and SKINNY [1].

This paper explores extensions of the CP approach to natively handle the full 10 rounds of AES and side-channel information. The paper presents two approaches for doing so and evaluates them empirically alongside the natural extensions that could be offered for the earlier pseudo-boolean formulation. The *staged* approach uses TASCA-CP as a “black box” subroutine to produce solutions in the full model. The *integrated* approach extends the constraint programming model to directly handle all 10 rounds of AES alongside the side-channel information for those rounds.

The remainder of the paper is organized as follows. Section 3 recalls the 1-round TASCA approach and how it is meant to be used. Section 4 articulates the limitations of the restricted formulation, the source of relaxations and the contributions of the paper. Section 5 is the core of the paper and presents two extensions needed to consider AES in full. Section 6 discusses the empirical results while Sect. 7 concludes.

2 AES Overview

AES is an iterated block cipher that supports a fixed block size of 128 bits, and key size of 128, 192 and 256 bits. The version used in this paper is AES with 128-bit cipher key (AES-128). The cipher key is first expanded into 10 round keys via key expansion [13]. The plaintext is separated into blocks of 16 bytes, denoted as $p_0 \dots p_{15}$ (where $p_i \in \{0, 1\}^8, i = 0, \dots, 15$). Each block is represented by a 4×4 matrix of bytes.

There are four elementary operations that build the whole function of AES:

- **SubBytes:** Apply a permutation on the input bytes using an “S-Box” [11].
- **ShiftRows:** Apply a byte-wise left-rotate operation for each of the four rows by 0, 1, 2, and 3 bytes.
- **MixColumns:** Multiply each column in the state with a matrix of constants.
- **AddRoundKey:** XOR the state with the round key.

During the encryption, the plaintext is first combined with the initial round key (cipher key) and then goes through 9 rounds of iteration consisting of 4 subrounds (SubBytes, ShiftRows, MixColumns, AddRoundKey), and the last iteration consist of 3 subrounds (without MixColumns) to produce the ciphertext $c_0 \dots c_{15}$. The AES encryption process is outlined in Algorithm 1. More details of AES can be found in [13].

Algorithm 1. Pseudocode for AES Encryption Algorithm

```

1: function AES(byte in[16], byte out[16], key_array round_key[Nr+1])
2:   byte state[16]
3:   state = in
4:   AddRoundKey(state, round_key[0])
5:   for i = 1 to Nr-1 do
6:     SubBytes(state)
7:     ShiftRows(state)
8:     MixColumns(state)
9:     AddRoundKey(state, round_key[i])
10:  SubBytes(state)
11:  ShiftRows(state)
12:  AddRoundKey(state, round_key[Nr])
13:  out = state
14: return out

```

3 TASCAs over Restricted AES

TACSA-CP presented a tolerant algebraic side-channel attack using constraint programming. The TASCAs-CP model includes one round AES structural constraints, one round side-channel constraints (which contains 10% errors) and

auxiliary constraint (e.g. plaintext). The goal is to minimize the errors in the side-channel information. With a customized search, the TASCAs-CP is able to find the global optima with an overwhelmingly better performance compared to pseudo-boolean model with SCIP [16] and IP model with Gurobi [7].

The global optima is then given to a CP solver to search for all candidate solutions. An attacker needs to enumerate over the candidate solutions to discover the true key used in the captured AES encryption process.

4 TASCAs over Full AES

The previous section showed that it is possible to carry out a Tolerant Algebraic Side-Channel Attack using CP on one round AES encryption with one round side-channel information. The empirical results established that the performance of TASCAs-CP is significantly better than the IP models making CP the technology of choice to conduct this type of side-channel attacks. Nonetheless, it is *essential* to remember that the TASCAs approach considered earlier remains a resolution technique for a *relaxation* of the true problem. Indeed, with only one round of the AES computation being modeled, one can only find *candidate solutions*. Those candidate solutions may not satisfy the structural constraints imposed by the subsequent rounds of AES and they may deliver a ciphertext that is different from the sought after solution. Specifically, it is worth noting that there are two distinct relaxations.

Round. The 1-round relaxation was essential in the case of the integer programming approach to sidestep difficulties arising from the sheer size of the model. Yet, the relaxed problem is still challenging and requires non-trivial efforts. While the relaxation will produce all the bits of the key and fix all the bits of the output state of round 1, those output bits should still be subjected to three classes of constraints: (1) the structural constraints of state derivation dictated by AES; (2) the side-channel constraints based on the measurements done in rounds 1–10; and (3) the equality to the target ciphertext when the attack is carried out with a known pair cleartext, ciphertext. This limitation can still be acceptable if the number of candidate solutions remains small, in which case an exhaustive check can verify each candidate solution. Yet, observe that since this is a relaxation, it is quite possible that *all candidate solutions* associated to the global optima of the relaxation are infeasible in the full problem. This indicates that the global optimum of the full problem has an objective value worse than what was reported by the relaxation. The instances considered in [7] were engineered to have a tight relaxation, i.e., the global optimum of the relaxation was also the global optimum of the true problem.

Tolerance. Both the MIP and the CP models from [7] consider that side-channel measurements for a state byte b are hamming weights estimates ($\hat{H}(b)$) and may be off from their true value $H(b)$ by ± 1 , i.e., for any state byte b , $\hat{H}(b) - 1 \leq H(b) \leq \hat{H}(b) + 1$. With both technologies, the optimization models see

a byte b as a decision variable that must satisfy these inequalities. If the measurement estimates are off by a wider margin, the true solution can never be found since the model will exclude it.

The purpose of this paper is to articulate a solution methodology that mitigates the shortcomings introduced by these relaxations to obtain a direct resolution technique capable of computing the key used by an AES encryption of a known plaintext. Specifically, the methodology must scale to support all 10 rounds of AES, exploit any additional side-channel measurements provided for those rounds, and gracefully handle the tolerance bounds. Note that the bit-vector formulation employed by the CP model holds the promise of a scalable representation as a single round consumes from 20 to 40 times less memory and delivers runtimes that are two order of magnitude better than state-of-the-art mathematical programming solvers.

5 Approaches

The remainder of the paper considers two natural approaches and articulates the rationale behind them and the needed improvements to make them competitive. Before describing the two approaches, a few preliminaries are in order.

5.1 Preliminaries

At a macroscopic level, the optimization model for one AES round uses decision variables to represent the input state of the round, the output state of the round and the round key. Its constraints define the relations connecting the input state and round key through intermediary states to the output state of the round. They also constrain the hamming weights of each state to be within the error tolerance requirements. The objective function collects all the deviation errors, positive or negative, that can be experienced to match the hamming weights with their estimates.

Definition 1 (State Variables). *Let S_k denote the 128-bit wide bit-vector variable denoting the input state to round $k \in 1..10$. Similarly, let O_k denote the 128-bit wide bit-vector variable denoting the output state of round k . Finally, let $I_{k,r}$ denote the 128-bit wide bit-vector variable denoting the internal state r within round k . Namely, this is the state after each stage encountered within a round of AES, e.g., the add round key, S-box and mix column stages ($r \in 1..4$).*

Definition 2 (Key Variable). *Let K_k denote the 128-bit wide bit-vector variable denoting the round key for round k .*

Definition 3 (Byte structure). *Given a state variable S (a 128-bit wide bit-vector), let S^b be a byte corresponding to the 8-bit subsequence of S aligned on an 8-bit boundary.*

Clearly, any state variable S is broken down into its 16 constituent bytes, numbered 0 through 15.

Definition 4 (Error Variable). $E_{k,r}^{b+}$ and $E_{k,r}^{b-}$ are the non-negative slack variables modeling noise in a side-channel equation. The binary variable $E_{k,r}^{b+}$ models the positive error while $E_{k,r}^{b-}$ models the negative error for byte $b \in 0..15$ of round $k \in 1..10$ and state $r \in 1..4$.

Definition 5 (Hamming Weight Estimates). Let $\tilde{H}(S^b)$ denote the estimate of the hamming weight for any state byte S^b ($b \in 0..15$) of a state S .

Definition 6 (Hamming Weights). Let $H(S^b)$ denote the actual hamming weight of byte $b \in 0..15$ for a state S .

AES Constraints

- AddRoundKey $ARK(X, Y, Z)$ is implemented as a bit-wise XOR operation between two 8-bit bit-vector variables, $X \oplus Y = Z$.
- SubBytes $SubByte(X, Z)$ is implemented as an *element constraint* [7] over bit-vectors. The element constraint $Z = c[\mathcal{I}(X)]$ takes in an array c of (constant) bit-vectors and requires Z to be equal to the $\mathcal{I}(X)^{th}$ entry of the array c . An array of 256 constant bit-vectors model the full substitution box.
- ShiftRows $ShiftRows(X, Z)$ is a logical circular shift on input variable and there are no changes in the values. Therefore ShiftRows operation does not leak any side-channel information and is combined with MixColumns [15].
- MixColumns $MixColumns(X, Z)$ is a complex operation that applies to a column of input variable matrix at a time. An 8-bit efficient MixColumns implementation [17] is used in CP encoding. Suppose $[a_0, a_1, a_2, a_3]$ is a column of input X , $[o_0, o_1, o_2, o_3]$ is a column of output Z , the MixColumns operation can be expressed as follows:

$$o_k = \left(\bigoplus_{i=0}^3 a_i \right) \oplus \mathbf{xtime}(a_k \oplus a_{(k+1) \bmod 4}) \oplus a_k \quad \forall k \in 0..3$$

bit-vector constraints are used to capture XOR as well as \mathbf{xtime} [13].

Side-Channel Constraint: are created using \mathbf{count} [7] constraint on bit-vectors. Using \mathbf{count} constraint the actual hamming weight $H(I_{k,r}^b)$ is encoded as $\mathbf{count}(I_{k,r}^b)$. The side-channel constraint is formulated as:

$$H(I_{k,r}^b) + E_{k,r}^{b+} - E_{k,r}^{b-} = \tilde{H}(I_{k,r}^b)$$

It is now possible to define a COP for round k .

Definition 7 (AES Round Model). Given a round $k \in 1..10$, the TASCA-CP model for round k is the COP

$$M_k = \langle X_k, C_k, f_k \rangle$$

Where

$$X_k = S_k \bigcup_{r \in 1..4} I_{k,r} \cup O_k$$

and

$$\begin{aligned} C_k &= ARK(S_k, K_k, I_{k,1}) && \cup SubBytes(I_{k,1}, I_{k,2}) && \cup \\ & ShiftRows(I_{k,2}, I_{k,3}) && \cup MixColumns(I_{k,3}, I_{k,4}) && \cup \\ I_{k,4} &= O_k && && \cup \\ \bigcup_{r \in 1..4, b \in 0..15} H(I_{k,r}^b) + E_{k,r}^{b+} - E_{k,r}^{b-} &= \tilde{H}(I_{k,r}^b) \end{aligned}$$

and the objective to minimize is s

$$f_k = \sum_{r \in 1..4, b \in 0..15} E_{k,r}^{b+} + E_{k,r}^{b-}$$

Constraints in M_k link the input state S_k to the output state O_k of round k .

Definition 8 (Whole Model). While C_k represents the AES constraints and side-channel constraints for round k , C represents the AES constraints and side-channel constraints for all 10 rounds. Similarly, f represents the overall sum of errors for 10 rounds:

$$f = \sum_{k \in 1..10} f_k$$

With the optimization models for the rounds of AES formalized, it is now possible to formally describe two approaches to solve the full AES model.

5.2 A Staged Approach

Consider the AES Round model $M_1 = \langle X_1, C_1, f_1 \rangle$ that captures the first round of AES. The COP model considered in [7] can be derived from M_1 by adding two additional sets of constraints to capture the *cleartext* requirement and the *key schedule*. The cleartext requirement is straightforward and consist in binding the bytes of S_1 to their value in the cleartext. The key schedule requirement derives the round keys K_1, K_2, \dots, K_{10} from the AES key K with invertible operations. Namely, the 16 bytes (byte number is the superscript) of the round key K_r are defined with

$$K_r^0 = SubByte(K_{r-1}^{13}) \oplus K_{r-1}^0 \oplus RC_r \tag{1}$$

$$K_r^1 = SubByte(K_{r-1}^{14}) \oplus K_{r-1}^1 \tag{2}$$

$$K_r^2 = SubByte(K_{r-1}^{15}) \oplus K_{r-1}^2 \tag{3}$$

$$K_r^3 = SubByte(K_{r-1}^{12}) \oplus K_{r-1}^3 \tag{4}$$

$$\forall i \in \{0, \dots, 11\} K_r^{i+4} = K_r^i \oplus K_{r-1}^{i+4} \tag{5}$$

where $r \in \{1, \dots, 10\}$, and $K_0 = K$. RC_r is a round-specific constant and $SubByte$ is the usual non-linear S-Box of AES. The key expansion function is

encoded as a series of cascading XOR constraints and substitutions via the element constraint. In the following $\text{KeySchedule}(K, [K_1, \dots, K_{10}])$ refers to the set of constraints implementing this key expansion requirement.

Therefore, the model in [7] is:

$$M'_1 = \langle X_1, C_1 \cup \text{ClearText}(S_1) \cup \text{KeySchedule}(K, [K_1, \dots, K_{10}]), f_1 \rangle$$

Clearly, this is a relaxation since this model ignores all but round 1. The resolution of that relaxation will produce a sequence of improving local optima culminating with the global optimum of the relaxation f_1^* . For any discrete value $v \in f_1^* \dots \frac{|E|}{2}$, it is possible to enumerate all candidate solutions that yield that specific objective with the *constraint satisfaction problem*

$$M_1^v = \langle X, C_1 \cup \text{ClearText}(S_1) \cup \text{KeySchedule}(K, [K_1, \dots, K_{10}]) \cup \{f_1 = v\} \rangle$$

$$M_{full} = \left\langle \begin{array}{l} X, C \cup \text{ClearText}(S_1) \cup \text{CipherText}(S_{41}), f \\ \cup \text{KeySchedule}(K, [K_1, \dots, K_{10}]) \end{array} \right\rangle$$

Algorithm 2. Staged Approach to solve the Full AES model

- 1: $v, f_1^* \leftarrow \text{optimize}(M'_1)$
 - 2: $Best \leftarrow +\infty$
 - 3: $F^* = \emptyset$
 - 4: **do**
 - 5: $Sols \leftarrow \text{solveAll}(M_1^v)$
 - 6: $F \leftarrow \{c \in Sols \mid \text{validate}(c, M_{full} \wedge f_1 = v \wedge f \leq Best)\}$
 - 7: $Best \leftarrow \min(Best, \min_{s \in F}(f(s)))$
 - 8: $F^* = \{s \in F^* \mid f(s) \leq Best\} \cup F$
 - 9: $v \leftarrow v + 1$
 - 10: **while** $v \leq \frac{|E|}{2} \wedge F^* = \emptyset$ **return** F^*
-

This observation gives rise to an algorithm for the full AES shown in Algorithm 2. The algorithm in line 1 produces the global optima of the relaxation f_1^* and the target value v . Line 2 sets the best objective function to ∞ and line 3 defines the set of solution F^* as empty. The loop covering lines 4–10 start by deriving (line 5) the solution pool obtained if the objective function for round 1 (f_1) is forced to adopt value v . Line 6 discards any candidate solution c from the relaxation that do not correspond to the current relaxed target v nor improve upon the incumbent value $Best$. This validation process clearly uses the full AES model M_{full} which features constraints for all the rounds of AES, all the side-channel information and auxiliary information (plaintext and ciphertext). The `validate` subroutine fixes the state variables of round 1 to their value in the candidate solution c and M_{full} has all the constraints for all the rounds, all the side-channels and its objective function f captures the sum of the errors

over all rounds. Lines 7 and 8 update the value of the incumbent and the set of solutions that deliver the optimal objective value which is stored in F^* . Line 9 finally increases the target value by 1 and the loop repeats the process. The terminating condition relies on a trivial upper bound for the objective function f equal to the number of error variables divided by 2 (no byte can have both a positive and negative slack) and whether F^* is empty. $F^* \neq \emptyset$ indicates that the optimal solution is found since both the plaintext and ciphertext are fixed in M_{full} and guarantee a valid and unique solution.

It is critical to distinguish the two objective functions. f_1 is the first round while f captures the sum of errors over all rounds. It is possible to encounter a solution of the relaxation that features many deviations from the hamming weights in round 1 – hence a poor value for the relaxed objective f_1 – but very few errors in subsequent rounds, ultimately delivering an excellent value for the global objective f . The subroutines in Fig. 2 are as follows

optimize(M) Solves M and produces a local optima f_1^* .

solveAll(M) Generates a solution pool for a fixed value of the objective.

validate(c,M) Verifies, in polynomial time, that c satisfies model M .

Analysis. In most cases, the round-1 relaxation happens to be tight. Namely, its candidate solutions for the relaxation optimum coincides with the global optimum when extended to the full AES and the algorithm produces a set F of cardinality 1 with a feasible and optimal solution in the first iteration. Subsequent iteration yield empty sets F as the candidate solutions of weaker solutions from the relaxation cannot be extended and meet the $f \leq Best$ requirement. Occasionally, the algorithm will have to consider weaker candidate pools that may contain globally feasible and better solutions. In the extreme case, the algorithm may have to expand considerable resources to eventually find out that, because of the *tolerance relaxation of ± 1* , none of the candidate solutions are globally feasible. In the worst case, the algorithm may perform $\mathcal{O}(|E|)$ iterations¹ since the objective function is bounded from above by $\frac{|E|}{2}$ and still not deliver a globally optimal solution because of the *tolerance relaxation*. In practice though, it performs reasonably well thanks to the tightness of the relaxation.

These limitations indicate that an alternative approach may be worthwhile.

5.3 Integrated Approach

The AES round model formulated in Definition 7 explicitly captures a single round k of AES with a COP $M_k = \langle X_k, C_k, f_k \rangle$ in which X_k contains input, output and internal state variables. It is therefore tempting to aggregate all these models since the output state variables of round k are the input state variables of round $k + 1$. Similarly, the objective function of the full AES is separable and *exactly* matches the sum of the objective functions. Namely $f = \sum_{k \in 1..10} f_k$. By accumulating all the variables, constraints and the additive objectives together

¹ Recall that E is the set of all error variables.

with the constraints for the key schedule, one can obtain a (large) mathematical formulation of the entire AES state transformation.

Definition 9 (AES Full Model). *Given n rounds, numbered 1 through n and their associated AES Round models M_1 through M_{10} , the AES Full Model for n rounds is the COP*

$$M(n) = \langle X, C, f \rangle$$

Where

$$X = \bigcup_{k \in 1..n} X_k \cup K$$

$$C = \bigcup_{k \in 1..n} (C_k \cup \{O_k = I_{k+1}\}) \cup \text{ClearText}(S_1) \cup \text{KeySchedule}(K, [K_1, \dots, K_n])$$

In which the objective is to minimize

$$f = \sum_{k \in 1..n} f_k$$

This formulation encapsulates in a single family of models all the details about AES and progressively eliminates the round relaxation as n increases from 1 to 10. It still depends on the *tolerance relaxation* though. Naturally, $M(10)$ coincides with the full AES model while $M(1)$ was detailed in [7]. As before, an $M(n)$ model can be submitted ‘as is’ to a constraint programming solver together with a suitable search, or it can be linearized for an IP solver.

Search Heuristic Revisited. The search heuristic is focused on the semantics of the AES transformations and the side-channel information. For each of the 16 bytes of the state variables, there is a hamming weight value attributed to each state. A set of candidate values are produced for each state variable, such that the hamming weight of the value is within the tolerance range as dictated by

$$M_r^i - k \leq H(S_r^i) \leq k + M_r^i$$

Where k is the allowable discrepancy between the measurement M_r^i of byte i in round r and the candidate value for the i^{th} byte of the state variable S_r . Recall that $H(b)$ simply counts the number of bits at 1 in byte b , i.e.,

$$H(b) = \sum_{i \in 0..7} (b \wedge (1 \ll i) = (1 \ll i))$$

The goal of TASCA is to minimize the number of deviations from the predicated hamming weight value delivered from the side-channel analysis. The search heuristic is responsible for driving the variable/value choices to reach the optimal solution early. Value assignments that contribute the least to the objective function will be effective in reaching high-quality solution early.

Variable Heuristic. The search introduced in TASCA-CP used branching on multiple variables at once. Branching on a pair of variables allows for calculation of high-quality *under-estimates* of the actual errors. The main intuition is that branching on a pair of variables allowed for the propagation to fix the values of connecting variables via the AES transform constraints.

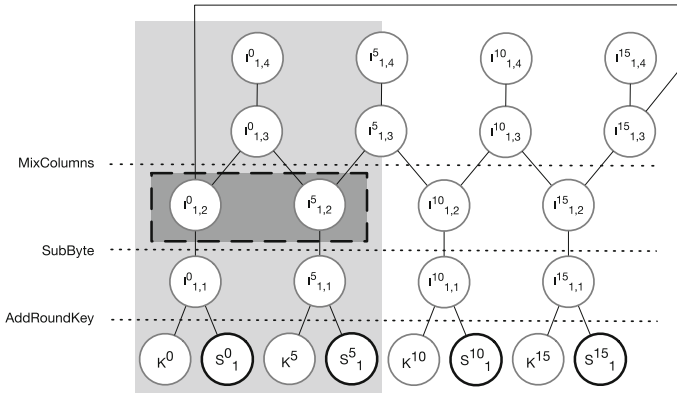


Fig. 1. Circuit for bytes {0, 1, 2, 3}.

Figure 1 offers a schematic view of the search for one quarter of the circuit modeling round 1 of AES. The schematic is best understood bottom-up. At the bottom layer, four bytes of the plaintext (Bold variables S_1^0 , S_1^5 , S_1^{10} and S_1^{15}) are combined via the **addRoundKey** constraints with the matching bytes of the round key (K^0 , K^5 , K^{10} and K^{15}) to produce the intermediate state $I_{1,1}^0$, $I_{1,1}^5$, $I_{1,1}^{10}$ and $I_{1,1}^{15}$. The internal state variables are mapped via the **SubByte** constraints to $I_{1,2}^0$, $I_{1,2}^5$, $I_{1,2}^{10}$ and $I_{1,2}^{15}$. These are then composed via the **mixColumn** operation which links the last two sets of internal state bytes (a mix of XOR and shifts). Note again, that the last internal state variables are simply made equal to the output variables of the round. The main insight is that *given the known plaintext*, the **addRoundKey** and **SubByte** transformations are bijective. Therefore labeling any variables connected to the **addRoundKey** and **SubByte** constraints will fix the other variable via propagation. In particular, labeling variable $I_{1,2}^0$, will, through propagation, fix the $I_{1,1}^0$ and K^0 variables. Fixing $I_{1,2}^5$ at the same time would, likewise, propagate to $I_{1,1}^5$ and K^5 . Simultaneously fixing $I_{1,2}^0$ and $I_{1,2}^5$ would exploit the ternary constraint connecting the dotted box containing that pair to $I_{1,3}^0$ and help propagation fix $I_{1,4}^0$ and therefore O_1^0 allowing the search to get a fair estimate of the hamming weight errors associated to these variables.

Subsequently, the search will consider other pairs that reuse one of the two variables from the first pair. For instance, if $\langle I_{1,2}^0, I_{1,2}^5 \rangle$ was selected first, it is tempting to consider the two pairs $\langle I_{1,2}^{15}, I_{1,2}^0 \rangle$ and $\langle I_{1,2}^5, I_{1,2}^{10} \rangle$ as the domain of $I_{1,3}^{15}$ and $I_{1,3}^5$ are already reduced by the first choice.

Value Heuristic. The objective is driven by the sum of measurement errors on intermediate state variables. If the search considers a pair of values $\langle a, b \rangle \in D(I_{1,2}^0) \times D(I_{1,2}^5)$ it can assess the impact that the simultaneous assignments $I_{1,2}^0 = a \wedge I_{1,2}^5 = b$ would have on the errors at the intermediate state variables in the leftmost gray column. This assessment is an under-approximation of the true error induced by the assignments. Indeed the intermediate variable $I_{1,3}^5$ can expose errors caused by the choice of value b for $I_{1,2}^5$, but that falls outside the gray column and is therefore ignored.

Therefore the value heuristic considers pairs of values and assess the quality of the pair with a scoring function SC given as:

$$SC(\langle a, b \rangle) = SC_{leg}(a, [I_{1,2}^0, I_{1,1}^0, K^0]) + SC_{leg}(b, [I_{1,2}^5, I_{1,1}^5, K^5]) + SC_{mc}(a \oplus b, I_{1,4}^0)$$

The functions SC_{leg} and SC_{mc} model the error attributes to a “leg” ($[I_{1,2}^0, I_{1,1}^0, K^0]$) or to the top of the “leg” (the MixColumns operation).

Optimality Pruning. The objective function is defined as the total contribution of the errors in the predicted hamming weight values of the state variables. As the search continues to find an improving solution, the objective function will dive deeper reducing the number of errors required to find the next best satisfiable solution. At each incumbent solution, the search tree can prune any path that will not lead to an improving solution.

Overall Search. The search aims at labeling variables that contribute the least amount of error to minimize the objective function. This allows the solver to reach high quality solutions quickly. The strategy that governs the order in which search tree nodes are explored is a standard depth first search. When considering a model with multiple rounds, it makes sense to partition all the variables according to their round and branch on them in increasing round order. Within a round, the same search heuristic as before can be adopted (branching on pairs of variables).

Analysis. This search procedure is sufficient to experiment with a full AES model. Surprisingly, the first empirical results were disappointing. Indeed, the integrated approach needed a considerable search effort (10–100 times more search nodes) to deliver a global optimum often exceeding the time out limit of 10 min. The analysis of the result revealed that: (1) The search delivered only one globally optimal solution; and (2) The search did not deliver *any* sub-optimal solutions. These two observations lead to a conjecture on the search behavior:

Conjecture 1 (Degeneracy). The AES Full COP Model has only one feasible solution, even in the presence of a ± 1 tolerance relaxation.

If true, all candidate solutions from the round 1 are ultimately infeasible which propagation discovers causing backtracking early. Fundamentally, when branching in round order, once round 1 is complete, the solver *checks* the constraints of

the subsequent round as propagation alone fixes everything. In essence, candidate solutions emerging from round 1 can only be either extended to all rounds and deliver a global solution without further branching, or a constraint (i.e., a hamming weight constraint) will *fail* and cause backtracking.

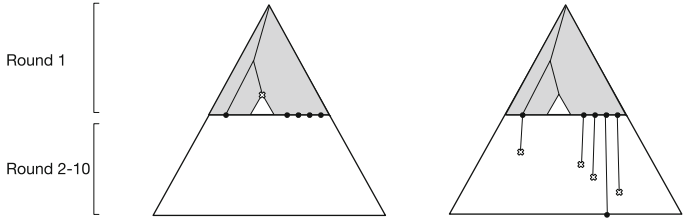


Fig. 2. Search tree comparison.

Figure 2 offers a depiction of two search trees. The left triangle represents the search tree for $M(1)$, i.e., the one-round model solved in [7]. The right triangle illustrates the search “tree” for $M(10)$, the AES full model. What it suggests is that the search over $M(1)$ delivered many feasible solutions that were used to bound away some subtrees like the small white triangle in the gray zone. However, in the right part of the picture, those solutions feasible at the boundary of round 1 become infeasible and therefore one never finds an incumbent and never tighten the upper bound. Consequently, the same little white triangle must be explored in the full model and this is what contributes to the significant increase in search effort.

Corollary 1 (DFS is unadapted). *If one (or very few) feasible solutions exist, the objective function can only be used to guide the search heuristic and it is essential to discover the (sole) feasible solution as early as possible.*

The search heuristic was designed specifically to exploit the objective function and minimize deviations from the prescribed hamming weights. Under the assumption that the heuristic is effective, a natural option is to consider *limited discrepancy search* [6] that slowly and progressively distrusts the search heuristic, authorizing a limited number of deviation from its recommendations.

Recall that there are 16 bit-vector variables to be labeled, which is a large search tree. If the feasible solution is on the far right side of the search tree, the DFS strategy will explore all nodes (left to right) before reaching the solution. Limited Discrepancy Search, by gradually increasing potential “wrong turns”, increases the likelihood to hit the feasible solution sooner. The implementation of LDS was detailed in [10]. Empirical results indicate that limited discrepancy search is quite effective on this problem.

6 Experimental Setup

The adopted CP solver is OBJECTIVE-CP [9] that combines modeling and search as well as user-defined searches. The IP approach relies on Gurobi (7.5.2). The experiments ran on a 16-core Intel Xeon E52640 at 2.40 Ghz with 16 MB cache and Ubuntu 16.04 LTS. 1000 instances were generated. Each instance is based on a randomly chosen plaintext and cipher key in $\{0, 1\}^{128}$. Each instance contains, for its plaintext, 996 Hamming weight leaks that correspond to the full 10 rounds. For each instance, a 10% error rate is uniformly applied to the 996 Hamming weight leaks, the Hamming weights are modified by ± 1 .

To evaluate the performance of the approaches considered in the paper, experiments are conducted with either 1, 2 or 10 rounds of AES encryption *and* side-channel information. In particular, the paper considers the integrated approach with the LDS and DFS search strategies as well as the staged approach.

6.1 IP Approach

The average runtime of IP using Gurobi with one round AES and side-channel information is 222 s with a standard deviation 135 s. This results matches what was reported in [7]. It becomes significantly harder for the IP solver when solving with 2 rounds of AES and side-channel information. First, ten easy (10 s), medium (300 s) and hard instances (500–600 s) are identified based on their runtime on the CP solver. This restricted set of 30 instances was then submitted to the IP solver with 2 rounds. Easy instances took the IP solver around 5 h. Medium instances took from 5 to 10 h of solving time. Out of the 10 hard instances, only one terminated with the solution right before the 24 h limit. All others timed out. Then 19 easy instances of MIP (finished within 2 h) are submitted to CP solver. For those instances that are easy for MIP, CP is over an order of magnitude faster (finished within 10 s). The results above show that CP has an overwhelming advantage over MIP.

As for the memory consumption, the IP solver took 5 to 8 times more memory than CP with LDS for solving 1 round. For 2 rounds, that memory increase factor climbs to a range of 10x–16x.

6.2 Integrated

Figure 3 below illustrates the performance for the integrated approach with 1 and 10 rounds of AES encryption and side-channel information respectively. More than 80% of instances finish within 100 s, and the majority of the instances finish within 20 s. Increasing the rounds of AES encryption and side-channel information does not heavily affect the run time.

Figure 4 reports the runtime of LDS and DFS for 10 rounds. LDS is substantially faster than DFS and has a more stable performance throughout all 1000 instances. DFS timeouts after 10 min on 30% of the instances when considering multiple rounds whereas LDS experiences only 4% of timeouts.

Table 1. Performance for LDS, DFS and Staged approach

Rounds	LDS(Integrated)		DFS(Integrated)		Staged	
	μ_T	σ_T	μ_T	σ_T	μ_T	σ_T
1	41.25	85.66	44.40	89.29	73.5	116.36
2	24.78	63.97	54.68	115.33		
10	26.71	65.74	58.58	122.34		

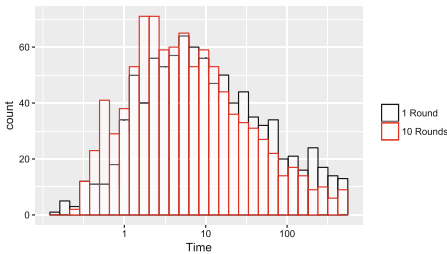


Fig. 3. LDS performance

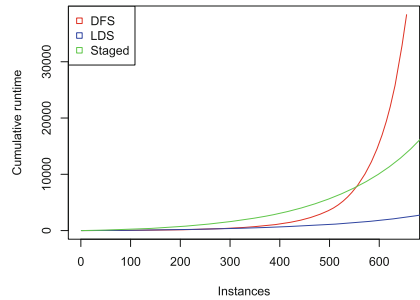


Fig. 4. Methods comparison

6.3 Staged

Since “staged” and “integrated” (10 rounds) are complete approaches, staged is compared to integrated on 10 rounds – with both LDS and DFS for integrated–. Table 1 shows that the staged approach is, on average, 3 times slower than LDS with a large deviation. Figure 4 shows that *LDS(Integrated)* has a much smaller “surface under the curve” when compared to *Staged*. While the staged approach displays reasonable running times, it is worth remembering that it requires the availability of the ciphertext as discussed in Sect. 5.2.

6.4 Solution Pool

The premise in [16] is that one is already able to effectively attack AES with only one round of relaxation since the pool of candidate solutions from the relaxation is reasonably small and therefore the candidates it offers can be checked exhaustively. That premise is based on the assumption that the pool of candidate solution associated to the global optimum f_1^* of $M(1)$ does indeed contain the global optimum. While this assumption happens to hold for the 1,000 randomly generated instances, it is not true in general. Nonetheless, it is informative to consider how the candidate pool size evolves when going from $M(1)$ to $M(10)$, i.e., as we consider increasingly tighter relaxations. As the empirical result shows, starting at round 2, the solution pool size for all instances collapses to 1. And the only one solution in the pool is indeed the correct key.

7 Conclusion

This paper extends [7] with the ability to conduct a Tolerant Algebraic Side-Channel Attack over the complete AES algorithm. Two approaches: *Staged* and *Integrated* are introduced. The staged approach obtains the optimum from TASCA-CP model to enumerate and check all the solutions against full-round AES structural constraints and the accompanying side-channel constraints and ciphertext. The integrated model aggregates each single round AES with COP and therefore is able to construct 1 to 10 rounds of AES model with side-channel information. To improve the performance of the customized search, Limited Discrepancy is adopted by the integrated approach. The empirical results show the integrated approach with LDS is faster and more consistent than the stock DFS strategy or the staged approach and are orders of magnitude more scalable in space and time than an IP.

References

1. Beierle, C., et al.: The skinny family of block ciphers and its low-latency variant mantis. Cryptology ePrint Archive, Report 2016/660 (2016). <https://eprint.iacr.org/2016/660>
2. Bogdanov, A., et al.: PRESENT: an ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74735-2_31
3. Gérard, D., Lafourcade, P., Minier, M., Solnon, C.: Revisiting AES related-key differential attacks with constraint programming. IACR Cryptology ePrint Archive 2017, 139 (2017). <http://eprint.iacr.org/2017/139>
4. Gerault, D., Minier, M., Solnon, C.: Constraint programming models for chosen key differential cryptanalysis. In: Rueher, M. (ed.) CP 2016. LNCS, vol. 9892, pp. 584–601. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44953-1_37
5. Gerault, D., Minier, M., Solnon, C.: Using constraint programming to solve a cryptanalytic problem. In: Sierra, C. (ed.) Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, 19–25 August 2017, pp. 4844–4848. ijcai.org (2017). <https://doi.org/10.24963/ijcai.2017/679>
6. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence, IJCAI 1995, vol. 1, pp. 607–613. Morgan Kaufmann Publishers Inc., San Francisco (1995). <http://dl.acm.org/citation.cfm?id=1625855.1625935>
7. Liu, F., Cruz, W., Ma, C., Johnson, G., Michel, L.: A tolerant algebraic side-channel attack on AES using CP. In: Beck, J.C. (ed.) CP 2017. LNCS, vol. 10416, pp. 189–205. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66158-2_13
8. Michel, L.D., Van Hentenryck, P.: Constraint satisfaction over bit-vectors. In: Milano, M. (ed.) CP 2012. LNCS, pp. 527–543. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33558-7_39
9. Michel, L., Van Hentenryck, P.: A microkernel architecture for constraint programming. Constraints **22**(2), 107–151 (2017). <https://doi.org/10.1007/s10601-016-9242-1>

10. Michel, L., See, A., Van Hentenryck, P.: Transparent parallelization of constraint programming. *INFORMS J. Comput.* **21**(3), 363–382 (2009). <https://doi.org/10.1287/ijoc.1080.0313>
11. Mister, S., Adams, C.: Practical S-box design. In: *Selected Areas in Cryptography* (1996)
12. Mohamed, M.S.E., Bulygin, S., Zohner, M., Heuser, A., Walter, M., Buchmann, J.: Improved algebraic side-channel attack on AES. *J. Cryptographic Eng.* **3**(3), 139–156 (2013). <https://doi.org/10.1007/s13389-013-0059-1>
13. NIST: Federal information processing standards publication (FIPS 197). *Advanced Encryption Standard (AES)* (2001)
14. Oren, Y., Kirschbaum, M., Popp, T., Wool, A.: Algebraic side-channel analysis in the presence of errors. In: Mangard, S., Standaert, F.-X. (eds.) *CHES 2010*. LNCS, vol. 6225, pp. 428–442. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15031-9_29
15. Oren, Y., Renaud, M., Standaert, F.-X., Wool, A.: Algebraic side-channel attacks beyond the hamming weight leakage model. In: Prouff, E., Schaumont, P. (eds.) *CHES 2012*. LNCS, vol. 7428, pp. 140–154. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33027-8_9
16. Oren, Y., Wool, A.: Tolerant algebraic side-channel analysis of AES. IACR Cryptology ePrint Archive, Report 2012/092 (2012). <http://iss.oy.ne.ro/TASCA-eprint>
17. Oren, Y., Wool, A.: Side-channel cryptographic attacks using pseudo-boolean optimization. *Constraints* **21**(4), 616–645 (2016). <https://doi.org/10.1007/s10601-015-9237-3>
18. Ramamoorthy, V., Silaghi, M.C., Matsui, T., Hirayama, K., Yokoo, M.: The design of cryptographic S-Boxes using CSPs. In: Lee, J. (ed.) *CP 2011*. LNCS, vol. 6876, pp. 54–68. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23786-7_7. <http://dl.acm.org/citation.cfm?id=2041160.2041169>
19. Renaud, M., Standaert, F.-X.: Algebraic side-channel attacks. In: Bao, F., Yung, M., Lin, D., Jing, J. (eds.) *Inscrypt 2009*. LNCS, vol. 6151, pp. 393–410. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16342-5_29
20. Renaud, M., Standaert, F.-X., Veyrat-Charvillon, N.: Algebraic side-channel attacks on the AES: why time also matters in DPA. In: Clavier, C., Gaj, K. (eds.) *CHES 2009*. LNCS, vol. 5747, pp. 97–111. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04138-9_8
21. Sun, S., et al.: Analysis of AES, skinny, and others with constraint programming. *IACR Trans. Symmetric Cryptol.* **2017**(1), 281–306 (2017). <https://doi.org/10.13154/tosc.v2017.i1.281-306>



Evaluating QBF Solvers: Quantifier Alternations Matter

Florian Lonsing^(✉) and Uwe Egly^(✉)

Research Division of Knowledge Based Systems,
Institute of Logic and Computation, TU Wien, Vienna, Austria
{florian.lonsing,uwe.egly}@tuwien.ac.at

Abstract. We present an experimental study of the effects of quantifier alternations on the evaluation of quantified Boolean formula (QBF) solvers. The number of quantifier alternations in a QBF in prenex conjunctive normal form (PCNF) is directly related to the theoretical hardness of the respective QBF satisfiability problem in the polynomial hierarchy. We show empirically that the performance of solvers based on different solving paradigms substantially varies depending on the numbers of alternations in PCNFs. In related theoretical work, quantifier alternations have become the focus of understanding the strengths and weaknesses of various QBF proof systems implemented in solvers. Our results motivate the development of methods to evaluate orthogonal solving paradigms by taking quantifier alternations into account. This is necessary to showcase the broad range of existing QBF solving paradigms for practical QBF applications. Moreover, we highlight the potential of combining different approaches and QBF proof systems in solvers.

1 Introduction

The logic of *quantified Boolean formulas (QBFs)* [33] extends propositional logic by existential and universal quantification of propositional variables. Consequently, the QBF satisfiability problem is PSPACE-complete [49]. QBF satisfiability is a restricted form of a *quantified constraint satisfaction problem (QCSP)*, cf. [13, 16, 17, 41], where all variables are defined over a Boolean domain.

The *polynomial hierarchy (PH)* [42, 48, 53] allows to describe the complexity of problems that are beyond the classes P and NP. The satisfiability problem of a QBF ψ in *prenex conjunctive normal form (PCNF)* with $k \geq 0$ quantifier alternations is located at level $k + 1$ of PH [48, 53] and either Σ_{k+1}^P -complete or Π_{k+1}^P -complete, depending on the first quantifier in ψ . Due to this property, practically relevant problems from any level of PH up to the class PSPACE (here with arbitrarily nested quantifiers) can succinctly be encoded as QBFs.

Efficient solvers are highly requested to solve QBF encodings of problems. Competitions like *QBFEVAL* or the *QBF Galleries* have been driving solver development [23, 29, 39]. State-of-the-art solvers are based on solving paradigms

Supported by the Austrian Science Fund (FWF) under grant S11409-N23.

like, e.g., expansion [2,10,30] or Q-resolution [34]. These two paradigms are *orthogonal* by proof complexity [7,31,50]. Informally, orthogonal paradigms have complementary strengths on certain families of formulas.

Motivated by the variety of available QBF solving paradigms and solvers, we present an *experimental study of the effects of quantifier alternations* on the evaluation of QBF solvers. To this end, we consider benchmarks, solvers, and preprocessors from QBFEVAL'17 [44]. As our main result, we show that the performance of solvers based on different and, notably, orthogonal solving paradigms substantially varies depending on the numbers of alternations. Instances with a particular number of alternations may be overrepresented (i.e., appear more frequently) in a benchmark set, thus resulting in *alternation bias*. In this case, overall solver rankings by total solved instances may not provide a comprehensive picture as they might only reflect the strengths of certain solvers on overrepresented instances, but not the (perhaps orthogonal) strengths of other solvers on underrepresented ones.

In related work [40], the correlation between solver performance and various syntactic features such as treewidth [1,45] was analyzed. In contrast to that, we do not study such correlations. By our study we *a posteriori* highlight diversity of solver performance based on the single feature of alternations, which are naturally related to the theoretical hardness of instances in PH. Recently, alternations have become of interest also in theoretical work on QBF proof complexity [6,9,18].

We aim at raising the awareness and importance of quantifier alternations in comparative studies of QBF solver performance and the potential negative impact on the progress of QBF solver development. If solvers are evaluated on benchmark sets with alternation bias and alternations are neglected in the analysis, then future research may inadvertently be narrowed down to only exploring approaches that perform well on overrepresented instances with a certain number of alternations. The risk of such detrimental effects on a research field driven by empirical analysis has been pointed already in the early days of propositional satisfiability (SAT) solving [27] and also with respect to more recent SAT solver competitions [3–5]. In contrast to the NP-completeness of SAT, the complexity landscape of QBF encodings defined by PH is more diverse, which gives rise to several sources of inadvertent convergence of research lines.

In addition to focusing on alternations, we report on *virtual best solver (VBS)* statistics, where the VBS solved between 50% and 70% more instances than the single overall best solver on a benchmark set. These results indicate the potential of combining orthogonal QBF proof systems in solvers. Moreover, we point out that overall low-ranked solvers potentially solve more instances uniquely and have larger contributions to the VBS than high-ranked ones. Similar observations were made in the context of SAT solver competitions [54].

The majority of benchmarks in QBFLIB [23], the QBF research community portal, has no more than two quantifier alternations. Hence problems from the first three levels in PH have been, and are, of primary interest to practitioners. However, to strengthen QBF solving as a key technology for solving problems

from *any* levels of PH up to PSPACE-complete problems, QBF solvers must be improved on instances with *any* number of alternations. Our empirical study motivates the development of methods to evaluate orthogonal solving paradigms by taking quantifier alternations into account. This is necessary to showcase the broad range of existing paradigms for practical QBF applications.

2 Preliminaries

We consider QBFs $\psi := \Pi.\phi$ in *prenex conjunctive normal form (PCNF)* consisting of a *quantifier prefix* $\Pi := Q_1B_1 \dots Q_nB_n$ and a quantifier-free propositional formula ϕ in *CNF*. A CNF consists of a conjunction of *clauses*. A clause is a disjunction of *literals*. A literal is either a propositional variable x or its negation $\neg x$. The prefix Π is a linearly ordered sequence of *quantifier blocks (qblocks)* Q_iB_i , where $Q_i \in \{\forall, \exists\}$ is a quantifier and B_i is a block (i.e., a set) of propositional variables with $B_i \cap B_j = \emptyset$ for $i \neq j$. The notation Q_iB_i is shorthand for $Q_ix_1 \dots Q_ix_m$ for all $x_j \in B_i$. Formula ϕ is defined precisely over the variables that appear in Π . If $Q_i = Q_{i+1}$ then B_i and B_{i+1} are merged to obtain $Q_i(B_i \cup B_{i+1})$. Hence adjacent qblocks are quantified differently. Without loss of generality, we assume that the innermost quantifier $Q_n = \exists$ is existential. (If $Q_n = \forall$ then B_n is eliminated by *universal reduction* [34]). A PCNF with n qblocks has $n - 1$ *quantifier alternations*.

The *semantics* of PCNFs are defined recursively. The PCNF consisting only of the *syntactic truth constant* \top (\perp) is satisfiable (unsatisfiable). A PCNF $\psi := Q_1B_1 \dots Q_nB_n.\phi$ with $Q_1 = \exists$ ($Q_1 = \forall$) is satisfiable iff, for $x \in B_1$, $\psi[x]$ or (and) $\psi[\neg x]$ is satisfiable, where $\psi[x]$ ($\psi[\neg x]$) is the PCNF obtained from ψ by replacing all occurrences of x by \top (\perp) and deleting x from B_1 .

To make the presentation of our experimental study self-contained, we introduce *QBF proof systems* only informally and refer to a standard, formal definition of propositional proof systems [20]. A *QBF proof system* \mathcal{PS} is a formal system consisting of *inference rules*. The inference rules allow to derive new formulas (e.g. clauses) from a given QBF ψ and from previously derived formulas. A QBF proof system \mathcal{PS} is *correct* if, for any QBF ψ , it holds that if the formula \perp (*false*, e.g., the empty clause) is derivable in \mathcal{PS} from ψ then ψ is unsatisfiable.¹ A QBF proof system \mathcal{PS} is *complete* if, for any QBF ψ , it holds that if ψ is unsatisfiable then \perp is derivable in \mathcal{PS} from ψ . A *proof* P of an unsatisfiable QBF ψ in \mathcal{PS} is a sequence of given formulas and formulas derived by inference rules ending in \perp . The *length* $|P|$ of a proof P is the sum of the sizes of all formulas in P .

Let \mathcal{PS} and \mathcal{PS}' be QBF proof systems and Ψ be a family of unsatisfiable QBFs. Let P be a proof of some QBF $\psi \in \Psi$ in \mathcal{PS} such that the length $|P|$ of P is polynomial in the size of ψ . Assume that the length $|P'|$ of every proof P' of $\psi \in \Psi$ in \mathcal{PS}' is exponential in the size of ψ . Then \mathcal{PS} is *stronger* than \mathcal{PS}' with respect to family Ψ . Two QBF proof systems \mathcal{PS} and \mathcal{PS}' are *orthogonal* if \mathcal{PS} is stronger than \mathcal{PS}' with respect to a family Ψ and \mathcal{PS}' is stronger than \mathcal{PS} with

¹ Theoretical work on QBF proof systems typically focuses on unsatisfiable QBFs.

respect to some other family Ψ' . The relation between QBF proof systems in terms of their strengths is studied in the research field of *QBF proof complexity*.

QBF proof systems are the formal foundation of QBF solver implementations. Expansion [2, 10, 30] and Q-resolution [34] are traditional QBF proof systems that are orthogonal [7, 31, 50]. Orthogonal proof systems are of particular interest for practical QBF solving since they give rise to solvers that have individual, complementary strengths on certain families of formulas. In our experiments, we highlight the potential of combining orthogonal proof systems in QBF solvers.

3 Experimental Setup

For our experimental study we use the set $S_{17|523}$ containing 523 PCNFs from QBFEVAL'17 [44]. Partitioning $S_{17|523}$ by numbers of qblocks results in 64 classes. Table 1 shows a histogram of $S_{17|523}$ by the numbers of formulas ($\#f$) in classes defined by the number of qblocks ($\#q$). Instances with up to three qblocks (row “1–3”) amount to 62% of all instances and hence are overrepresented in $S_{17|523}$. To generate $S_{17|523}$, instances were sampled from instance categories in QBFLIB in addition to newly submitted ones based on empirical hardness results from previous competitions. We also computed a histogram of a QBFLIB snapshot containing 16,748 instances (column $\#f_L$ in Table 1). Instances with no more than three qblocks (row “1–3”) are also overrepresented (69%) in that snapshot. Hence alternation bias in $S_{17|523}$ follows from a related bias in QBFLIB, which is due to the focus of QBF practitioners on problems located at low levels in PH. Moreover, the bias does *not* result from a flawed selection of competition instances. We use the terminology “overrepresented” and “bias” for the statistical fact that instances with few qblocks appear more frequently in $S_{17|523}$.

In order to evaluate the impact of qblocks on solver performance, we consider 11 solvers that participated in QBFEVAL'17 and were top-ranked.² The solvers implement the following six different *solving paradigms*:

1. *Expansion* [2, 10] eliminates variables from a PCNF ψ until the formula reduces to either *true* or *false*. RAReQS 1.1 [30] applies recursive expansion based on *counterexample-guided abstraction refinement (CEGAR)* [19], while Ijtihad operates in a non-recursive way. Rev-Qfun 0.1 [28] extends RAReQS by machine learning techniques, and DynQBF [15] exploits QBF tree decompositions. Theoretical properties of expansion as a proof system, which underlies implementations of expansion solvers, have been intensively studied [7, 31].

² For some solvers where version numbers are not reported, the authors kindly provided us with the competition versions, which were not publicly available. We excluded the solver AIGSolve because we observed assertion failures on certain instances.

Table 1. Histograms.

$\#q$	$\#f$	$\#f_L$
1	0	253
2	90	7,319
3	236	4,110
4–10	70	2,185
11–20	42	437
21–	85	2,444
1–3	326	11,682
4–	197	5,066

2. *QDPLL* [14] is a backtracking search procedure that generalizes the DPLL algorithm [21]. *GhostQ* [30, 35] combines QDPLL with clause and cube learning (a cube is a conjunction of literals) based on the *Q-resolution proof system* [34]. Additionally, it reconstructs the structure of PCNFs encoded by Tseitin translation [51], and applies CEGAR-based learning.
3. *Nested SAT solving* uses one SAT solver per qblock in a PCNF, where universal quantification is handled as negated existential quantification. The solver QSTS [11, 12] combines nested SAT solving with structure reconstruction. Propositional resolution is the proof system that underlies SAT solving.
4. *Clause selection* and *clausal abstraction* as implemented in the solvers QESTO 1.0 [32] and CAQE [46, 50], respectively, decompose the given PCNF into a sequence of propositional formulas and apply CEGAR techniques. The proof system implemented in CAQE has been presented recently [50].
5. *Backtracking search with clause and cube learning (QCDCL)* [24, 25, 36, 55] based on Q-resolution extends the CDCL approach for SAT solving [47] to QBFs. The solver DepQBF [37] implements QCDCL with generalized Q-resolution axioms allowing for a stronger calculus to derive learned clauses and cubes. Qute [43] learns variable dependencies lazily in a run.
6. Heretic is based on a *hybrid approach* that combines expansion and QCDCL in a sequential portfolio style. Thereby, the QCDCL solver DepQBF is applied to learn clauses from the given QBF, which are then heuristically added to the expansion solver Ijtihad.

4 Experimental Results

We illustrate a substantial performance diversity of the above solvers from QBFEVAL'17 on instances with different numbers of quantifier alternations. To this end, we rank solvers based on *instance classes* given by numbers of qblocks similar to Table 1. Our empirical results are consistent on instances with and without preprocessing by the state-of-the-art tools Bloqqer [26] and HQSpre [52]. Alternation bias in original instances is present also in preprocessed ones. Unless stated otherwise, all experiments were run on Intel Xeon CPUs (E5-2650v4, 2.20 GHz) with Ubuntu 16.04.1 using CPU time and memory limits of 1800 s and seven GB. Exceeding the memory limit is counted as a time out.

It is well known that preprocessing may have positive effects on the performance of certain solvers while negative effects on others (cf. [39, 40]). To compensate for these effects, we applied preprocessing both to filter the original benchmark set $S_{17|523}$ and to preprocess instances. Many preprocessing techniques used to simplify a QBF by eliminating clauses and literals are restricted variants of solving approaches, hence instances might be solved already by preprocessing.

We ran Bloqqer (version 37) with a time limit of two hours as a filter on set $S_{17|523}$ to obtain the set $S_{17|437}$ containing 437 *original* PCNFs, where we discarded 76 instances from $S_{17|523}$ that were solved already by Bloqqer and ten instances that became propositional, i.e., which ended up having a single

quantifier block of existential variables only. **Bloqer** exceeded the time limit on 39 instances, which we included in their original form in set $S_{17|437}$.

In a similar way, we filtered set $S_{17|523}$ using **HQSpre** to obtain the set $S_{17|312}$ containing 312 *original* instances, where we discarded 183 instances solved by **HQSpre** and 28 which became propositional, and we included 42 original ones in $S_{17|312}$ where **HQSpre** exceeded the resource limits. We did not consider a variant of **HQSpre** that applies a restricted form of preprocessing to preserve gate structure present in formulas [52]. Compared to the unrestricted variant of **HQSpre** we used, the restricted one did not improve overall solver performance.

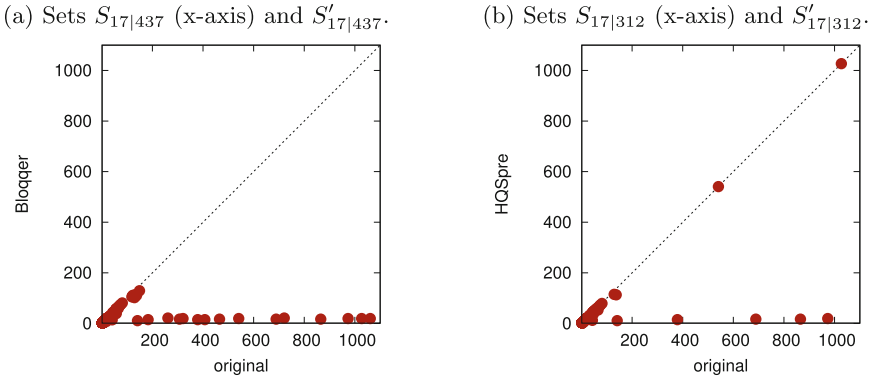


Fig. 1. Numbers of qblocks before (“original”) and after preprocessing by **Bloqer** (a) and **HQSpre** (b) on filtered (x-axes) and preprocessed instances (y-axes), respectively.

By applying **Bloqer** and **HQSpre** to the filtered sets $S_{17|437}$ and $S_{17|312}$ again, we generated the sets $S'_{17|437}$ and $S'_{17|312}$, respectively, containing preprocessed instances and those original instances where the preprocessors exceeded the resource limits. We disabled any additional use of **Bloqer** or **HQSpre** as separate preprocessing modules integrated in some solvers. *In the following, we focus our analysis on the four sets $S_{17|437}$, $S'_{17|437}$, $S_{17|312}$, and $S'_{17|312}$.*

The application of **Bloqer** and **HQSpre** to sets $S_{17|437}$ and $S_{17|312}$ reduces the number of qblocks in instances considerably. This is illustrated by the scatter plots in Figs. 1a and b, respectively. The average number of qblocks decreases from 29 in set $S_{17|437}$ to 10 in set $S'_{17|437}$. Likewise, the average decreases from 24 in set $S_{17|312}$ to 14 in set $S'_{17|312}$. As an extreme case, the number of qblocks in an instance in $S_{17|437}$ was reduced by **Bloqer** from 1061 to 19.

In all sets $S_{17|437}$, $S'_{17|437}$, $S_{17|312}$, and $S'_{17|312}$, the median number of qblocks is three. This is due to alternation bias like in the original set $S_{17|523}$ (Table 1). The related histograms are shown in Tables 2a to d, where instances with no more than three qblocks are overrepresented (rows “2–3”) as they amount to between 63% and 68% of all 437, respectively, 312 instances. Set $S_{17|437}$ has 59 classes by numbers of qblocks compared to 45 in set $S'_{17|437}$, and set $S_{17|312}$ has

Table 2. Histograms of the benchmark sets $S_{17|437}$ and $S'_{17|437}$ (filtered and preprocessed by **Bloqer**), and $S_{17|312}$ and $S'_{17|312}$ (filtered and preprocessed by **HQSpre**) illustrating the numbers of formulas ($\#f$) in classes given by the number of qblocks ($\#q$).

(a) Set $S_{17 437}$.		(b) Set $S'_{17 437}$.		(c) Set $S_{17 312}$.		(d) Set $S'_{17 312}$.	
$\#q$	$\#f$	$\#q$	$\#f$	$\#q$	$\#f$	$\#q$	$\#f$
2	63	2	65	2	70	2	70
3	215	3	218	3	145	3	145
4–10	63	4–10	59	4–10	26	4–10	26
11–20	36	11–20	53	11–20	30	11–20	40
21–	60	21–	42	21–	41	21–	31
2–3	278	2–3	283	2–3	215	2–3	215
4–	159	4–	154	4–	97	4–	97

42 compared to 40 in set $S'_{17|312}$. **Bloqer** reduces the number of instances with 21 or more qblocks (lines “21–”) from 60 in $S_{17|437}$ to 42 in $S'_{17|437}$ (Tables 2a and b). **HQSpre** reduces this number from 41 in $S_{17|312}$ to 31 in $S'_{17|312}$ (Tables 2c and d).

4.1 Solved Instances: Overall Rankings

We first analyze overall solver performance by ranking solvers according to total numbers of instances solved in the benchmark sets $S_{17|437}$, $S'_{17|437}$, $S_{17|312}$, and $S'_{17|312}$. Then we show that the strengths of certain solvers and solving paradigms are not reflected in such overall rankings. To highlight these individual strengths, in Sect. 4.2 below we carry out a more fine-grained analysis of solver performance based on instances that were solved in instance classes defined by their number of qblocks. Our results show that there is a considerable performance diversity between solvers and solving paradigms with respect to classes.

Tables 3a to d show overall solver rankings by total numbers of solved instances. Solver performance greatly varies depending on preprocessing. For example, while **RAReQS**, **CAQE**, and **QESTO** clearly benefit from preprocessing, it is harmful for **GhostQ** and **Rev-Qfun**. The expansion solvers **RAReQS** and **Rev-Qfun** (paradigm 1) dominate the rankings on sets $S_{17|437}$ and $S'_{17|437}$ (Tables 3a and b), and are ranked second on sets $S_{17|312}$ and $S'_{17|312}$ (Tables 3c and d). The first three places in the respective rankings of each set are taken by solvers based on paradigms 1, 2, 4, and 6. That is, solvers **QSTS**, **DepQBF**, and **Qute** (paradigms 3 and 5) are not among the three top-performing solvers.

There is a large performance diversity between different solvers based on the same paradigm. For example, the expansion solver **DynQBF** is ranked last on three sets, which is in contrast to the overall good performance of the expansion solvers **RAReQS** and **Rev-Qfun**. Likewise, there is a difference between the **QCDC**L solvers **DepQBF** and **Qute**. Such differences between implementations

Table 3. Solvers and corresponding paradigms (P) from Sect. 3, solved instances (S), unsatisfiable (\perp) and satisfiable ones (\top), total CPU time including time outs, and uniquely solved instances (U) on sets $S_{17|437}$ (a), $S'_{17|437}$ (b), $S_{17|312}$ (c), and $S'_{17|312}$ (d).

(a) Set $S_{17 437}$ filtered by Bloqqr.							(b) Set $S'_{17 437}$ preprocessed by Bloqqr.						
<i>Solver</i>	P	S	\perp	\top	<i>Time</i>	U	<i>Solver</i>	P	S	\perp	\top	<i>Time</i>	U
Rev-Qfun	1	174	106	68	497K	6	RAReQS	1	175	127	48	499K	5
GhostQ	2	145	79	66	547K	12	CAQE	4	169	114	55	514K	0
RAReQS	1	126	94	32	577K	4	Heretic	6	164	119	45	513K	0
CAQE	4	126	87	39	578K	6	Ijtihad	1	136	103	33	555K	2
Heretic	6	122	95	27	580K	0	Rev-Qfun	1	135	92	43	563K	3
DepQBF	5	115	78	37	603K	16	QSTS	3	127	98	29	576K	12
Ijtihad	1	110	88	22	599K	1	QESTO	4	115	84	31	601K	1
QSTS	3	103	75	28	618K	3	DepQBF	5	102	64	38	624K	3
Qute	5	77	47	30	658K	0	GhostQ	2	82	47	35	661K	1
QESTO	4	76	56	20	661K	0	Qute	5	73	56	17	672K	0
DynQBF	1	47	27	20	714K	9	DynQBF	1	65	37	28	684K	25

(c) Set $S_{17 312}$ filtered by HQSpre.							(d) Set $S'_{17 312}$ preprocessed by HQSpre.						
<i>Solver</i>	P	S	\perp	\top	<i>Time</i>	U	<i>Solver</i>	P	S	\perp	\top	<i>Time</i>	U
GhostQ	2	112	61	51	373K	15	CAQE	4	114	65	49	378K	6
Rev-Qfun	1	110	58	52	376K	6	RAReQS	1	103	63	40	390K	3
CAQE	4	68	42	26	454K	6	QESTO	4	97	63	34	402K	1
DepQBF	5	64	41	23	461K	4	Rev-Qfun	1	90	57	33	414K	6
QSTS	3	56	34	22	470K	3	Heretic	6	87	55	32	424K	0
RAReQS	1	50	34	16	482K	1	QSTS	3	72	46	26	448K	1
Heretic	6	49	34	15	485K	0	DepQBF	5	72	44	28	451K	5
Qute	5	47	25	22	486K	0	Qute	5	70	42	28	449K	2
DynQBF	1	46	24	22	488K	9	Ijtihad	1	58	43	15	465K	1
QESTO	4	45	30	15	491K	0	GhostQ	2	58	33	25	475K	0
Ijtihad	1	36	27	9	504K	1	DynQBF	1	45	24	21	487K	17

of the same solving paradigm (or proof system) can be attributed to the fact that the solvers might apply different heuristics to explore the search space to find a proof.

The numbers of instances solved uniquely by a particular solver (columns U in Tables 3a to d) highlight the strengths of solvers such as QSTS, DynQBF, and DepQBF which do not show top performance in the overall rankings. Most notably DynQBF by far solved the largest number of instances uniquely on pre-processed sets $S'_{17|437}$ (Table 3b) and $S'_{17|312}$ (Table 3d). With respect to uniquely solved instances, QSTS is second after DynQBF on set $S'_{17|437}$, and DepQBF solved the largest number of instances uniquely on set $S_{17|437}$ (Table 3a).

Towards a more fine-grained analysis of solver performance, we consider the number of qblocks of instances solved by individual solvers and in total by solving paradigms. Table 4 shows related average and median numbers of qblocks. In

Table 4. Solvers and corresponding solving paradigms (P) as listed in Sect. 3, solved instances (S , cf. Tables 3a to d), average (\bar{q}) and median number (\tilde{q}) of qblocks of respective solved instances in the considered benchmark sets. Rows “ \cup ” show statistics for the total number of instances solved by any solver based on a particular paradigm.

P	$Solver$	$S_{17 437}$			$S'_{17 437}$			$S_{17 312}$			$S'_{17 312}$		
		S	\bar{q}	\tilde{q}	S	\bar{q}	\tilde{q}	S	\bar{q}	\tilde{q}	S	\bar{q}	\tilde{q}
1	DynQBF	47	6.1	3.0	65	9.0	3.0	46	4.8	3.0	45	3.3	2.0
	Ijtihad	110	42.1	5.0	136	12.7	3.0	36	40.5	3.0	58	17.6	3.0
	RAReQS	126	39.8	3.0	175	11.2	3.0	50	22.6	3.0	103	11.5	3.0
	Rev-Qfun	174	55.1	3.0	135	12.5	3.0	110	47.4	3.0	90	24.0	3.0
	\cup	228	45.9	3.0	238	9.6	3.0	145	37.8	3.0	150	16.6	3.0
2	GhostQ	145	12.5	3.0	82	15.8	3.0	112	7.5	3.0	58	8.1	3.0
3	QSTS	103	63.2	5.0	127	15.6	5.0	56	65.3	3.0	72	22.6	3.0
4	CAQE	126	44.3	5.0	169	12.9	3.0	68	37.4	3.0	114	12.0	3.0
	QESTO	76	47.7	3.0	115	15.5	3.0	45	15.6	3.0	97	8.1	3.0
	\cup	134	41.9	3.5	182	12.5	3.0	74	34.7	3.0	127	11.6	3.0
5	DepQBF	115	45.7	5.0	102	17.8	8.5	64	21.2	8.0	72	10.5	3.0
	Qute	77	30.0	4.0	73	20.7	9.0	47	16.4	3.0	70	9.7	3.0
	\cup	137	38.8	3.0	117	16.2	6.0	83	17.0	3.0	97	9.2	3.0
6	Heretic	122	39.5	5.0	164	12.5	5.0	49	34.4	3.0	87	14.1	3.0

general, averages are greater for instances from filtered sets ($S_{17|437}$ and $S_{17|312}$) than from preprocessed ones ($S'_{17|437}$ and $S'_{17|312}$), since preprocessing reduces the numbers of qblocks (cf. Fig. 1). The difference in averages between solvers based on the same paradigm, e.g., DynQBF and Rev-Qfun in set $S_{17|437}$, is due to few solved instances having many qblocks (up to more than 1000).

Although the median number of qblocks of instances in *every* considered set is three (due to alternation bias), the median number of instances solved by certain solvers as shown in Table 4 is greater than three. For example, this is the case for the QCDCL solvers DepQBF and Qute on sets $S_{17|437}$, $S'_{17|437}$, and $S_{17|312}$ (DepQBF only). Moreover, QCDCL is the solving paradigm with the greatest median (6.0 in set $S'_{17|437}$) among all sets when considering instances solved by any solver based on a particular paradigm (rows “ \cup ”). Ijtihad has the greatest median among expansion solvers, QSTS and Heretic have a median of 5.0 on sets $S_{17|437}$ and $S'_{17|437}$, and CAQE has a median of 5.0 on set $S_{17|437}$. These statistics indicate that there are solvers which tend to perform well on instances with relatively many qblocks, which however is not reflected in overall rankings in Tables 3a to d as many of these solvers are *not* among the top-performing ones.

4.2 Solved Instances: Class-Based Analysis

Motivated by the above observations related to median numbers of qblocks of solved instances, we aim to provide a more detailed picture of the strengths of the different solvers and implemented solving paradigms. To this end, we analyze the *numbers of solved instances in classes defined by their numbers of qblocks*.

Tables 5a to d show the numbers of instances that were solved in the individual classes in the considered sets. Only class winners are shown (bold face),³ i.e., solvers that solved the largest number of instances in at least one class, where ties are not broken. The bottom rows of the tables show statistics for instances with up to three (row “2–3”) and more than three qblocks (row “4–”).

The *five different class winners* Rev-Qfun, GhostQ, CAQE, Heretic, and DepQBF in set $S_{17|437}$ (Table 5a) implement *five different solving paradigms*

Table 5. Instances solved in classes by numbers of qblocks (#q) and numbers of formulas in each class (#f) for sets $S_{17|437}$ (a), $S'_{17|437}$ (b), $S_{17|312}$ (c), $S'_{17|312}$ (d). Only class winners (bold face) are shown, paradigms (P:) are indicated in the first row.

(a) Set $S_{17 437}$ filtered by Bloqger.							(b) Set $S'_{17 437}$ preprocessed by Bloqger.					
P:		1	2	4	6	5	P:		1	4	6	1
#q	#f	Rev-Qfun	GhostQ	CAQE	Heretic	DepQBF	#q	#f	RAReQS	CAQE	Heretic	DynQBF
2	63	17	32	5	2	6	2	65	16	15	13	24
3	215	101	89	56	50	47	3	218	80	81	65	18
4–10	63	25	4	25	34	14	4–10	59	37	26	38	13
11–20	36	6	3	10	11	20	11–20	53	25	25	31	4
21–	60	25	17	30	25	28	21–	42	17	22	17	6
2–3	278	118	121	61	52	53	2–3	283	96	96	78	42
4–	159	56	24	65	70	62	4–	154	79	73	86	23

(c) Set $S_{17 312}$ filtered by HQSpre.						(d) Set $S'_{17 312}$ preprocessed by HQSpre.					
P:		2	1	4	5	P:		4	6	5	1
#q	#f	GhostQ	Rev-Qfun	CAQE	DepQBF	#q	#f	CAQE	Heretic	DepQBF	DynQBF
2	70	36	18	5	7	2	70	18	15	15	24
3	145	62	71	33	23	3	145	67	42	24	14
4–10	26	3	5	7	7	4–10	26	6	10	7	5
11–20	30	3	5	8	16	11–20	40	14	15	20	2
21–	41	8	11	15	11	21–	31	9	5	6	0
2–3	215	98	89	38	30	2–3	215	85	57	39	38
4–	97	14	21	30	34	4–	97	29	30	33	7

³ We refer to an online appendix for complete tables [38].

(rows P :). In set $S'_{17|437}$ (Table 5b) the four class winners implement three different paradigms. In sets $S_{17|312}$ and $S'_{17|312}$ (Tables 5c and d), there are four different paradigms implemented in the respective four class winners. Overall, with respect to all four benchmark sets, there are seven different solvers out of the 11 considered ones that win in a class. These class winners implement five out of the six paradigms listed in Sect. 3, all except paradigm 3 implemented in QSTS.

Notably, class winners are not always overall top-ranked, and an overall top-ranked solver does not always win a class. For example, RAReQS is ranked third in set $S_{17|437}$ (Table 3a) and second in set $S'_{17|312}$ (Table 3d) but does not win a class in the respective set (Tables 5a and d). As an extreme case, DynQBF is ranked last on sets $S'_{17|437}$ and $S'_{17|312}$ (Tables 3b and d) but wins the class of instances with no more than two qblocks (row “2” in Tables 5b and d).

Instances with few qblocks are overrepresented in the benchmark sets. Alternation bias of this kind in general bears the risk of masking the strengths of certain solvers on underrepresented instances. The variety of class winners and paradigms shown in Tables 5a to d is not reflected when only considering overall solver rankings by total numbers of solved instances in Tables 3a to d.

The expansion solvers Rev-Qfun and RAReQS (paradigm 1) tend to perform better on instances with relatively few qblocks, while solvers applying QCDCL (paradigms 5 and 6) tend to perform better on many qblocks. For example, either DepQBF or Heretic win on instances with four or more qblocks (row “4-”) in any set. These statistics are interesting in the context of QBF proof complexity as the proof systems underlying expansion and QCDCL are orthogonal [7, 31]. CAQE based on paradigm 4 wins on instances with 21 or more qblocks (rows “21-”) in all sets (Tables 5a to d). Further, it also wins on instances with no more than three qblocks in set $S'_{17|312}$ (Table 5d). The proof systems underlying paradigms 4 and 1 (expansion) are orthogonal [50]. The performance diversity of orthogonal proof systems on instances with different numbers of qblocks is not reflected in overall rankings and motivates further, theoretical study in QBF proof complexity.

Due to alternation bias, classes of instances with few qblocks are larger than those with many qblocks. Hence solvers often win in a class of instances with many qblocks by only a small margin. For example, the top-ranked solvers on classes “4–10”, “11–20”, and “21–” tend to be close to each other in terms of solved instances (cf. appendix [38]). Moreover, solvers implementing the same paradigm might show diverse performance due to different heuristics in proof search. To consider these factors, we carry out a *class-based analysis of solving paradigms*. To this end, we count instances solved by any solver implementing a particular paradigm. This study is related to statistics in rows “U” of Table 4.

Tables 6a to d show instances solved by each of the solving paradigms 1 to 6 (first row) in classes of instances. Class winners are highlighted in bold face. Paradigm 1 (expansion) dominates the other paradigms on complete benchmark sets (row “2-”). On instances obtained by Bloqqer (Tables 6a and b), in total only four classes are won by paradigms other than expansion: class “2” by paradigm 2

Table 6. Instances solved by solving paradigms 1 to 6 (cf. Sect. 3) in classes by numbers of qblocks (#q) for sets $S_{17|437}$ (a), $S'_{17|437}$ (b), $S_{17|312}$ (c), and $S'_{17|312}$ (d).

(a) Set $S_{17 437}$ filtered by Bloqqer.							(b) Set $S'_{17 437}$ preprocessed by Bloqqer.						
#q	1	2	3	4	5	6	#q	1	2	3	4	5	6
2	26	32	8	6	7	2	2	37	3	11	17	10	13
3	121	89	43	61	66	50	3	103	53	46	86	40	65
4–10	38	4	21	27	16	34	4–10	49	5	25	28	18	38
11–20	10	3	8	10	20	11	11–20	31	9	24	27	32	31
21–	33	17	23	30	28	25	21–	18	12	21	24	17	17
2–3	147	121	51	67	73	52	2–3	140	56	57	103	50	78
4–	81	24	52	67	64	70	4–	98	26	70	79	67	86
2–	228	145	103	134	137	122	2–	238	82	127	182	117	164

(c) Set $S_{17 312}$ filtered by HQSpre.							(d) Set $S'_{17 312}$ preprocessed by HQSpre.						
#q	1	2	3	4	5	6	#q	1	2	3	4	5	6
2	28	36	9	6	8	2	2	37	7	17	18	21	15
3	85	62	27	36	40	23	3	78	40	35	71	40	42
4–10	9	3	1	9	8	5	4–10	10	1	2	13	7	10
11–20	8	3	7	8	16	9	11–20	17	6	13	15	21	15
21–	15	8	12	15	11	10	21–	8	4	5	10	8	5
2–3	113	98	36	42	48	25	2–3	115	47	52	89	61	57
4–	32	14	20	32	35	24	4–	35	11	20	38	36	30
2–	145	112	56	74	83	49	2–	150	58	72	127	97	87

(QDPLL) on set $S_{17|437}$, class “11–20” by paradigm 5 (QCDCL) on sets $S_{17|437}$ and $S'_{17|437}$, and class “21–” by paradigm 4 (clause selection/abstraction) on set $S'_{17|437}$. Regarding the dominance of paradigm 1 (expansion) in Tables 6a and b, we note that four solvers among the considered ones are based on expansion, while there are at most two solvers implementing the other paradigms.

Performance is more diverse on instances filtered and preprocessed by HQSpre (Tables 6c and d). There, paradigms other than expansion either win or are on par with expansion in nine classes in total. Notably, paradigms 4 and 5 win in classes “4–” of sets $S'_{17|312}$ and $S_{17|312}$ containing instances with many qblocks. Although CAQE (paradigm 4) is overall top-ranked on set $S'_{17|312}$ (Table 3d), the strong performance of paradigms 4 and 5 on instances with many qblocks is not reflected in overall rankings (Tables 3c and d).

4.3 Virtual Best Solver Analysis

We strengthen our above observations of performance diversity of solvers and solving paradigms with respect to numbers of qblocks by a *virtual best solver* (VBS) analysis, which is common in QBF [40] and SAT competitions (cf. [4]). The VBS is an ideal portfolio where the solving time of the fastest solver on an instance is attributed to the VBS. Thus the VBS reflects the best performance that can be achieved when running a set of solvers in parallel on an instance.

Table 7. Instances solved by the virtual best solver (VBS) in classes by number of qblocks (#q), number of formulas (#f) in each class, and relative contribution (%) of each solver to instances solved by the VBS for sets $S_{17|312}$ (a) and $S'_{17|312}$ (b).

(a) Set $S_{17|312}$ filtered by HQSpre.

#q	#f	VBS	GhostQ	Rev-Qfun	CAQE	DepQBF	QSTS	RAREQS	Heretic	Quite	DynQBF	QESTO	Ijtihad
2	70	46	41.3	6.5	6.5	6.5	6.5	0.0	0.0	0.0	30.4	2.1	0.0
3	145	89	12.3	33.7	2.2	2.2	15.7	22.4	0.0	3.3	2.2	4.4	1.1
4-10	26	19	5.2	0.0	26.3	26.3	0.0	0.0	0.0	15.7	10.5	10.5	5.2
11-20	30	18	0.0	0.0	11.1	50.0	27.7	5.5	0.0	0.0	5.5	0.0	0.0
21-	41	21	4.7	14.2	19.0	9.5	28.5	14.2	0.0	0.0	9.5	0.0	0.0
2-3	215	135	22.2	24.4	3.7	3.7	12.5	14.8	0.0	2.2	11.8	3.7	0.7
4-	97	58	3.4	5.1	18.9	27.5	18.9	6.8	0.0	5.1	8.6	3.4	1.7
2-	312	193	16.5	18.6	8.2	10.8	14.5	12.4	0.0	3.1	10.8	3.6	1.0

(b) Set $S'_{17|312}$ preprocessed by HQSpre.

#q	#f	VBS	CAQE	RAREQS	QESTO	Rev-Qfun	Heretic	QSTS	DepQBF	Quite	Ijtihad	GhostQ	DynQBF
2	70	40	7.5	17.5	2.5	7.5	2.5	10.0	10.0	0.0	0.0	2.5	40.0
3	145	87	9.1	40.2	8.0	12.6	1.1	6.8	0.0	8.0	3.4	4.5	5.7
4-10	26	20	25.0	10.0	15.0	5.0	0.0	0.0	25.0	5.0	5.0	0.0	10.0
11-20	40	26	3.8	19.2	7.6	0.0	7.6	26.9	30.7	0.0	0.0	0.0	3.8
21-	31	11	9.0	27.2	9.0	9.0	0.0	27.2	9.0	9.0	0.0	0.0	0.0
2-3	215	127	8.6	33.0	6.2	11.0	1.5	7.8	3.1	5.5	2.3	3.9	16.5
4-	97	57	12.2	17.5	10.5	3.5	3.5	17.5	24.5	3.5	1.7	0.0	5.2
2-	312	184	9.7	28.2	7.6	8.6	2.1	10.8	9.7	4.8	2.1	2.7	13.0

Tables 7a and b show numbers of instances solved by the VBS in classes for sets $S_{17|312}$ and $S'_{17|312}$ and the relative contribution of solvers (percentage) to the VBS in terms of solved instances. Similar to instances solved in classes (Tables 5a to d), the VBS contributions differ and provide a more fine-grained picture of the strengths of solvers and solving paradigms than the VBS contributions on the entire benchmark set (rows “2-” in Tables 7a and b). In the following, we comment on general VBS statistics for all considered benchmark sets, with a focus on sets $S_{17|312}$ and $S'_{17|312}$ generated using HQSpre. We refer to the appendix [38] for tables related to sets $S_{17|437}$ and $S'_{17|437}$ generated using Bloqper.

On all benchmark sets the VBS solved between 50% and 70% more instances than the single overall best solver (Tables 3a to d). These results highlight the complementary strengths of solvers and solving paradigms that are not among the top-ranked ones. On each of the four benchmark sets, there are five different solvers, respectively, which have the largest VBS contribution in a class. Interestingly, from the respective overall winning solvers (Tables 3a to d), only RAREQS on set $S'_{17|437}$ also has the largest VBS contribution on the entire benchmark set. While RAREQS is ranked second on set $S'_{17|312}$ (Table 3d), it has the largest overall VBS contribution (row “2-” in Table 7b).

Consistent with Tables 5b and d, where DynQBF solved the largest number of instances in class “2” of sets $S'_{17|437}$ and $S'_{17|312}$, it has the largest VBS contributions in this class (cf. Table 7b and appendix [38]) although it is ranked last in overall rankings (Tables 3b and d). The large VBS contributions of DynQBF conform to the fact that it solved the largest numbers of instances uniquely in sets $S'_{17|437}$ and $S'_{17|312}$. Similar observations regarding VBS contributions of solvers that are not top-ranked were made in the context of SAT solver competitions [54].

QSTS neither is among the overall top-ranked solvers (Tables 3a to d) nor among the class winners (Tables 5a to d), yet it has the largest VBS contribution in class “21-” on all sets except $S'_{17|312}$ (Table 7b), where it is on par with RReQS.

Similar to the analysis presented in Tables 6a to d, we analyze the *VBS contribution of each solving paradigm* for sets $S_{17|312}$ and $S'_{17|312}$ in Tables 8a and b, respectively. We refer to the appendix [38] for tables related to sets $S_{17|437}$ and $S'_{17|437}$. Considering instances with many qblocks (row “4-”), paradigm 5 (QCDCL) has the largest contribution in set $S_{17|312}$ and is on par with paradigm 1 (expansion) in set $S'_{17|312}$. This is remarkable, given that paradigm 1, where four solvers are based on, clearly has the largest VBS contribution on the entire sets (rows “2-”). However, only two solvers implement paradigm 5.

4.4 Discussion

In the following, we discuss threats to the validity of our study and related issues.

Heuristics. The performance of solvers implementing the same paradigm might be diverse due to different heuristics applied in proof search. To comprehensively evaluate the impact of heuristics, it is necessary to consider further syntactic parameters of instances other than alternations, such as ratio of variables per clause, size of clauses, and the like. In our study, we focused on alternations as they impact the theoretical hardness of PCNFs, thus resulting in a larger complexity landscape than, e.g., in propositional logic (SAT). To even out the effects of heuristics, we studied and observed performance diversity of *paradigms* (Tables 6 and 8). Such diversity cannot be explained by different heuristics, in contrast to diversity between individual solvers based on the same paradigm.

Dominance of Single Solvers and Paradigms. We are not aware of solvers being specifically targeted to instances with a particular number of alternations. Similar to the effects of heuristics, we even out a potential dominance of single solvers and overrepresented paradigms in solvers by a paradigm-based analysis (Tables 6 and 8). This provides a more comprehensive picture of the strengths of different paradigms. This way, e.g., we observed remarkable results regarding the VBS contribution of QCDCL on instances with many alternations (Table 8).

Choice of Benchmarks and Solvers. The benchmarks we considered contain few instances with many alternations, which follows from alternation bias in original benchmarks (cf. Sect. 3). We observed performance diversity in the large

Table 8. Instances solved by the virtual best solver (VBS) in classes by number of qblocks (#q), number of formulas (#f) in each class, and relative contribution (%) of solving paradigms to instances solved by the VBS for sets $S_{17|312}$ (a), and $S'_{17|312}$ (b).

(a) Set $S_{17|312}$ filtered by HQSpre.

#q	#f	VBS	1	2	3	4	5	6
2	70	46	36.9	41.3	6.5	8.6	6.5	0.0
3	145	89	59.5	12.3	15.7	6.7	5.6	0.0
4–10	26	19	15.7	5.2	0.0	36.8	42.1	0.0
11–20	30	18	11.1	0.0	27.7	11.1	50.0	0.0
21–	41	21	38.0	4.7	28.5	19.0	9.5	0.0
2–3	215	135	51.8	22.2	12.5	7.4	5.9	0.0
4–	97	58	22.4	3.4	18.9	22.4	32.7	0.0
2–	312	193	43.0	16.5	14.5	11.9	13.9	0.0

(b) Set $S'_{17|312}$ preprocessed by HQSpre.

#q	#f	VBS	1	2	3	4	5	6
2	70	40	65.0	2.5	10.0	10.0	10.0	2.5
3	145	87	62.0	4.5	6.8	17.2	8.0	1.1
4–10	26	20	30.0	0.0	0.0	40.0	30.0	0.0
11–20	40	26	23.0	0.0	26.9	11.5	30.7	7.6
21–	31	11	36.3	0.0	27.2	18.1	18.1	0.0
2–3	215	127	62.9	3.9	7.8	14.9	8.6	1.5
4–	97	57	28.0	0.0	17.5	22.8	28.0	3.5
2–	312	184	52.1	2.7	10.8	17.3	14.6	2.1

classes “2–3” and “4–”, which is more robust than in smaller classes containing fewer instances. Class “4–” is the largest one with many alternations that can be selected in the given benchmarks. Our choice of solvers was predetermined by the ranking of the top-performing solvers in the PCNF track of QBFEVAL’17.

Relation to QBF Proof Complexity. We emphasize that our study does *not* show that certain proof systems *provably* perform differently with respect to alternations. This is an open research problem in QBF proof complexity.

Overrepresented Problems and Different Prenex Forms. Several QBF encodings of a problem with different numbers of alternations may exist. Hence in the instance classes we defined by alternations certain problems might be overrepresented. These problems may be detected based on detailed information about the encoding process. However, such information is often not available for PCNF benchmarks. A related issue is the impact of different quantifier prefixes in PCNFs on solver performance, which was studied in theory [8] and practice [22].

5 Conclusion

We analyzed the effects of quantifier alternations on the evaluation of QBF solvers. Our empirical results indicate that the performance of solvers based on

different solving paradigms substantially varies on classes of formulas defined by their numbers of alternations. While the *theoretical hardness* of QBFs in prenex CNF with a particular number of alternations is naturally related to levels in the polynomial hierarchy, our study *a posteriori* sheds light on solver performance *observed in practice*. We observed a substantial performance diversity of solvers based on orthogonal QBF proof systems [7, 31, 50] on instances with different numbers of alternations, e.g., expansion and Q-resolution. Thereby, our work is in line with a recent focus on alternations in QBF proof complexity [6, 9, 18]. As a future direction in practice, and motivated by virtual best solver statistics we presented, it is promising to combine orthogonal approaches to leverage their individual strengths in a single QBF solver.

The class- and paradigm-based performance analysis we presented is a methodology to evaluate QBF solvers that takes quantifier alternations of under- and overrepresented instances into account. This is necessary to highlight the strengths of solving paradigms in a comprehensive way. In doing so, we aim to reach out to users of QBF technology who are inexperienced with solver implementations and look for solvers that are suitable to solve a particular problem. Ultimately, QBF technology must be improved as a general approach to tackle PSPACE problems.

References

1. Atserias, A., Oliva, S.: Bounded-width QBF is PSPACE-complete. In: STACS. LIPIcs, vol. 20, pp. 44–54. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013)
2. Ayari, A., Basin, D.: QUBOS: deciding quantified boolean logic using propositional satisfiability solvers. In: Aagaard, M.D., O’Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 187–201. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36126-X_12
3. Balint, A., Belov, A., Järvisalo, M., Sinz, C.: Overview and analysis of the SAT challenge 2012 solver competition. *Artif. Intell.* **223**, 120–155 (2015)
4. Balyo, T., Biere, A., Iser, M., Sinz, C.: SAT race 2015. *Artif. Intell.* **241**, 45–65 (2016). <https://doi.org/10.1016/j.artint.2016.08.007>
5. Balyo, T., Heule, M.J.H., Järvisalo, M.: SAT competition 2016: recent developments. In: AAAI, pp. 5061–5063. AAAI Press (2017)
6. Beyersdorff, O., Blinkhorn, J., Hinde, L.: Size, cost and capacity: a semantic technique for hard random QBFs. In: ITCS. LIPIcs, vol. 94, pp. 9:1–9:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)
7. Beyersdorff, O., Chew, L., Janota, M.: Proof complexity of resolution-based QBF calculi. In: STACS. LIPIcs, vol. 30, pp. 76–89. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)
8. Beyersdorff, O., Chew, L., Janota, M.: Extension variables in QBF Resolution. In: Beyond NP Workshop, AAAI Workshops, vol. WS-16-05. AAAI Press (2016)
9. Beyersdorff, O., Hinde, L., Pich, J.: Reasons for hardness in QBF proof systems. In: FSTTCS. LIPIcs, vol. 93, pp. 14:1–14:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)

10. Biere, A.: Resolve and expand. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 59–70. Springer, Heidelberg (2005). https://doi.org/10.1007/11527695_5
11. Bogaerts, B., Janhunnen, T., Tasharrofi, S.: SAT-to-SAT in QBF Eval 2016. In: QBF Workshop, CEUR Workshop Proceedings, vol. 1719, pp. 63–70. CEUR-WS.org (2016)
12. Bogaerts, B., Janhunnen, T., Tasharrofi, S.: Solving QBF instances with nested SAT solvers. In: Beyond NP Workshop 2016 at AAAI 2016 (2016)
13. Bordeaux, L., Cadoli, M., Mancini, T.: CSP properties for quantified constraints: definitions and complexity. In: AAAI, pp. 360–365. AAAI Press/The MIT Press (2005)
14. Cadoli, M., Giovanardi, A., Schaerf, M.: An algorithm to evaluate quantified Boolean formulae. In: AAAI, pp. 262–267. AAAI Press/The MIT Press (1998)
15. Charwat, G., Woltran, S.: Expansion-based QBF solving on tree decompositions. In: RCRA Workshop, CEUR Workshop Proceedings, vol. 2011, pp. 16–26. CEUR-WS.org (2017)
16. Chen, H.: A rendezvous of logic, complexity, and algebra. *ACM Comput. Surv.* **42**(1), 2:1–2:32 (2009)
17. Chen, H.: Meditations on quantified constraint satisfaction. In: Constable, R.L., Silva, A. (eds.) Logic and Program Semantics. LNCS, vol. 7230, pp. 35–49. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29485-3_4
18. Chen, H.: Proof complexity modulo the polynomial hierarchy: understanding alternation as a source of hardness. *TOCT* **9**(3), 15:1–15:20 (2017)
19. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* **50**(5), 752–794 (2003)
20. Cook, S.A., Reckhow, R.A.: The relative efficiency of propositional proof systems. *J. Symb. Log.* **44**(1), 36–50 (1979)
21. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. *Commun. ACM* **5**(7), 394–397 (1962)
22. Egly, U., Seidl, M., Tompits, H., Woltran, S., Zolda, M.: Comparing different prenexing strategies for quantified Boolean formulas. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 214–228. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_17
23. Giunchiglia, E., Narizzano, M., Pulina, L., Tacchella, A.: Quantified Boolean Formulas Library (QBFLIB) and Solver Evaluation Portal (QBFEVAL) (2004). www.qbflib.org
24. Giunchiglia, E., Narizzano, M., Tacchella, A.: Learning for quantified boolean logic satisfiability. In: AAAI, pp. 649–654. AAAI Press/The MIT Press (2002)
25. Giunchiglia, E., Narizzano, M., Tacchella, A.: Clause/term resolution and learning in the evaluation of quantified Boolean formulas. *JAIR* **26**, 371–416 (2006)
26. Heule, M., Järvisalo, M., Lonsing, F., Seidl, M., Biere, A.: Clause elimination for SAT and QSAT. *JAIR* **53**, 127–168 (2015)
27. Hooker, J.N.: Testing heuristics: we have it all wrong. *J. Heuristics* **1**(1), 33–42 (1995)
28. Janota, M.: Towards generalization in QBF solving via machine learning. In: Proceedings of the AAAI 2018 (2018, to appear)
29. Janota, M., Jordan, C., Klieber, W., Lonsing, F., Seidl, M., Van Gelder, A.: The QBFGallery 2014: the QBF competition at the FLoC olympic games. *JSAT* **9**, 187–206 (2016)
30. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with counterexample guided refinement. *Artif. Intell.* **234**, 1–25 (2016)

31. Janota, M., Marques-Silva, J.: Expansion-based QBF solving versus Q-resolution. *Theor. Comput. Sci.* **577**, 25–42 (2015)
32. Janota, M., Marques-Silva, J.: Solving QBF by clause selection. In: IJCAI, pp. 325–331. AAAI Press (2015)
33. Kleine Büning, H., Bubeck, U.: Theory of quantified Boolean formulas. In: Handbook of Satisfiability, FAIA, vol. 185, pp. 735–760. IOS Press (2009)
34. Kleine Büning, H., Karpinski, M., Flögel, A.: Resolution for quantified Boolean formulas. *Inf. Comput.* **117**(1), 12–18 (1995)
35. Klieber, W., Sapra, S., Gao, S., Clarke, E.: A non-prenex, non-clausal QBF solver with game-state learning. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 128–142. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14186-7_12
36. Letz, R.: Lemma and model caching in decision procedures for quantified Boolean formulas. In: Egly, U., Fermüller, C.G. (eds.) TABLEAUX 2002. LNCS (LNAI), vol. 2381, pp. 160–175. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45616-3_12
37. Lonsing, F., Egly, U.: DepQBF 6.0: a search-based QBF solver beyond traditional QCDCL. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 371–384. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_23
38. Lonsing, F., Egly, U.: Evaluating QBF solvers: quantifier alternations matter. CoRR abs/1701.06612 (2018). <http://arxiv.org/abs/1701.06612>, CP 2018 proceedings version with appendix
39. Lonsing, F., Seidl, M., Van Gelder, A.: The QBF gallery: behind the scenes. *Artif. Intell.* **237**, 92–114 (2016)
40. Marin, P., Narizzano, M., Pulina, L., Tacchella, A., Giunchiglia, E.: Twelve years of QBF evaluations: QSAT is PSPACE-hard and it shows. *Fundam. Inf.* **149**(1–2), 133–158 (2016)
41. Martin, B.: Quantified constraints in twenty seventeen. In: The Constraint Satisfaction Problem: Complexity and Approximability, Dagstuhl Follow-Ups, vol. 7, pp. 327–346. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
42. Meyer, A.R., Stockmeyer, L.J.: The equivalence problem for regular expressions with squaring requires exponential space. In: 13th Annual Symposium on Switching and Automata Theory, pp. 125–129. IEEE Computer Society (1972)
43. Peitl, T., Slivovsky, F., Szeider, S.: Dependency learning for QBF. In: Gaspers, S., Walsh, T. (eds.) SAT 2017. LNCS, vol. 10491, pp. 298–313. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_19
44. Pulina, L., Seidl, M.: QBFEVAL'17: competitive evaluation of QBF solvers (2017). http://www.qbflib.org/event_page.php?year=2017
45. Pulina, L., Tacchella, A.: Treewidth: a useful marker of empirical hardness in quantified Boolean Logic Encodings. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 528–542. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89439-1_37
46. Rabe, M.N., Tentrup, L.: CAQE: a certifying QBF solver. In: FMCAD, pp. 136–143. IEEE (2015)
47. Silva, J.P.M., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, FAIA, vol. 185, pp. 131–153. IOS Press (2009)
48. Stockmeyer, L.J.: The polynomial-time hierarchy. *Theor. Comput. Sci.* **3**(1), 1–22 (1976)
49. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time: preliminary report. In: STOC, pp. 1–9. ACM (1973)

50. Tentrup, L.: On expansion and resolution in CEGAR based QBF solving. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 475–494. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_25
51. Tseitin, G.S.: On the complexity of derivation in propositional calculus. *Stud. Constr. Math. Math. Log.*, 115–125 (1968)
52. Wimmer, R., Reimer, S., Marin, P., Becker, B.: HQSpre – an effective preprocessor for QBF and DQBF. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 373–390. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_21
53. Wrathall, C.: Complete sets and the polynomial-time hierarchy. *Theor. Comput. Sci.* **3**(1), 23–33 (1976)
54. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: Evaluating component solver contributions to portfolio-based algorithm selectors. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 228–241. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_18
55. Zhang, L., Malik, S.: Towards a symmetric treatment of satisfaction and conflicts in quantified Boolean formula evaluation. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 200–215. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46135-3_14



Quantified Valued Constraint Satisfaction Problem

Florent Madelaine¹(✉)  and Stéphane Secouard²

¹ Université Paris-Est Créteil, LACL, Créteil, France
florent.madelaine@uca.fr

² Université Caen Normandie, CNRS, GREYC, Caen, France

Abstract. We study the complexity of the quantified and valued extension of the constraint satisfaction problem (QVCSP) for certain classes of languages. This problem is also known as the weighted constraint satisfaction problem with min-max quantifiers [1].

The multimorphisms that preserve a language is the starting point of our analysis. We establish some situations where a QVCSP is solvable in polynomial time by formulating new algorithms or by extending the usage of collapsibility, a property well known for reducing the complexity of the quantified CSP (QCSP) from Pspace to NP. In contrast, we identify some classes of problems for which the VCSP is tractable but the QVCSP is Pspace-hard.

As a main Corollary, we derive an analogue of Shaeffer's dichotomy between P and Pspace for QCSP on Boolean languages and Cohen *et al.* dichotomy between P and NP-complete for VCSP on Boolean valued languages: we prove that the QVCSP follows a dichotomy between P and Pspace-complete.

Finally, we exhibit examples of NP-complete QVCSP for domains of size 3 and more, which suggest at best a trichotomy between P, NP-complete and Pspace-complete for the QVCSP.

Keywords: Complexity classification · Valued CSP
Quantified CSP · Polymorphisms · Multimorphisms · Collapsibility

1 Introduction

Modern SAT and CSP solvers are quite efficient on industrial instances, so much so that there is a current impetus in the community towards solvers that tackle computational problems that lie beyond NP [2]. Meanwhile on the theoretical front, several proofs [3, 4] have just been proposed for Feder and Vardi celebrated dichotomy conjecture for the CSP [5]. There has been some advances for its quantified counterpart the QCSP that seems to follow a trichotomy between P, NP-complete and Pspace-complete [6–9]. Its optimisation counterpart the VCSP

Supported by Université Clermont Auvergne, CNRS, LIMOS and Université Caen normandie, CNRS, GREYC.

has been classified, first for finite valued cost functions [10] then for arbitrary cost function assuming the dichotomy conjecture holds [11]. The reader may also consult these recent surveys on QCSP [12] and VCSP [13].

It is now well established that the presence or absence of certain well behaved *polymorphisms* of a constraint language characterises the complexity of the corresponding CSP. Schaefer proved a dichotomy in the Boolean case [14], which may be reformulated as follows.

Theorem 1 [15]. *Let Γ be a constraint language over $D = \{0, 1\}$. $\text{CSP}(\Gamma)$ is in P if Γ admits one of the following six good polymorphisms and NP-complete otherwise.*

Good polymorphisms: $\{\text{Mjrty}, \text{Mnrty}, \text{Max}, \text{Min}, \text{Const}_0, \text{Const}_1\}$.

Here, Mjrty denotes the ternary operation that returns its repeated argument, while Mnrty is the ternary minority operation; Max and Min are binary operations that returns the maximum and minimum, respectively; Const_0 and Const_1 are the unary constant operations that sends their argument to 0 and 1 respectively. We delay further formal definitions to the next section.

For the QCSP and VCSP, *surjective polymorphisms* and *fractional polymorphisms* play the same role as polymorphisms for the CSP. As an illustration, let us state two classification results in the Boolean case. For the QCSP, the following dichotomy was announced by Schaefer [14].

Theorem 2 [15, 16]. *Let Γ be a constraint language over $D = \{0, 1\}$. $\text{QCSP}(\Gamma)$ is in P if Γ admits one of the following four good surjective polymorphisms and Pspace-complete otherwise.*

Good surjective polymorphisms: $\{\text{Mjrty}, \text{Mnrty}, \text{Max}, \text{Min}\}$.

For the Boolean VCSP, the good multimorphisms must combine good polymorphisms from Theorem 1, as otherwise the *feasibility* of the VCSP would readily allow to solve a hard CSP.

Theorem 3 [17]. *Let Γ be a valued constraint language on $D = \{0, 1\}$. $\text{VCSP}(\Gamma)$ is in P if it admits at least one of the following eight good multimorphisms. Otherwise, the problem is NP-hard.*

Good multimorphisms for VCSP:

$\{3\text{Mjrty}, 3\text{Mnrty}, 2\text{Mjrty} + \text{Mnrty}, 2\text{Max}, 2\text{Min}, \text{Max} + \text{Min}, \text{Const}_0, \text{Const}_1\}$.

In this paper, we combine universal quantification (QCSP) and valued constraints (VCSP) and work in the framework of the Quantified Valued Constraint Satisfaction Problem (QVCSP). Unbeknownst to us until the reviewers pointed it out, this problem was in fact already introduced in [1] as the *weighted CSPs with min-max quantifiers* and studied from an experimental perspective in the context of solver designs: the authors showed that alpha-beta pruning can be adapted in this context in a relevant fashion. While their name for the problem is very natural in their context, we will stick to our terminology which makes more apparent that we merge the QCSP and the VCSP frameworks. A natural way of building a QCSP instance from a CSP instance consists in assuming that

a malicious opponent or uncertainty with respect to the environment is limiting certain resources or strengthening some constraints after some decisions have been made (see for example [18] for examples of scheduling with opponents). In the same manner, we could build natural example of QVCSP from natural instances of the VCSP.

Note that some of the tractable languages for the VCSP are in fact not genuine valued languages since the costs are the same for all feasible tuples. This is the case for the multimorphisms $3Mjrty$ and $3Mnrty$ in Theorem 3 (actually this is the case for any finite domain size). For such so called *essentially crisp* languages, one can therefore deduce immediately the complexity of their QVCSP from the QCSP classification: they are both *collapsible* (so in NP) and reduce in fact to a CSP in P.

For the QVCSP, the fact that the complexity drops from Pspace to NP is explained by a property known as *switchability* [19–22]. Here we will only consider a restricted form of this property known as *collapsibility*, which asserts that a language is *k-collapsible*, whenever to satisfy an input sentence, it suffices to satisfy all sentences induced from this input sentence by fixing all but a bounded number of universal variables to take the same value. This is true in particular for the languages preserved by Max or Min for $k = 1$. We will show that this approach can be applied in some cases to QVCSP as well, which will in turn lead to tractability in some cases.

In particular, we obtain a complete classification of the complexity of the QVCSP in the Boolean case.

Theorem 4 (main result). *Let Γ be a valued constraint language on $D = \{0, 1\}$. If Γ has one of the following good multimorphism then $QVCSP(\Gamma)$ is tractable, otherwise it is Pspace-hard.*

Good multimorphisms for QVCSP:

$\{3Mjrty, 3Mnrty, 2Mjrty + Mnrty, 2Max, 2Min\}$.

The hardness part of our proof relies on a fairly non trivial case analysis of tractable languages from Theorem 3 that are not tractable according to Theorem 4. We show that we can always express in this case a cost function that is hard for the QVCSP (and was of course not hard for the VCSP). Among other, we borrow and adapt the technique of *compression* used in [17] for the proof of Theorem 3.

The paper is organised as follows. In the next section we recall definitions and notations. In Sect. 3, we introduce the QVCSP and provide some examples of Pspace-hard Boolean QVCSP. In Sect. 4, we show that some valued constraint languages are tractable for essentially trivial reasons: either because they are crisp or because they are non crisp but any instance with “too many” universal quantifiers must be rejected. In Sect. 5, we extend the notion of collapsibility from the QCSP to the valued setting of QVCSP and use it to obtain tractability results for the QVCSP. In Sect. 6, we finish the proof of Theorem 4. Finally we conclude with some remarks.

2 Preliminaries

A *VCSP instance* ϕ is a finite collection of valued constraints over some finite variable set X ranging over a label set D with value in the rationals augmented with the non feasible ∞ . Here a valued constraint is given by a so-called *cost function* ρ from D^n to $\mathbb{Q} \cup \{\infty\}$ for some positive integer n (the arity) and an n -tuple of elements of X (the scope). Given the assignment $\alpha := \{\sigma_x \in D \mid x \in X\}$, we write $\text{obj}(\phi, \alpha)$ as a short hand for $\sum_{\rho(\bar{x}) \text{ in } \phi} \rho(\alpha(\bar{x}))$ where by $\alpha(\bar{x})$ we mean that the value of each variable x from the scope \bar{x} is replaced by its assigned label σ_x . The VCSP is an optimisation problem, the aim of which is to compute an α such that $\text{obj}(\phi, \alpha)$ is minimum. There are two other problems that arise in the context of VCSP.

- decision: given an additional input k in \mathbb{Q} , is there an α such that $\text{obj}(\phi, \alpha)$ is at most k ?
- feasibility: is there an α such that $\text{obj}(\phi, \alpha)$ is finite?

All these problems are NP-hard in general, and a standard way of better understanding the complexity is to study language restrictions, that is restricts costs functions to come from a certain set Γ . A valued constraint language is NP-hard (for the VCSP), iff it contains a finite language for which the VCSP is NP-hard. A cost function is *crisp* (resp., *essentially crisp*) if it ranges over $\{0, \infty\}$ (resp., over $\{c, \infty\}$ for some c in \mathbb{Q}). A language is (essentially) *crisp* if it contains only (essentially) *crisp* cost functions. The *crisp language* associated with a valued constraint language Γ denoted by $\text{crisp}(\Gamma)$ consists of the set of corresponding relations, where $\text{crisp}(\rho)(t) := 0$ iff $\rho(t) < \infty$.

An m -ary *operation* on D is a function $g : D^m \rightarrow D$. Let $\mathcal{O}_D^{(m)}$ denote the set of all m -ary operations on D . An m -ary *fractional operation* is a function ω from $\mathcal{O}_D^{(m)}$ to the positive rationals such that $\sum_g \omega(g) = 1$. The set $\{g \mid \omega(g) > 0\}$ of operations is called the *support* of ω and is denoted by $\text{supp}(\omega)$.

A fractional operation ω is called an m -ary *fractional polymorphism* of a r -ary valued constraint ρ if for any tuples t_1, t_2, \dots, t_m in D^r , it holds that

$$\frac{1}{m}(\rho(t_1) + \rho(t_2) + \dots + \rho(t_m)) \geq \sum_{g \in \mathcal{O}_D^{(m)}} \omega(g)\rho(g(t_1, t_2, \dots, t_m)) \quad (1)$$

where the operations g are applied component-wise. We will alternatively say that a fractional operation *improves* a cost function. A *multimorphism* is a fractional polymorphism with integral weights¹. If ω is a fractional polymorphism of every cost function in a constraint language Γ , then ω is called a fractional polymorphism of Γ . A fractional polymorphism of a crisp language is a collection of *polymorphisms* (one identifies a crisp cost function ρ with the relation

¹ We deviate marginally from the standard definition, which would require to rescale by the arity.

$\{t|\rho(t) < \infty\}$). If ω is a fractional polymorphism of Γ , any g in $\text{supp}(\omega)$ is a polymorphism of $\text{crisp}(\Gamma)$. Rewriting (1) as

$$\rho(t_1) + \rho(t_2) + \dots + \rho(t_m) \geq \sum_{g \in \mathcal{O}_D^{(m)}} \mathbf{m}.\omega(\mathbf{g})\rho(g(t_1, t_2, \dots, t_m)) \tag{2}$$

we will mildly abuse the notation of m -ary fractional polymorphisms in this paper, and write them as a weighted sum of operations of arity m , such that the sum of weights equals m . This explains the notation used in the statement of Theorems 3 and 4, where we listed several examples including ternary fractional operations such as 3Mjrty and 2Mjrty + Mnrty, binary ones such as 2Max and Max + Min and, unary ones Const₀, Const₁.

We recall some operation of importance. For a constant c in D , let Const _{c} denote the unary operation that always return c . Given a total order over D , let Max (resp, Min) denote the binary operation that returns the largest argument. Over the Boolean domain, we shall consider the usual order $0 < 1$. More generally, any partial order over D such that the greatest lower bound of any pair of elements exist, induces naturally a *semi-lattice operation*, which is a binary operation \wedge that is idempotent ($x \wedge x = x$), associative and commutative. The element $\perp := \bigwedge_{d \in D} d$ satisfies for any x in D , $\perp \wedge x = \perp$. If there is a constant \top such that for any x in D , $\top \wedge x = x$, then we say that \top is a *unit* and \wedge a *semi-lattice with unit*. For $1 \leq i \leq 3$, let Mjrty _{i} denote the ternary operation that returns the argument that occurs the most if some are equal, and its i th argument otherwise. When the domain is Boolean, we drop the unnecessary subscript. We define similarly Mnrty _{i} , which returns the least frequent argument if some are equal. A k -ary *Hubie operation*² f over D with respect to a constant c in D is a surjective operation that remains surjective even when any coordinate is fixed to c . That is, $f(c, D, \dots, D) = f(D, c, D, \dots, D) = \dots = f(D, \dots, D, c) = D$. We say that a cost function $\phi(x_1, \dots, x_m)$ can be *expressed* by Γ if there is an instance I of VCSP(Γ) with objective function $\phi_I(x_1, \dots, x_m, x_{m+1}, \dots, x_n)$, such that

$$\phi(x_1, \dots, x_m) = \text{Min}_{x_{m+1}, \dots, x_n} \phi_I(x_1, \dots, x_m, x_{m+1}, \dots, x_n).$$

We can also easily implement cost functions by *scaling* and *translating* that is a cost function $a.\phi + b$, for any $\phi \in \Gamma$, any $a \in \mathbb{Q}^+$ and any $b \in \mathbb{Q}$. Let Γ^* be the closure of Γ under expressibility, scaling and translation. It is known that this closure preserve (in)tractability and that Γ^* is the same as the set of cost functions that are invariant under the fractional polymorphisms of Γ [13, Theorem 35].

3 Definition and Examples of QVCSP

An instance of the *quantified valued constraint satisfaction problem* (QVCSP) is defined as above with the addition of a prefix of quantification \mathcal{P} applying

² It was anonymous in [23] and the term was coined in [21].

to all variables: that is, \mathcal{P} is a strict linear order over variables where variables are either existential or universal. For convenience, we will denote the set of existential variables by $X^{\mathcal{P}}$ and universal variables by $Y^{\mathcal{P}}$. Given an existential variable x of $X^{\mathcal{P}}$, we denote by $Y_x^{\mathcal{P}}$ the set of universal variables that precede x in the prefix order given by \mathcal{P} . When the prefix of quantification is clear from context, we feel free to drop the superscript from our notation.

A *Skolem function* σ_x for the variable x is a D ranging function that takes as input values corresponding to the values of the universal variables that precedes it in \mathcal{P} , that is from $D^{|Y_x^{\mathcal{P}}|}$ to D . If β is a family of Skolem functions for our instance $\beta = \{\sigma_x : D^{|Y_x^{\mathcal{P}}|} \rightarrow D \mid x \in X\}$ (we will call such a family a *strategy* for our instance) and $\pi : Y \rightarrow D$ an assignment of the universal variables, we write $\beta \circ \pi$ for the assignment to the variables which assigns a universal variable y in Y to $\pi(y)$ and an existential variable x in X to $\sigma_x(\pi|_{Y_x^{\mathcal{P}}})$, where $\pi|_{Y_x^{\mathcal{P}}}$ denotes the restriction to $Y_x^{\mathcal{P}}$ of π .

We are now in a game setting pitching a universal player (male) and an existential player (female). Informally, she is trying to give a label to existential variables with the long term view of optimising the objective function, while he is a malicious opponent trying to prevent her from doing so. She tries to minimise the objective no matter what her opponent plays. This is reasonable if she knows that he is maliciously trying to make sure that after play her objective is as large as possible. We extend therefore the objective function to quantified valued constraints and let $\text{obj}(\phi, \beta) := \max_{\pi: Y \rightarrow D} \text{obj}(\phi, \beta \circ \pi)$. We will consider the QVCSP to be the optimisation problem, the aim of which is to compute a β such that $\text{obj}(\phi, \beta)$ is minimum. We will not as such request that β be given in full as it would be of size at least D^m where m is the number of universal variables. Instead, we will ask for a procedure that can play the underlying game according to the strategy β . Like for the VCSP, there are again two natural decision problems that arise:

- decision: given an additional input k in \mathbb{Q} , is there a β such that $\text{obj}(\phi, \beta)$ is at most k ?
- feasibility: is there a β such that $\text{obj}(\phi, \beta)$ is finite?

Note that this definition extends naturally the usual semantic of the QCSP and the feasibility question for the QVCSP amounts to solving the QCSP for the underlying crisp language. This means that the above three problems are Pspace-hard in general, and we study in this paper their restrictions to a valued constraint language Γ .

Example 1. Let Γ_{nae} be the boolean constraint language that consists of the cost function.

$$\rho_{\text{nae}}(x, y, z) = \begin{cases} \infty & \text{if } x = y = z \\ 0 & \text{otherwise} \end{cases}$$

This language is crisp and we know that the complexity for the VCSP is that of the corresponding CSP, namely NP-complete, and that for the QVCSP, we should look at the QCSP, well known to be Pspace-complete [12].

Example 2. Let Γ_{neq} be the boolean constraint language that consists of the cost function

$$\rho_{\text{neq}}(x, y) = \begin{cases} 0 & \text{if } x \neq y \\ 1 & \text{otherwise} \end{cases}$$

For this non crisp language, we again get an NP-hard VCSP but not because of the feasibility which is tractable, but because of the optimisation, by reduction from MAX Sat for XOR [17]. Alternatively, one can simulate a variant of ρ_{nae} :

$$\rho'_{\text{nae}}(x, y, z) := \rho_{\text{neq}}(x, y) + \rho_{\text{neq}}(x, z) + \rho_{\text{neq}}(y, z) - 1 = \begin{cases} 2 & \text{if } x = y = z \\ 0 & \text{otherwise} \end{cases}$$

We can reduce $\text{VCSP}(\Gamma_{\text{nae}})$ to $\text{VCSP}(\Gamma_{\text{neq}})$ by replacing every occurrence of the cost function by ρ'_{nae} . The former instance holds iff the latter has a solution reaching an objective of 0. The same reduction applies for the QVCSP, whose decision version is therefore Pspace-complete.

Example 3. The following boolean cost function

$$\rho_{\text{eq}}(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{otherwise} \end{cases}$$

together with two unary crisp cost functions that encodes the constants 0 and 1 forms the boolean language Γ_{cut} , whose VCSP corresponds essentially to the problem MIN-CUT and is tractable [17]. In contrast we will show below that the language $\Gamma_{\text{eq}} = \{\rho_{\text{eq}}\}$ has already a QVCSP that is Pspace-hard.

Proposition 1. *The QVCSP for the constraint language Γ_{eq} is Pspace-hard.*

Proof. We reduce the decision version of QVCSP for Γ_{neq} (see example above) to that of Γ_{eq} as follows.

Given an instance ϕ of the former with a quantifier prefix \mathcal{P} , we reduce to the instance $\tilde{\phi}$ obtained by replacing every occurrence of the cost function ρ_{neq} by ρ_{eq} in ϕ and picking the dual quantifier prefix $\tilde{\mathcal{P}}$ (that is turn existential variables to universal and vice versa).

Let N be the number of occurrences of the ρ_{neq} cost function in ϕ . We claim that the objective for $\tilde{\phi}$ must be more than $N - k$, iff the objective for ϕ is less than k .

Indeed, otherwise pitting a strategy $\tilde{\beta}$ for $\tilde{\phi}$ that would attain an objective of less than N minus k , against any strategy β for ϕ in the game for ϕ , we would obtain a final objective of more than k .

The dual argument applies for the other direction, which proves our claim.

The claim gives us the (Turing) reduction. We answer the opposite answer of that for $\tilde{\phi}$ with the threshold $N - k$. □

4 Some Tractable Languages

4.1 Essentially Crisp Languages

We can deduce the complexity of such languages from the complexity of the associated QCSP.

Proposition 2. *Let Γ be a valued constraint language over some finite set D . If Γ admits 3Mjrty or 3Mnrty as a multimorphism, where Mjrty (respectively, Mnrty) is any majority (respectively, minority) operation, then $\text{QVCSP}(\Gamma)$ is tractable.*

Proof. By Proposition 6.20 (majority) and 6.22 (minority) in [17], Γ is an essentially crisp language. Thus the problem $\text{QVCSP}(\Gamma)$ is the same as $\text{QCSP}(\Gamma')$ where $\Gamma' = \text{crisp}(\Gamma)$. By construction, Γ' admits Mjrty or Mnrty as a polymorphism, which are known to be tractable by Theorem 4.2 (mal'tsev) and 4.5 (near-unanimity) in [6]. \square

Remark 1. The above can be generalised to a language that admits a multimorphism $3f$ where f is Mal'tsev or a multimorphism $k.f$ where f is a k -ary near-unanimity operation.

4.2 Permutations and Unary

The proof principle used to discard universal quantifiers for the language of the following result is reminiscent of the case of a language that consists of a single bipartite or a single disconnected graph for the QCSP [24]. In a nutshell, an instance boils down to a collection (conjunction) of instances with a prefix of quantification with at most one leading universal variable or it must be rejected.

Theorem 5. *Let Γ be a valued constraint language over some finite set D . If Γ admits $\text{Mjrty}_1 + \text{Mjrty}_2 + \text{Mnrty}_3$ as a multimorphism, then $\text{QVCSP}(\Gamma)$ is tractable.*

Proof. By Theorem 6.25 in [17], any cost function from Γ can be expressed as a sum of unary cost functions and binary permutation restrictions. The latter are crisp cost functions with costs ranging in $\{0, \infty\}$, that amount to a restricted permutation in the sense that for any x , there is at most one y_2 such that $\phi(x, y_2)$ holds (has non ∞ weight) and at most one y_1 such that $\phi(y_1, x)$ holds.

Our algorithm will apply some simple preprocessing and detect that the instance is not feasible or it will deduce that each connected component of the constraint graph contains at most one universal variable and by some simple case analysis deduce the (worst) cost for these components. In effect this reduces the instance to a VCSP instance for which a simple algorithm is already known.

Let ϕ be a permutation restriction occurring in the instance.

If $\exists x \forall y \phi(x, y)$ occurs in the instance then it is not feasible since any but at most one value for y will yield an objective of ∞ for a given value of x . In this

case, we may answer ∞ . Of course, the symmetric case of $\exists x \forall y \phi(y, x)$ is dealt with in the same way. We ignore symmetric cases from now on.

Similarly, if $\forall y_1 \forall y_2 \phi(y_1, y_2)$ occurs in the instance, then we may answer ∞ .

In the degenerate case of $\forall y \phi(y, y)$, we may also answer ∞ unless ϕ is the crisp function for equality over D , in which case, we may simply discard $\phi(y, y)$ from the sum.

So we may assume from now on that any occurrence of a permutation restriction is of the form $\forall y \exists x \phi(x, y)$. If there is one y_0 such that $|\{x \mid \phi(x, y_0) = 0\}| = 0$ then we may answer ∞ . Otherwise, let $\zeta(y)$ be the unique x such that $\phi(x, y) = 0$. The only Skolem function for x that yields feasibility is essentially unary and depends only of y : $\sigma_x(y) = \zeta(y)$.

If there is a path $y_1, x_1, x_2, \dots, x_n$ where y_1 is universal and x_1, x_2, \dots, x_n are existential variables in the constraint graph then there are some permutations ζ_i on D such that the only Skolem functions for these existential variables that could possibly yield feasibility are of the form $\sigma_{x_1}(y) = \zeta_1(y_1)$, $\sigma_{x_2}(y) = \zeta_2(\zeta_1(y_1)) \dots \sigma_{x_n}(y) = \zeta_n(\dots \zeta_2(\zeta_1(y_1)) \dots)$. If there is an edge from x_n to some universal variable y_2 then the instance is necessarily unfeasible since any but one value for y_2 will yield an objective of ∞ for a given value of x_n .

We can solve in parallel the part of the instance induced by each connected component of the constraint graph, and we may assume that a connected component of this graph contains at most one universal variable that is quantified ahead of the existential variables of this connected component.

A connected component that does not contain any universal variable can be solved efficiently by some simple propagation (see [17]).

If a universal variable y occurs only within the scope of unary constraints then we simply assume that y takes the value yielding the worst cost.

More generally, for each connected component that contains one universal variable y , we can check in parallel for all values d of y , the corresponding cost $M_{y=d}$ for the component (all other variables are now fixed). Let M be the maximum combined cost among $M_{y=d}$. \square

Remark 2. The tractable languages for the QCSP from [24] mentioned above can be shown to exhibit collapsibility thanks to specific polymorphisms (see examples 2 and 3 in [23]). While we shall proceed similarly for the valued languages of the next section with a suitable multimorphism, we do not yet know of a multimorphism that witnesses directly “collapsibility” for the language of Theorem 5.

5 Collapsibility in the Valued Settings

Following Chen [20], for an input of the QVCSP with m universal variables, we restrict the universal opponent to play universal variables from a specific set of tuples, and investigate the interpolation of unrestricted game from restricted (small sized) ones in the presence of good multimorphisms.

As a concrete application, we will see the case of a language closed under the multimorphism $2.g$ (2 times g) where g is a semi-lattice with unit \top . On

an instance involving cost functions improved by this multimorphism, we may interpolate a winning strategy from winning strategies for all instances induced by replacing all but one universal variable by \top .

The necessary definitions and notations to discuss such an interpolation in general can be a bit off-putting, and the keen reader may refer to the appendix. Here, we will only eventually state our tractability result and give first a detailed and concrete example to illustrate it.

Example 4. Consider the instance $\forall y_1 \exists x_1 \forall y_2 \forall y_3 \exists x_2 \phi(y_1, x_1, y_2, y_3, x_2)$, where ϕ is the 5-ary Boolean cost function such that the cost of $(0, 0, 0, 0, 0)$ is 51, that of $(0, 0, 0, 0, 1)$ and $(0, 0, 0, 1, 0)$ is ∞ , that of $(0, 0, 0, 1, 1)$ is 21, etc. This instance is depicted on Fig. 1. It can be checked that the cost function admits 2Max as a multimorphism: that is, the sum of the costs of any two tuples dominates twice the cost of their max taken component-wise. For example, when $t_1 = (0, 0, 1, 0, 1)$ and $t_2 = (1, 0, 0, 1, 1)$; their max is $t_3 = (1, 0, 1, 1, 1)$; the costs are $\phi(t_1) = 51$, $\phi(t_2) = 13$ and $\phi(t_3) = 5$; it is indeed the case that $10 = 2 \times 5 \leq 51 + 13 = 64$.

We replace all but one universal variable by 0 (the value of the unit \top for the specific case of Max) and derive three restricted games, which amounts to solve the instances that are depicted on Fig. 2. Of course, each restricted game is a relaxation of the original instance. Thus, if one of them is not feasible then the original instance is also not feasible. The same argument applies to the objective reached by feasible instances.

The important point is that the converse holds since we can interpolate a strategy for the original instance from three strategies for the restricted games, in a way that can only improve them.

In what follows, we will assume that we have at our disposal the three strategies that are optimal for each restricted game.

Imagine that the first universal quantifier takes value 1, that is $y_1 = 1$. We will play also $y_1 = 1$ in the first restricted game and $y_1 = 0$ in the other two restricted games (we may not do otherwise). Observe that the max of the triple $(1, 0, 0)$ is 1. Next, we look up where the subsequent existential variable x_1 is played in each restricted game. For example, we must have $x_1 = 1$ in the first restricted game, and $x_1 = 0$ in the two other games (otherwise, we would end up being necessarily unfeasible). We apply max to this triple $(1, 0, 0)$ and play $x_1 = 1$ in the original game. We proceed in this fashion going back and forth taking antecedent and image under max. For example, $y_2 = 0$ brings us back to y_2 being played on $(0, 0, 0)$ in the three games, and $y_3 = 1$ brings us back to y_3 being played on $(0, 0, 1)$ in the three games. In these three games x_2 must be played on 1, 0, and 1, respectively. We play x_2 on their maximum which is 1. The “branches” of play in the four games alluded to above are highlighted on Figs. 1 and 2.

The fact that Max is surjective means that we can always go back. The fact that 2Max is a multimorphism means (with a little bit of work) that the strategy we have interpolated from those for the restricted games can only improve them.

In the previous example, we have explained how a general strategy can be interpolated from a set of strategies applying to restricted games, where we

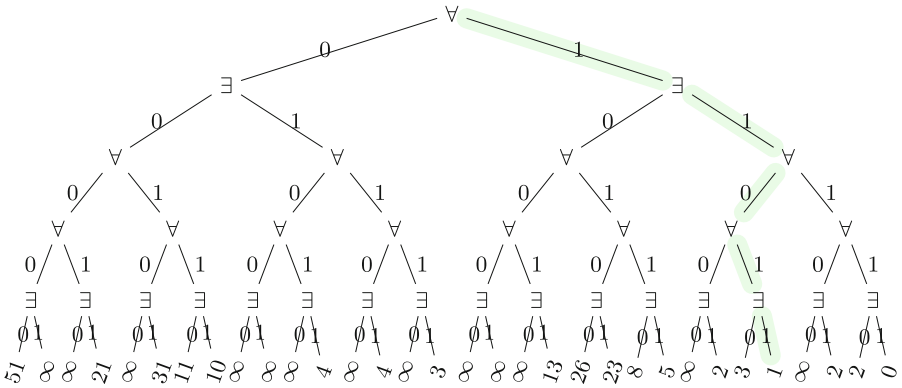


Fig. 1. An instance of the QVCSP

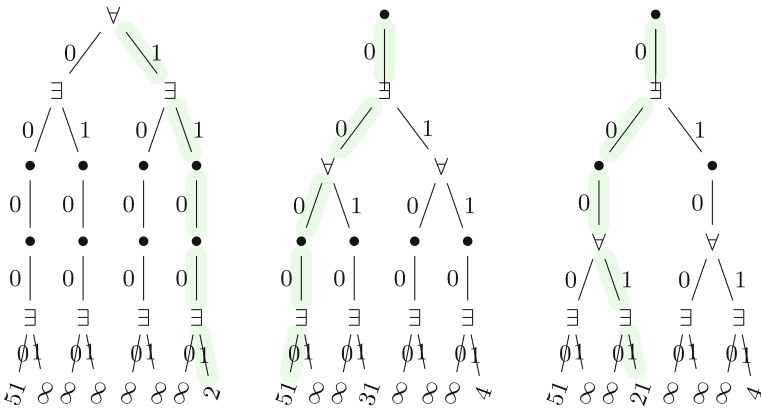


Fig. 2. Restricted Games: from left to right, we keep the first, second and third universal quantifier. The other universal variables are assumed to take value 0 (we write a \bullet to denote that they are pinned to a constant).

are left with a single universal quantifier. Each such strategy can be computed by an adaptation of Generalized Arc Consistency that runs also in polynomial time, and there are linearly many such strategies to compute. Thus, we have a tractable QVCSP in this case.

Theorem 6. *Let Γ be a valued constraint language. Let g be a semi-lattice with unit \top . If Γ admits 2 times g ($2.g$) as a multimorphism the $QVCSP(\Gamma)$ is tractable.*

6 Proof of Theorem 4

We have proved in Sect. 4 that any valued Boolean constraint language that admits one of the good multimorphism from the statement is tractable: for

3Mjrty and 3Mnrty by Proposition 2, for 2Mjrty + Mnrty by Theorem 5, for 2Max and 2Min by Theorem 6. So, we are left with the hardness part of the statement.

If a constraint language does not admit any of these multimorphisms, and is essentially crisp it must be hard by Theorem 2. If a constraint language does not admit any of the multimorphisms of Theorem 3, and is not essentially crisp, then by Lemma 7.10 in [17] Γ^* contains ρ_{neq} which has a Pspace hard QVCSP as seen in Example 2.

The next lemma concludes the proof, as we show that even if a language must admit **all** multimorphisms from Theorem 3 that are not in Theorem 4, then it can simulate a cut function.

A language Γ' that admits less multimorphisms than Γ can express any finite subset of Γ^* . So any such Γ' would also simulate a cut function.

A *cut function* is a binary function from $\{0, 1\}^2$ in $\mathbb{Q} \cup \{\infty\}$ of the following form $\phi_{cut_\alpha}(x, y) = \begin{cases} \alpha & \text{if } x = y, \\ \beta & \text{otherwise} \end{cases}$, where $\alpha, \beta \in \mathbb{Q} \cup \{\infty\}$ with $\alpha < \beta < \infty$.

The QVCSP of a cut function is Pspace-hard by Proposition 1, since ρ_{eq} can be simulated by scaling and translating from any cut function.

Since we are in a quantified and valued setting, we will be able to use expressivity, scaling and translating as for the VCSP but also universal quantifiers as for the QCSP. We will call this *simulation with some universal quantifiers* in what follows to stress that we go beyond the $*$ closure from the VCSP.

Lemma 1. *Let Γ be a valued Boolean constraint language. If Γ admits $Const_1, Const_0$ and $Max + Min$ as multimorphism but no multimorphism from $\{3Mjrty, 3Mnrty, 2Mjrty + Mnrty, 2Max, 2Min\}$ then Γ can simulate with some universal quantifiers a cut function.*

Consequently, the decision problem of the QVCSP of Γ is Pspace-complete.

The rest of this section is devoted to a proof of this Lemma.

Fact 1. *If Γ admits $Max + Min$ as a multimorphism then $crisp(\Gamma)$ admits Mjrty as a polymorphism.*

Proof. $crisp(\Gamma)$ admits both Max and Min as polymorphisms and the majority can be defined as follows:

$$Mjrty(a, b, c) := \text{Max}[\text{Max}(\text{Min}(a, b), \text{Min}(a, c)), \text{Min}(b, c)].$$

Fact 2. *If Γ does not admit 3Mjrty as a multimorphism and $crisp(\Gamma)$ does admit Mjrty as a polymorphism then Γ is not essentially crisp.*

Proof. Let ρ be a cost function in Γ and u, v, w such that $\rho(u), \rho(v), \rho(w) < \infty$. Since Mjrty is a polymorphism of $crisp(\Gamma)$ then $\rho(Mjrty(u, v, w)) < \infty$. If ρ is essentially crisp then $3\rho(Mjrty(u, v, w)) = 3\rho(u) = \rho(u) + \rho(v) + \rho(w)$. If Γ was essentially crisp then it would admit 3Mjrty as a multimorphism which would contradict our assumption.

Recall that ρ is *finitely modular*, whenever for all tuples s, t such that $\phi(s), \phi(t), \phi(\text{Max}(s, t))$, and $\phi(\text{Min}(s, t))$ have finite costs, we have that $\phi(s) + \phi(t) = \phi(\text{Max}(s, t)) + \phi(\text{Min}(s, t))$.

Fact 3. *If Γ does not admit $2\text{Mjrty} + \text{Mnrty}$ as a multimorphism and $\text{crisp}(\Gamma)$ does admit Mjrty as a polymorphism then there exist a cost function in ρ that is not finitely modular or $\text{crisp}(\rho)$ does not admit Mnrty as a polymorphism.*

Proof. Corollary 6.26 in [17] establishes that a cost function ρ does admit $2\text{Mjrty} + \text{Mnrty}$ as a multimorphism iff it is both finitely modular and $\text{crisp}(\rho)$ admits as polymorphisms both Mjrty and Mnrty .

Fact 4. *If Γ is not essentially crisp, and it admits Const_0 and Const_1 as multimorphisms, and $\text{crisp}(\Gamma)$ admits Mjrty but does not admit Mnrty then Γ^* contains a cut function.*

Before proving this, let us point out that this means that we are only left with the case when there is a ρ that is not finitely modular, a case that we will settle in the last Fact.

Proof. We follow the same argument as in case 3 of the proof of Theorem 6.27 from [17] and establish that Γ contains a binary cost function ρ such that for exactly one $(a, b) \in D^2$ there is $\rho(a, b) = \infty$ (other values being finite). Since Γ admits Const_0 and Const_1 as multimorphisms, we know that $\rho(0, 0) = \rho(1, 1) \leq \rho(b, a) < \rho(a, b) = \infty$. W.l.o.g. up to symmetry, we can suppose that $a = 0$ and $b = 1$ and we have $\rho(0, 0) = \rho(1, 1) \leq \rho(1, 0) < \rho(0, 1) = \infty$.

We must ensure $\rho(1, 0) > \rho(0, 0)$ for our next construction to work. If it is not the case, then since Γ is not essentially crisp and has the multimorphisms Const_0 and Const_1 , there is a cost function ρ_m (of arity m) and a m -tuple u such that $\rho_m(0, \dots, 0) = \rho_m(1, \dots, 1) < \rho_m(u) < \infty$. Let ρ_2 be the binary function obtained by $\rho_2(x_1, x_0) = \rho_m(x_{u[1]}, \dots, x_{u[m]})$. We do not know for sure the value of $\rho_2(0, 1)$ but we know that $\rho_2(0, 0) = \rho_2(1, 1) < \rho_2(1, 0) = \rho_m(u) < \infty$.

Let $\rho_3(x, y) := \rho(x, y) + \rho_2(x, y)$. By construction, $\rho_3(0, 0) = \rho_3(1, 1) < \rho_3(1, 0) < \rho_3(0, 1) = \infty$.

The last function which is the desired cut function is created by expressibility as follows:

$$\rho_4(x, y) := \text{Min}_{z,t}[\rho_3(x, t) + \rho_3(z, t) + \rho_3(z, y) + \rho_3(y, z) - 4\rho_3(0, 0)]. \quad \square$$

The next step will rely heavily on the technique of *compression* from [17], which we shall adapt to our purpose. Given an m -ary cost function ρ_m and two m -tuples u and v , let the *compression* ρ_4 of ρ_m w.r.t. u and v is defined as: $\rho_4(x_{00}, x_{01}, x_{10}, x_{11}) = \rho(x_{u[1]v[1]}, x_{u[2]v[2]}, \dots, x_{u[m]v[m]})$. One can verify that $\rho_4(0, 0, 0, 1) = \rho_m(\text{Min}(u, v))$, $\rho_4(0, 0, 1, 1) = \rho_m(u)$, $\rho_4(0, 1, 0, 1) = \rho_m(v)$ and $\rho_4(0, 1, 1, 1) = \rho_m(\text{Max}(u, v))$.

Next, we want to ensure that the first and last coordinate of ρ_4 must take values 0 and 1 in order to simulate the binary cost function $\rho_4(0, x_1, x_2, 1)$.

Fact 5. *If Γ is not essentially crisp and admits Const_1 and Const_0 as multimorphisms, then Γ^* contains a cut function or a small quantified instance with 2 free variables x_1 and x_2 and two universal variables built from cost functions from Γ together with ρ_4 allows to simulate the binary cost function $\rho_4(0, x_1, x_2, 1)$ for any m -ary cost function ρ_m from Γ .*

Proof. Let $\rho_2(0, 0) = \rho_2(1, 1) < \rho_2(1, 0) = \rho_m(u) < \infty$ be defined as in the proof of the previous fact (we only need the assumptions that Γ is not essentially crisp and admits Const_1 and Const_0 as multimorphisms). If $\rho_2(0, 1)$ is also finite then $\rho_2(x, y) + \rho_2(y, x)$ expresses a cut function and we are done.

Otherwise, $\exists x_1 \exists x_2 \forall y_1 \forall y_2 \rho_2(x_1, y_1) + \rho_2(y_2, x_2) - 2\rho_2(1, 0)$ forces $x_1 = 0$ and $x_2 = 1$ (because this is the only way to avoid an infinite cost).

So for any cost function ρ_m from Γ and its compression ρ_4 , if we insert at the beginning of an instance $\exists x_1 \exists x_2 \forall y_1 \forall y_2 \rho_2(x_1, y_1) + \rho_2(y_2, x_2) - 2\rho_2(0, 1)$, all subsequent constraint of the form $\rho_4(x_1, x, y, x_2)$ plays the same role as $\rho_4(0, x_1, x_2, 1)$.

Fact 6. *If there exists a cost function which is not finitely modular in Γ , which does admit $\text{Min} + \text{Max}$ as a multimorphism but does not admit either 2Max or 2Min as a multimorphism then Γ can simulate with some universal variables a cut function.*

Proof. Since 2Max is not a multimorphism there is a function ρ_{NMax} and u, v such that $2\rho_{\text{NMax}}(\text{Max}(u, v)) > \rho_{\text{NMax}}(u) + \rho_{\text{NMax}}(v)$. Both $\rho_{\text{NMax}}(\text{Max}(u, v)) < \infty$ and $\rho_{\text{NMax}}(\text{Min}(u, v)) < \infty$ because $\text{Min} + \text{Max}$ is a multimorphism and so both Min and Max are polymorphisms of $\text{crisp}(\Gamma)$.

By the binarisation method of the compression from the previous fact, either we have a cut function and we are done or we can simulate the binary function $\rho_{2\text{NMax}}$ and let $\rho_M(x, y) = \rho_{2\text{NMax}}(x, y) + \rho_{2\text{NMax}}(y, x) - 2\rho_{2\text{NMax}}(0, 0)$.

$$\text{We have } \rho_M : \begin{cases} 1, 1 \mapsto A + \epsilon_1 \text{ with } 0 < \epsilon_1 \leq A \\ 1, 0 \mapsto A \text{ with } A > 0 \\ 0, 1 \mapsto A \\ 0, 0 \mapsto 0 \end{cases}$$

$0 < \epsilon_1$ because ρ_{NMAX} does not have the multimorphism 2Max and $\epsilon_1 \leq A$ because ρ_{NMAX} has the multimorphism $\text{Min} + \text{Max}$.

There is a function ρ_{NMod} which is not finitely modular and admits $\text{Min} + \text{Max}$ as a multimorphism. So there are u, v such that $\rho_{\text{NMod}}(\text{Max}(u, v)) + \rho_{\text{NMod}}(\text{Min}(u, v)) < \rho_{\text{NMod}}(u) + \rho_{\text{NMod}}(v) < \infty$. By the binarisation method from the previous fact, either we have a cut function and we are done or we can simulate $\rho_{2\text{NMod}}$ and define $\rho_s(x, y) := \rho_{2\text{NMod}}(x, y) + \rho_{2\text{NMod}}(y, x) - 2\rho_{2\text{NMod}}(0, 0)$.

$$\text{By construction, we have, } \rho_s : \begin{cases} 1, 1 \mapsto b_0 < 2b \\ 1, 0 \mapsto b \\ 0, 1 \mapsto b \\ 0, 0 \mapsto 0 \end{cases} .$$

Let α be a positive integer such that $\alpha > \frac{b-b_0}{\epsilon_1}$ and $\varphi_M := \alpha\rho_M + \rho_s$

- $\varphi_M(1, 1) = \alpha A + \alpha\epsilon_1 + b_0 > \alpha A + b - b_0 + b_0 = \varphi_M(1, 0)$
- $\varphi_M(1, 1) = (A + \epsilon_1)\alpha + b_0 < 2A\alpha + 2b = 2(\alpha A + b) = 2\varphi_M(1, 0)$
- $\varphi_M(1, 0) = \varphi_M(0, 1) = b + \alpha A \geq b + \alpha\epsilon_1 > b + b - b_0 > 0$
- $\varphi_M(0, 0) = 0$

We have $\varphi_M : \begin{cases} 1, 1 \mapsto M + \epsilon_M \text{ with } 0 < \epsilon_M < M \\ 1, 0 \mapsto M \text{ with } M > 0 \\ 0, 1 \mapsto M \\ 0, 0 \mapsto 0 \end{cases}$

A similar proof with 2Min instead of 2Max can be used to construct the

binary function ρ_m such that, $\varphi_m : \begin{cases} 1, 1 \mapsto 0 \\ 1, 0 \mapsto m \text{ with } m > 0 \\ 0, 1 \mapsto m \\ 0, 0 \mapsto m + \epsilon_m \text{ with } 0 < \epsilon_m < m \end{cases}$

We define the function $\rho(x, y) = (m + \epsilon_m)\rho_M + (M + \epsilon_M)\rho_m$ and we have:

- $\rho(1, 1) = \rho(0, 0) = mM + m\epsilon_M + M\epsilon_m + \epsilon_M\epsilon_m$
- $\rho(1, 0) = \rho(0, 1) = mM + m\epsilon_M + M\epsilon_m + mM$
- $\epsilon_M\epsilon_m < mM$

So ρ is a cut function as required. □

7 Conclusion

We have studied the quantified valued constraint satisfaction problem, also known as the weighted CSPs with min-max quantifiers, and established preliminary results regarding its complexity when restricted by a valued language.

Without introducing any new Galois connection and using only the tools for the VCSP, and only adapting collapsibility from the QCSP, we get several tractability and intractability results, which allows us to derive in particular a dichotomy for the Boolean case. The proof is somewhat complex, and we plan to introduce the correct Galois connection for the QVCSP in the hope that it will allow to streamline this proof, and extend this result to larger domains.

Another line of enquiry would be to better understand collapsibility in the context of valued constraints. Our current attempt does not seem to provide us with transitivity as it does in the non valued case.

Finally, let us note that any attempt at classifying the QVCSP for 3 or more elements might hit the same hurdle as in the case of the QCSP. We can easily build problems that fall in NPO and are NP-hard. For example consider

$$\rho : \begin{cases} \{0, 1, 2\} \rightarrow \mathbb{Q} \cup \{\infty\} \\ (x, y) \mapsto \begin{cases} \rho_{\text{neq}}(x, y) \text{ if } (x, y) \in \{0, 1\} \\ \infty \text{ otherwise} \end{cases} \end{cases}$$

Every instance with a universal quantifier y can be trivially answered as the objective is ∞ as soon as y is 2. We are left with existential instances which likewise must play on $\{0, 1\}$. Consequently, QVCSP has the same complexity as the VCSP on ρ_{neq} , namely it is NP-hard and in NPO.

Acknowledgments. The authors are thankful to the three anonymous reviewers for their valuable comments which have helped us improve the manuscript.

References

1. Lee, J.H., Mak, T.W.K., Yip, J.: Weighted constraint satisfaction problems with min-max quantifiers. In: IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011, Boca Raton, FL, USA, 7–9 November 2011, pp. 769–776. IEEE Computer Society (2011). <https://doi.org/10.1109/ICTAI.2011.121>
2. Beyond np. <http://beyonddnp.org/>. Accessed 21 June 2017
3. Bulatov, A.A.: A dichotomy theorem for nonuniform CSPs. In: Umans [26], pp. 319–330. <https://doi.org/10.1109/FOCS.2017.37>
4. Zhuk, D.: A proof of CSP dichotomy conjecture. In: Umans [26], pp. 331–342. <https://doi.org/10.1109/FOCS.2017.38>
5. Feder, T., Vardi, M.Y.: The computational structure of monotone monadic SNP and constraint satisfaction: a study through datalog and group theory. *SIAM J. Comput.* **28**(1), 57–104 (1998). <https://doi.org/10.1137/S0097539794266766>
6. Börner, F., Bulatov, A.A., Chen, H., Jeavons, P., Krokhin, A.A.: The complexity of constraint satisfaction games and QCSP. *Inf. Comput.* **207**(9), 923–944 (2009)
7. Martin, B.: QCSP on partially reflexive forests. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 546–560. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23786-7_42
8. Madelaine, F., Martin, B.: QCSP on partially reflexive cycles – the wavy line of tractability. In: Bulatov, A.A., Shur, A.M. (eds.) CSR 2013. LNCS, vol. 7913, pp. 322–333. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38536-0_28
9. Dapic, P., Markovic, P., Martin, B.: Quantified constraint satisfaction problem on semicomplete digraphs. *ACM Trans. Comput. Log.* **18**(1), 2:1–2:47 (2017). <https://doi.org/10.1145/3007899>
10. Thapper, J., Zivny, S.: The complexity of finite-valued CSPs. *J. ACM* **63**(4), 37:1–37:33 (2016). <https://doi.org/10.1145/2974019>
11. Kolmogorov, V., Krokhin, A.A., Rolinek, M.: The complexity of general-valued CSPs. In: Guruswami, V. (ed.) IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17–20 October 2015, pp. 1246–1258. IEEE Computer Society (2015). <https://doi.org/10.1109/FOCS.2015.80>
12. Martin, B.: Quantified constraints in twenty seventeen. In: Krokhin and Zivny [25], pp. 327–346. <https://doi.org/10.4230/DFU.Vol7.15301.12>
13. Krokhin, A.A., Zivny, S.: The complexity of valued CSPs. In: The Constraint Satisfaction Problem: Complexity and Approximability [25], pp. 233–266. <https://doi.org/10.4230/DFU.Vol7.15301.9>
14. Schaefer, T.: The complexity of satisfiability problems. In: STOC (1978)
15. Creignou, N., Khanna, S., Sudan, M.: Complexity Classifications of Boolean Constraint Satisfaction Problems. Society for Industrial and Applied Mathematics, Philadelphia (2001)

16. Dalmau, V.: Some dichotomy theorems on constant-free quantified boolean formulas. Technical report LSI-97-43-R., Departament LSI, Universitat Pompeu Fabra (1997)
17. Cohen, D.A., Cooper, M.C., Jeavons, P., Krokhin, A.A.: The complexity of soft constraint satisfaction. *Artif. Intell.* **170**(11), 983–1016 (2006). <https://doi.org/10.1016/j.artint.2006.04.002>
18. Benedetti, M., Lallouet, A., Vautard, J.: Modeling adversary scheduling with $qcsp^+$. In: Wainwright, R.L., Haddad, H. (eds.) *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC)*, Fortaleza, Ceara, Brazil, 16–20 March 2008, pp. 151–155. ACM (2008). <https://doi.org/10.1145/1363686.1363727>
19. Chen, H.: The complexity of quantified constraint satisfaction: collapsibility, sink algebras, and the three-element case. *SIAM J. Comput.* **37**(5), 1674–1701 (2008)
20. Chen, H.: Quantified constraint satisfaction and the polynomially generated powers property. *Algebra Universalis* **65**(3), 213–241 (2011). <https://doi.org/10.1007/s00012-011-0125-4>. an extended abstract appeared in *ICALP B 2008*
21. Carvalho, C., Madelaine, F.R., Martin, B.: From complexity to algebra and back: digraph classes, collapsibility, and the PGP. In: *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, 6–10 July 2015*, pp. 462–474. IEEE Computer Society (2015). <https://doi.org/10.1109/LICS.2015.50>
22. Carvalho, C., Martin, B., Zhuk, D.: The complexity of quantified constraints. *CoRR abs/1701.04086* (2017). <http://arxiv.org/abs/1701.04086>
23. Chen, H.: Meditations on quantified constraint satisfaction. *CoRR abs/1201.6306* (2012)
24. Martin, B., Madelaine, F.: Towards a trichotomy for quantified H -Coloring. In: Beckmann, A., Berger, U., Löwe, B., Tucker, J.V. (eds.) *CiE 2006. LNCS*, vol. 3988, pp. 342–352. Springer, Heidelberg (2006). https://doi.org/10.1007/11780342_36
25. Krokhin, A.A., Zivny, S. (eds.): *The Constraint Satisfaction Problem: Complexity and Approximability, Dagstuhl Follow-Ups*, vol. 7. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017). <http://www.dagstuhl.de/dagpub/978-3-95977-003-3>
26. Umans, C. (ed.): *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, 15–17 October 2017*. IEEE Computer Society (2017). <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8100284>



MLIC: A MaxSAT-Based Framework for Learning Interpretable Classification Rules

Dmitry Malioutov¹ and Kuldeep S. Meel²(✉)

¹ T. J. Watson IBM Research Center, Yorktown Heights, USA
dmal@alum.mit.edu

² School of Computing, National University of Singapore, Singapore, Singapore
meel@comp.nus.edu.sg

Abstract. The wide adoption of machine learning approaches in the industry, government, medicine and science has renewed the interest in interpretable machine learning: many decisions are too important to be delegated to black-box techniques such as deep neural networks or kernel SVMs. Historically, problems of learning interpretable classifiers, including classification rules or decision trees, have been approached by greedy heuristic methods as essentially all the exact optimization formulations are NP-hard. Our primary contribution is a MaxSAT-based framework, called *MLIC*, which allows principled search for interpretable classification rules expressible in propositional logic. Our approach benefits from the revolutionary advances in the constraint satisfaction community to solve large-scale instances of such problems. In experimental evaluations over a collection of benchmarks arising from practical scenarios we demonstrate its effectiveness: we show that the formulation can solve large classification problems with tens or hundreds of thousands of examples and thousands of features, and to provide a tunable balance of accuracy vs. interpretability. Furthermore, we show that in many problems interpretability can be obtained at only a minor cost in accuracy.

The primary objective of the paper is to show that recent advances in the MaxSAT literature make it realistic to find optimal (or very high quality near-optimal) solutions to large-scale classification problems. We also hope to encourage researchers in both interpretable classification and in the constraint programming community to take it further and develop richer formulations, and bespoke solvers attuned to the problem of interpretable ML.

1 Introduction

The last decade has witnessed an unprecedented adoption of machine learning techniques to make sense of available data and make predictions to support decision making for a wide variety of applications ranging from health-care analytics to customer churn predictions, movie recommendations and macro-economic

The original version of this chapter was revised: There was a typing error in the family name of the first author. This has now been corrected. The correction to this chapter is available at https://doi.org/10.1007/978-3-319-98334-9_49

policy. The focus in the machine learning literature has been on increasingly sophisticated systems with the paramount goal of improving the accuracy of their predictions at the cost of making such systems essentially black-box. While in certain tasks such as ad predictions, accuracy is the main objective, in other domains, e.g., in legal, medical, and government, it is essential that the human decision makers who may not have been trained in machine learning can interpret and validate the predictions [17, 28].

The most popular interpretable techniques that tend to be adopted and trusted by decision makers include classification rules, decision trees, and decision lists [8, 10, 24, 25]. In particular, decision rules with a small number of Boolean clauses tend to be the most interpretable. Such models can be used both to learn interpretable models from the start, and also as proxies that provide post-hoc explanations to pre-trained black-box models [1, 12].

On the theoretical front, the problem of rule learning was shown to be computationally intractable [27]. Consequently, the earliest practical efforts such as decision list and decision tree approaches relied on a combination of heuristically chosen optimization objectives and greedy algorithmic techniques, and the size of the rule was controlled by either early stopping or ad-hoc rule pruning. Only recently there have been some formulations that attempt to balance the accuracy and the size of the rule in a principled optimization objective either through combinatorial optimization, linear programming (LP) relaxations, submodular optimization, or Bayesian methods [4, 7, 21, 22, 29] as we review in Sect. 5.

Motivated by the significant progress in the development of combinatorial solvers (in particular, MaxSAT), we ask: *can we design a combinatorial framework to efficiently construct interpretable classification rules that takes advantage of these recent advances?* The primary contribution of this paper is to present a combinatorial framework that enables a precise control of accuracy vs. interpretability, and to verify that the computational advances in the MaxSAT community can make it practical to solve large-scale classification problems.

In particular, this paper makes following contributions:

1. A MaxSAT-based framework, *MLIC*, that provably trades off accuracy vs. interpretability of the rules
2. A prototype implementation of *MLIC* based on MaxSAT that is capable of finding optimal (or high-quality near-optimal) classification rules from modern large-scale data-sets
3. We show that in many classification problems interpretability can be achieved at only a minor loss of accuracy, and furthermore, *MLIC*, which specifically looks for interpretable rules, can learn from much fewer samples than black-box ML techniques.

Furthermore, we hope to share our excitement with applications of constraint programming/MaxSAT in Machine Learning, and to encourage researchers in both interpretable classification and in the CSP/SAT communities to consider this topic further: both in developing new SAT-based formulations for interpretable ML, and in designing bespoke solvers attuned to the problem of interpretable ML.

The rest of the paper is organized as follows: We discuss notations and preliminaries in Sect. 2. We then present \mathcal{MLIC} , which is the primary contribution of this paper, in Sect. 3 and follow up with experimental setup and results over a large set of benchmarks in Sect. 4. We then discuss related work in Sect. 5 and finally conclude in Sect. 7.

2 Preliminaries

We use capital boldface letters such as \mathbf{X} to denote matrices while lower boldface letters \mathbf{y} are reserved for vectors/sets. For a matrix \mathbf{X} , \mathbf{X}_i represents i -th row of \mathbf{X} while for a vector/set \mathbf{y} , y_i represents i -th element of \mathbf{y} .

Let F be a Boolean formula and $\mathbf{b} = \{b_1, b_2, \dots, b_n\}$ be the set of variables appearing in F . A literal is a variable (b_i) or its complement ($\neg b_i$). A *satisfying assignment* or a *witness* of F is an assignment of variables in \mathbf{b} that makes F evaluate to *true*. If σ is an assignment of variables and $b_i \in \mathbf{b}$, we use $\sigma(b_i)$ to denote the value assigned to b_i in σ . F is in Conjunctive Normal Form (CNF) if $F := C_1 \wedge C_2 \cdots C_m$, where each clause C_i is represented as disjunction of literals. We use $|C_i|$ to denote the number of literals in C_i . For two vectors \mathbf{u} and \mathbf{v} over propositional variable/constants, we define $\mathbf{u} \vee \mathbf{v} = \bigvee_i (u_i \vee v_i)$, where u_i and v_i denote variables/constants at i -th index of \mathbf{u} and \mathbf{v} respectively. In this context, note that the operation \wedge between a variable and a constant follows standard interpretation, i.e. $0 \wedge b = 0$ and $1 \wedge b = b$.

We consider standard binary classification, where we are given a collection of training samples $\{\mathbf{X}_i, y_i\}$ where each vector $\mathbf{X}_i \in \mathcal{X}$ contains valuation of the features $\mathbf{x} = \{x^1, x^2, \dots, x^m\}$ for sample i , and $y_i \in \{0, 1\}$ is the binary label for sample i . A classifier \mathcal{R} is a mapping that takes in a feature vector \mathbf{x} and return a class y , i.e. $y = \mathcal{R}(\mathbf{x})$. The goal is not only to design \mathcal{R} to approximate our training set, but also to generalize to unseen samples arising from the same distribution. In this work, we restrict \mathbf{x} and y to be Boolean¹ and focus on classifiers that can be expressed compactly in Conjunctive Normal Form (CNF). We use C_i to denote the i th clause of \mathcal{R} . Furthermore, we use $|\mathcal{R}|$ to denote the sum of the counts of literals in all the clauses, i.e. $|\mathcal{R}| = \sum_i |C_i|$.

In this work, we focus on weighted variant of CNF wherein a weight function is defined over clauses. For a clause C_i and weight function $W(\cdot)$, we use $W(C_i)$ to denote the weight of clause C_i . We say that a clause C_i is hard if $W(C_i) = -\infty$, otherwise C_i is called as soft clause. To avoid notational clutter, we overload $W(\cdot)$ to denote the weight of an assignment or clause, depending on the context. We define weight of an assignment σ as the sum of weight of clauses that σ does not satisfy. Formally, $W(\sigma) = \sum_{i|\sigma \not\models C_i} W(C_i)$.

Given F and weight function $W(\cdot)$, the problem of MaxSAT is to find an assignment σ^* that has the maximum weight, i.e. $\sigma^* = \text{MaxSAT}(F, W)$ if $\forall \sigma \neq \sigma^*, W(\sigma^*) \geq W(\sigma)$. Our formulation will have negative clause weights, hence MaxSAT corresponds to satisfying as many clauses as possible, and picking the

¹ We discuss in Sect. 3 that such a restriction can be achieved without loss of generality.

weakest clauses among the unsatisfied ones. Note that the above formulation is different from the typical definition of MaxSAT but the difference is only syntactic. Borrowing terminology of community focused on developing MaxSAT solvers, we are solving a partial weighted MaxSAT instance wherein we mark all the clauses with $-\infty$ weight as hard and negate weight of all the other clauses and ask for a solution that optimizes the partial weighted MaxSAT formula. The knowledge of inner working of MaxSAT solvers and encoding of our representation into weighted MaxSAT is not required for this paper and we defer the details to release of source code post-publication.

3 *MLIC*: MaxSAT-Based Learning of Interpretable Classifiers

We now discuss the primary technical contribution of this paper, *MLIC*: MaxSAT-based Learning of Interpretable Classifiers. We first describe a metric for interpretability of CNF rules. Since our formulation employs binary features, we discuss how non-binary features such as categorical and continuous features can be represented as binary features. We then move on to formulate the problem of learning interpretable classification rules as a MaxSAT query and provide a proof of its theoretical soundness regarding controlling sparsity of the rules. As discussed in Sect. 5, prior work does not provide a sound procedure for controlling sparsity and accuracy. We then discuss the representational power of our CNF framework – in particular, we demonstrate that the proposed framework generalizes to handle complex objective function and rules in forms other than CNF.

3.1 Balancing Accuracy and Intrepretability

While in general interpretability may be hard to define precisely, in the context of decision rules, an effective proxy is merely the count of clauses or literals used in the rule. Rules involving few clauses with few literals are natural for humans to evaluate and understand, while complex rules involving hundreds of clauses will not be interpretable even if the individual clauses are. In addition to interpretability, such sparsity also controls model complexity and gives a handle of the generalization error.²

First, suppose that there exists a rule \mathcal{R} that perfectly classifies all the examples, i.e. $\forall i, y_i = \mathcal{R}(\mathbf{X}_i)$. Among all possible functions that satisfy this we would like to find the most interpretable (sparse) one:

$$\min_{\mathcal{R}} |\mathcal{R}| \quad \text{such that } \mathcal{R}(\mathbf{X}_i) = y_i, \quad \forall i$$

Since most ML datasets do not allow perfect classification, we introduce a penalty on classification errors. We balance the two terms by a parameter λ ,

² The framework proposed in this paper allows generalization to other forms of rules, as we discuss in Sect. 3.6.

where large λ gives more accurate but more complex rules, and smaller λ gives smaller rules at the cost of reduced accuracy. Let $\mathcal{E}_{\mathcal{R}}$ be the set of examples on which our classifier \mathcal{R} makes an error, then our objective is³:

$$\min_{\mathcal{R}} |\mathcal{R}| + \lambda |\mathcal{E}_{\mathcal{R}}| \quad \text{such that } \mathcal{R}(\mathbf{X}_i) = y_i, \quad \forall i \notin \mathcal{E}_{\mathcal{R}} \quad (1)$$

3.2 Discretization of Features

In our MaxSAT-based formulation, we focus on learning rules based on Boolean variables. We do also allow categorical and continuous features for our classifier, which are pre-processed before being presented to the MaxSAT-formulation. To handle categorical features one may use the common ‘one-hot’ encoding, where a Boolean vector variable is introduced with the cardinality equal to the number of categories. For example a categorical feature with values ‘red’, ‘green’, ‘blue’ would get converted to three binary variables, which take values 100, 010, and 001 for the three categorical values.

For continuous features, we introduce discretization, by comparing feature values to a collection of thresholds. The thresholds may be chosen for example based on quantiles of their distribution, or alternatively, on uniform partition of the range of feature values. Specifically, for a continuous feature x^c we consider a number of thresholds $\{\tau_k\}$ and define two separate Boolean features $I[x^c \geq \tau_k]$ and $I[x^c < \tau_k]$ for each τ_k . The number of thresholds may vary by feature. Thus, each continuous feature is represented using a collection of $2q$ Boolean features, where q is the number of thresholds.

In principle, one could use all the values occurring in the data as thresholds, and this would be equivalent to the original continuous features. In practice, however, such granularity is typically not necessary, and a handful of thresholds could be used, e.g., age-groups for each 5 years to discretize a continuous age variable. This typically leads to only a very minor (if any) loss in accuracy, and in fact improves the presentations and understanding of the rules to human users. In our experiments, we used 10 thresholds based on the quantiles of the feature distribution (10-th, 20-th, ... 100-th percentile), unless the number of unique values of the feature was less than 10, in which case we kept all of them.

We note that we could easily define arbitrary other Boolean functions of continuous or categorical variables within our framework. For example, categorical variables with many possible values (e.g. states or countries) may be grouped into more interpretable coarser units (regions or continents). Such groupings are application specific and would typically require relevant domain knowledge. They could perhaps be learned from data, but this is outside the scope of the current paper.

³ Cost-sensitive classification is defined analogously by allowing a separate parameter for false positives and false negatives.

3.3 Transformation to Max-SAT Query

We now describe our Max-SAT formulation for learning interpretable rules. \mathcal{MLIC} takes in four inputs: (i) a (0,1)-matrix \mathbf{X} of dimension $n \times m$ describing values of all m features for n samples with \mathbf{X}_i corresponding to feature vector $\mathbf{x} = \{x^1, x^2, \dots, x^m\}$ for sample i , (ii) (0,1)-vector \mathbf{y} containing class labels y_i for sample i , (iii) k , the desired number of clauses in CNF rule, (iv) the regularization parameter λ . Consequently, \mathcal{MLIC} constructs a MaxSAT query and invokes a MaxSAT solver to compute the underlying rule \mathcal{R} as we now describe.

The key idea of \mathcal{MLIC} is to define a MaxSAT query over $k \times m$ propositional variables, denoted by $\{b_1^1, b_1^2, \dots, b_1^m \dots b_k^m\}$, such that every truth assignment σ defines a k -clause CNF rule \mathcal{R} , where feature x^j appears in clause \mathcal{R}_i if $\sigma(b_i^j) = 1$. Corresponding to every sample i , we introduce a noise variable η_i that is employed to distinguish whether the labeling for sample i should be considered as noise or not. Let $\mathbf{B}_i = \{b_i^j \mid j \in [m]\}$.

The Max-SAT query constructed by \mathcal{MLIC} consists of the following three sets of constraints:

1. $N_i := (\neg\eta_i); \quad W(N_i) = -\lambda$
2. $V_i^j := (\neg b_i^j); \quad W(V_i^j) = -1$
3. $D_i := (\neg\eta_i \rightarrow (y_i \leftrightarrow \bigwedge_{l=1}^k (\mathbf{X}_i \vee \mathbf{B}_l))); W(D_i) = -\infty$

Please refer to Sect. 2 for the interpretation of $(\mathbf{X}_i \vee \mathbf{B}_j)$. Finally, the set of constraints Q^k constructed by \mathcal{MLIC} is defined as follows:

$$Q^k := \bigwedge_{i=1}^n N_i \wedge \bigwedge_{i=1, j=1}^{i=k, j=m} V_i^j \wedge \bigwedge_{i=1}^n D_i \quad (2)$$

Note that the elements of \mathbf{X}_i and y_i are not variables but constants whose values (0 or 1) are provided as inputs. Therefore, the set of variables for Q^k is $\{\eta_1, \eta_2, \dots, \eta_n, b_1^1, b_1^2, \dots, b_1^m \dots b_k^m\}$. We now explain the intuition behind the design of Q^k .

We assign a weight of $-\lambda$ to every N_i as we would like to satisfy as many N_i , i.e. falsify as many η_i as possible. Similarly, we assign a weight of -1 to every clause V_i^j as we are, again, interested in sparse solutions (i.e., ideally, we would prefer as many V_i^j to be satisfied as possible). Every clause D_i can be read as follows: if η_i is assigned to false, i.e. sample i is not considered as noise, then $\mathbf{y}_i = \mathcal{R}$. As noted in Sect. 2, equivalent representation of the $W(\cdot)$, as described above, for MaxSAT solvers involves usage of hard clauses.

Next, we extract \mathcal{R} from the solution of Q^k as follows.

Construction 1. Let $\sigma^* = \text{MaxSAT}(Q^k, W)$, then $x^j \in \mathcal{R}_i$ iff $\sigma^*(b_i^j) = 1$.

Before proceeding further, it is important to discuss CNF encodings for the above sets of constraints. The constraints arising from N_i and V_i are unit clauses and do not require further processing. Furthermore, note that \mathbf{y}_i is already known and is a constant. Therefore, when \mathbf{y}_i is 1, the constraint D_i can be

directly encoded as CNF by using equivalence of $(a \rightarrow b) \equiv (\neg a \vee b)$. Finally, when \mathbf{y}_i is 0, we use Tseitin encoding wherein we introduce an auxiliary variable z_i^j corresponding to each clause $(\mathbf{X}_i \vee \mathbf{B}_j)$. Formally, we replace $D_i := (\neg \eta_i \rightarrow (\bigvee_{j=1}^k \neg(\mathbf{X}_i \vee \mathbf{B}_j)))$ with $\bigwedge_{j=0}^k D_i^j$ where $D_i^0 := (\neg \eta_i \rightarrow \bigvee_j z_i^j)$, and $D_i^j := (z_i^j \rightarrow \neg(\mathbf{X}_i \vee \mathbf{B}_j))$. Furthermore, $W(D_i^j) = -\infty$. The following lemma establishes the theoretical soundness of parameter λ .

Lemma 1. *For all $\lambda_2 > \lambda_1 > 0$, if $\mathcal{R}_1 \leftarrow \mathcal{MLIC}(\mathbf{X}, \mathbf{y}, k, \lambda_1)$ and $\mathcal{R}_2 \leftarrow \mathcal{MLIC}(\mathbf{X}, \mathbf{y}, k, \lambda_2)$, then $|\mathcal{R}_1| \leq |\mathcal{R}_2|$ and $\mathcal{E}_{\mathcal{R}_1} \geq \mathcal{E}_{\mathcal{R}_2}$.*

Proof. First, note that construction of Q^k depends only on \mathbf{X} and \mathbf{y} . Furthermore, the parameter λ influences only the associated weight function. We denote weight functions corresponding to λ_1 and λ_2 as W_{λ_1} and W_{λ_2} respectively. Furthermore, let $\sigma_1 = \text{MaxSAT}(Q^k, W_{\lambda_1})$ and $\sigma_2 = \text{MaxSAT}(Q^k, W_{\lambda_2})$. If $\sigma_1 = \sigma_2$, the lemma trivially holds. We now complete proof by contradiction argument for the case when $\sigma_1 \neq \sigma_2$.

Let $|\mathcal{R}_1| > |\mathcal{R}_2|$. As $\sigma_1 \neq \sigma_2$, we have $W_{\lambda_2}(\sigma_1) \leq W_{\lambda_2}(\sigma_2)$. Since $W_\lambda(\sigma) = |\mathcal{R}| + \lambda \mathcal{E}_{\mathcal{R}}$, where \mathcal{R} is extracted from σ as stated above. Therefore, we have $\lambda_2(\mathcal{E}_{\mathcal{R}_2} - \mathcal{E}_{\mathcal{R}_1}) \geq |\mathcal{R}_1| - |\mathcal{R}_2|$. But we also have $W_{\lambda_1}(\sigma_1) \leq W_{\lambda_1}(\sigma_2)$, which implies that $\lambda_1(\mathcal{E}_{\mathcal{R}_2} - \mathcal{E}_{\mathcal{R}_1}) \leq |\mathcal{R}_1| - |\mathcal{R}_2|$. Since $\lambda_1 > \lambda_2$, we have contradiction. Therefore, it must be the case that $|\mathcal{R}_1| \leq |\mathcal{R}_2|$.

3.4 Illustrate Example

We illustrate our encoding with the help of a toy example. Let $n = 2, m = 3, k = 2$ and $X = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$ and $y = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. Then we have following clauses:

$$\begin{aligned} N_1 &:= (\neg \eta_1); & N_2 &:= (\neg \eta_2); \\ V_1^1 &= (-b_1^1); & V_1^2 &= (-b_1^2); & V_1^3 &= (-b_1^3); \\ V_2^1 &= (-b_2^1); & V_2^2 &= (-b_2^2); & V_2^3 &= (-b_2^3); \\ D_1 &:= (\neg \eta_1 \rightarrow (\neg(b_1^1 \vee b_1^3) \vee \neg(b_2^1 \vee b_2^3))); \\ D_2 &:= (\neg \eta_2 \rightarrow ((b_1^2 \vee b_1^3) \wedge (b_2^2 \vee b_2^3))) \end{aligned}$$

3.5 Beyond CNF Rules

While CNF formulas are general enough to express every Boolean formula, the length of representation may not be polynomial size. Therefore, one might wonder if we can extend \mathcal{MLIC} to learn rules in other canonical forms as well. In fact, early CSP based approaches to rule learning focused on rules in DNF form. We now show that with a minor change, we are able to learn rules expressible in DNF. Suppose that we are interested in learning a rule S that is expressible in DNF, such that $y = S(\mathbf{x})$, where S is a DNF formula. We note that

$(\mathbf{y} = S(\mathbf{x})) \leftrightarrow \neg(y = \neg S(\mathbf{x}))$. And if S is a DNF formula, then $\neg S$ is a CNF formula. Therefore, to learn rule S , we simply call \mathcal{MLIC} with $\neg \mathbf{y}$ as input and negate the learned rule.

3.6 Complex Objective Functions

We now discuss how \mathcal{MLIC} can be easily extended to handle complex objective functions. The objective function for \mathcal{MLIC} as defined in Eq. 1 treats all features equally. In some cases, the user might prefer rules that contain certain features. Such an extension is fairly easy to achieve as we need only to change the weight function corresponding to clauses V_i^j . Furthermore, in certain cases, one might want to minimize the total number of different features across different clauses rather than minimize the total number of terms. Such an extension is fairly easy to handle as we can simply replace $\bigwedge_{j=1}^k V_i^j$ with \hat{V}_i where $\hat{V}_i = (\bigvee_{j=1}^k b_i^j)$. It is worth noting that the proposed modifications impact only the MaxSAT query and does not require any modifications to the underlying MaxSAT solver. *We believe that such a separation is a key strength of \mathcal{MLIC} as it separates modeling and solving completely.*

4 Evaluation

To evaluate the performance of \mathcal{MLIC} , we implemented a prototype implementation in Python that employs MaxHS [13] to handle MaxSAT instances. We also experimented with LMHS [3], another state of the art MaxSAT solver and MaxHS outperformed LMHS for our benchmarks⁴. We conducted an extensive set of experiments on diverse publicly available benchmarks, seeking to answer the following questions⁵:

1. Do advancements in MaxSAT solving enable \mathcal{MLIC} to be run with datasets involving tens of thousands of variables with thousands of binary features?
2. How does the accuracy of \mathcal{MLIC} compare to that of state of the art but typically non-interpretable classifiers?
3. How does the accuracy of \mathcal{MLIC} vary with the size of training set?
4. How does the accuracy of \mathcal{MLIC} vary with λ ?
5. How does the size of learnt rules of \mathcal{MLIC} vary with λ ?

In summary, our experiments demonstrate that \mathcal{MLIC} can handle datasets involving tens of thousands of variables with thousands of binary features. Furthermore, \mathcal{MLIC} can generate rules that are not only interpretable but with accuracy comparable to that of other competitive classifiers, which often produce hard to interpret rules/models. We demonstrate that \mathcal{MLIC} is able to achieve sufficiently high accuracy with very few samples.

⁴ A detailed evaluation among different MaxSAT solvers is beyond the scope of this work and left for future work.

⁵ The source code of \mathcal{MLIC} and benchmarks can be viewed at <https://github.com/meelgroup/mlic>.

Table 1. Comparison of classification accuracy with 10-fold cross validation for different classifiers. For every cell in the last five columns, the top value represents the accuracy, while the value surrounded by parenthesis represent average training time.

Dataset	Size	# Features	RIPPER	Log Reg	NN	RF	SVC	\mathcal{MLIC}
Toms hardware	28170	830	0.968 (92.8)	0.976 (0.2)	0.977 (3.4)	0.976 (64.9)	Timeout	0.969 (2000)
Twitter	49990	1050	0.938 (187.3)	0.963 (0.2)	0.965 (6.8)	0.962 (250.9)	0.962 (1010.0)	0.958 (2000)
Adult-data	32560	262	0.852 (0.5)	0.801 (0.3)	0.866 (3.0)	0.844 (41.8)	Timeout	0.755 (2000)
Credit-card	30000	334	0.811 (0.7)	0.781 (0.1)	0.822 (3.9)	0.82 (25.5)	Timeout	0.82 (2000)
Ionosphere	350	564	0.886 (0.1)	0.909 (0.1)	0.926 (1.2)	0.909 (1.3)	0.886 (0.1)	0.889 (15.04)
PIMA	760	134	0.774 (0.1)	0.749 (0.1)	0.764 (1.3)	0.761 (1.3)	0.77 (21.4)	0.736 (2000)
Parkinsons	190	392	0.868 (0.1)	0.884 (0.1)	0.921 (1.2)	0.895 (1.1)	0.879 (1.6)	0.895 (245)
Trans	740	64	0.78 (0.0)	0.759 (0.0)	0.788 (1.2)	0.788 (1.2)	0.765 (372.3)	0.797 (1177)
WDBC	560	540	0.961 (0.1)	0.936 (0.0)	0.961 (1.3)	0.943 (1.4)	0.955 (3.0)	0.946 (911)

4.1 Experimental Methodology

We conducted extensive experiments on publicly available data sets obtained from UCI repository [6]. The data sets involved both real- and categorical-valued features. Specifically, the specific datasets are: buzz events from two different social networks: Twitter, Tom’s Hardware, Adult Data (adult_data), Credit Approval Data Set (credit_data), Ionosphere (Ionos), Pima Indians Diabetes (PIMA), Parkinsons, connectionist bench sonar (Sonar), blood transfusion service center (Trans), and breast cancer Wisconsin diagnostic (WDBC).

For purposes of comparison of the accuracy of \mathcal{MLIC} , we considered a variety of popular classifiers: ℓ_1 -penalized Logistic regression (LogReg), Nearest neighbors classifier (NN), and the black box random forests (RF), and support vector classification (SVC).

We perform 10-fold cross-validation to perform an assessment of accuracy on a validation set. We compute the mean across the 10 folds for each choice of a regularization (or complexity control) parameter for each technique (baseline and \mathcal{MLIC}), and report the best cross-validation accuracy. The number of parameter values is comparable (10) for each technique. For RF and RIPPER we use control based on the cutoff of the number of examples in the leaf node. For SVC and LogReg we discretize the regularization parameter on a logarithmic grid. In case of \mathcal{MLIC} we have 2 choices of $\lambda \in \{1, 10\}$ and number of clauses, $k \in \{1, 2, 3\}$ and the type of rule as {CNF, DNF}. We set the training time cutoff for each classifier (on each fold) to be 2000 seconds. *Again, note that some classifiers*

Table 2. Comparison of RIPPER vis-a-vis \mathcal{MLIC} in terms of the size of rules. **Note that despite using only a small number of literals, the proposed classifier, \mathcal{MLIC} mostly has better accuracy than RIPPER.**

Dataset	Size	# Features	RIPPER	\mathcal{MLIC}
Toms hardware	28170	830	57.5	4
Twitter	49990	1050	78.5	15
Adult-data	32560	262	74.5	51.5
Credit-card	30000	334	7.5	4
Ionosphere	350	564	3	5.5
PIMA	760	134	5	9
Parkinsons	190	392	6.5	6
Trans	740	64	6	4
WDBC	560	540	7.5	3.5

can be much faster than others, but in this paper we focus on the best tradeoff of accuracy vs interpretability in mission-critical settings, and the training time (which can be off-line) is secondary, as long as it is realistic. In this context, note that testing time for each of these techniques is less than 0.01 seconds for a given set of labels.

4.2 Illustrative Example

We illustrate the interpretable rules that are computed by \mathcal{MLIC} on the iris data set, which is a simple benchmark and widely used by machine learning community to illustrate new classification techniques. We consider the binary problem of classifying iris versicolor from the other two species, setosa and virginica. Of the four features, sepal length, sepal width, petal length, and petal width, we learn the following rule: $\mathcal{R}:=$

1. (sepal length $> 6.3 \vee$ sepal width $> 3.0 \vee$ petal width ≤ 1.5) \wedge
2. (sepal width $\leq 2.7 \vee$ petal length $> 4.0 \vee$ petal width > 1.2) \wedge
3. (petal length ≤ 5.0)

Let us pause a bit to understand how to apply the above rule. The above rule implies that when the three constraints are satisfied, the flower must be classified as Iris otherwise, non-iris. The size of the above rule, i.e. $|\mathcal{R}| = \sum_i |C_i| = 3 + 3 + 1 = 7$.

4.3 Results

Table 1 presents results of comparison of \mathcal{MLIC} vis-a-vis typical non-interpretable classifiers. The first three columns list the name, size (number of samples) and the number of binary features for each Dataset. The next five

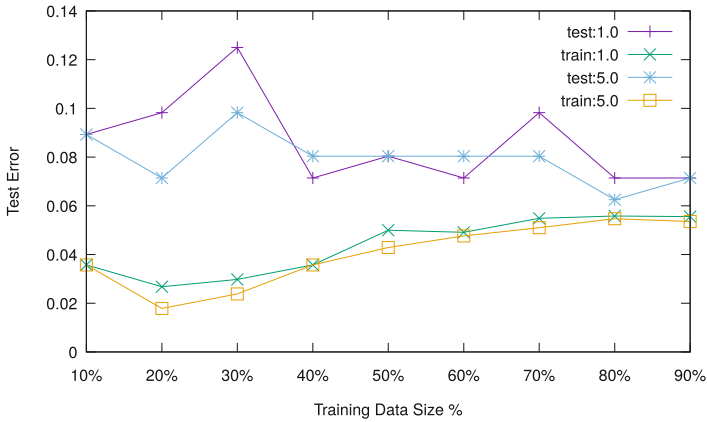


Fig. 1. Plot demonstrating behavior of training and test accuracy vs size of training data for WDBC.

columns present test accuracy of the classifiers RIPPER, Logistic Regression (Log Reg), Nearest Neighbor (NN), Random Forest (RF), and SVC. The final column contain the median test accuracy for *MLIC*. For every cell in the last five columns, the top value represents the accuracy, while the value surrounded by parenthesis represent average training time. We draw the following two conclusions from the table: First, *MLIC* is able to handle datasets with tens of thousands of examples with hundreds of features. The scalability of *MLIC* demonstrates the potential presented by remarkable progress in SAT solving. Recent research efforts have often used NP-hardness of the problem to justify the usage of heuristics but our experience with *MLIC* shows that SAT solving is able to solve many large-scale problems directly. Note that when MaxHS times out, it is able to provide the best solution found so far. In this context, it is worth noting that for some of the benchmarks, even state of the art classifiers such as SVC time out. Secondly, *MLIC* is often able to achieve accuracy that is sufficiently close to accuracy achieved by typical non-interpretable classifiers but produces easy to state rules that often have just a few literals.

To demonstrate *MLIC*'s ability to compute easy to state rules in comparison to the state of the art classifiers such as RIPPER, we computed the size of rules returned by RIPPER and *MLIC*. Table 2 presents results of comparison of *MLIC* vis-a-vis RIPPER. The first three columns list the name, size (number of samples) and the number of binary features for each Dataset. The next two columns state the median size of rules returned by RIPPER and *MLIC*. The size of a rule is computed as the number of terms involved in a rule. First, note that except for two cases where RIPPER has produced marginally shorter rules compared to *MLIC*, *MLIC* produces significantly shorter rules and sometimes, these rules could be orders of magnitude larger than those produced by *MLIC*. For example, for Toms hardware, the rule produced by RIPPER has 57 terms compared to just 4 literals for *MLIC*. Note that with *MLIC* has better accuracy than RIPPER. One might wonder if the rule learned by RIPPER could have been

simply transformed into a sparser rule; it is not the case here. Furthermore, it is worth noting that RIPPER does not provide sound handle to tune rule size and therefore, user is left to trying out combination of input parameters without any guarantee of improvement of the interpretability of generated rules, which we experienced in this case. A in-depth study into failure of RIPPER to generate sparser rules than *MLIC* is beyond the scope of this work.

To measure the accuracy of *MLIC* w.r.t. the size of training data, we consider test errors when only a fraction of training data is available (we vary it from 10 % to 90 % in steps of 10 %). Due to lack of space, we present result for only one benchmark, WDBC, for $\lambda = 1$ and 5 and $k = 1$ in Fig. 1. We plot median training and test accuracy of *MLIC* over 10 trials, which is also known as learning curve in machine learning literature. The y-axis represents the error as the ratio of incorrect predictions to total examples while the x-axis represents the size of training set. The plot shows how training and test error vary for $\lambda = 1$ and 5. Note that *MLIC* is able to achieve sufficiently high test accuracy with just 40% of the complete dataset. We observe similar behavior for other benchmarks as well.

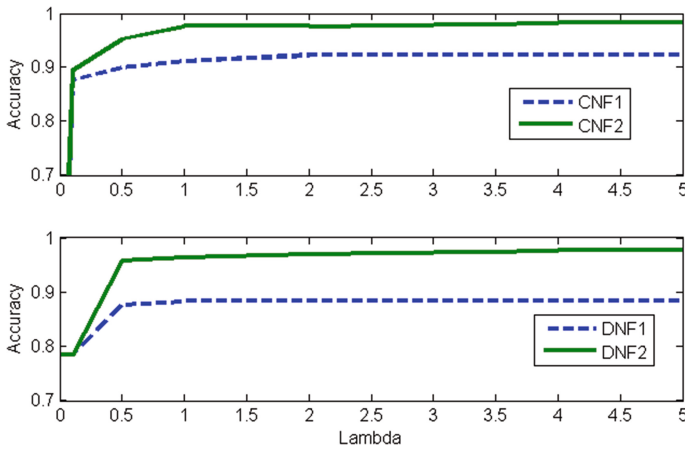


Fig. 2. Plot demonstrating monotone behavior of training accuracy vs λ for CNF and DNF rules with $k = 1$ and 2.

Figures 2 and 3 illustrate how training accuracy and rule sizes vary with λ for one of the representative benchmark, parkinsons. CNF1, CNF2, DNF1, DNF2 refer to invocations of *MLIC* with (rule type, k) set to (CNF, 1), (CNF, 2), (DNF, 1), and (DNF, 2) respectively. For each of the plots, x-axis refers to the value of λ while y-axis represents Rule size (i.e. $|\mathcal{R}|$) and accuracy for Figs. 3 and 2 respectively. First, note that for both CNF and DNF, the accuracy of rules is generally higher for larger k . Significantly, the plots clearly demonstrate monotonicity of rule size and accuracy with respect to λ . In contrast, the state of the art interpretable classifier, RIPPER, can lead to rules that can be order

of magnitude larger than those produced by \mathcal{MLIC} . For example, for Toms hardware, the rule produced by RIPPER has 57 terms compared to just 4 literals for \mathcal{MLIC} . In this context, it is worth noting that RIPPER does not provide sound handle to tune rule size and therefore, user is left to trying out combination of input parameters without any guarantee of improvement of the interpretability of generated rules.

5 Related Work

There is a long history of learning interpretable classification models from data, including popular approaches such as decision trees [5, 24], decision lists [25], and classification rules [10]. While the form of such classifiers is highly amenable to human interpretation, unfortunately, most of the objective functions that arise for these problems are intractable combinatorial optimization problems. Hence, most popular existing approaches rely on various greedy heuristics, pruning, and ad-hoc local criteria such as maximizing information gain, coverage, e.t.c. For example various popular decision rule approaches, such as C4.5.rules [24], CN2 [9], RIPPER [10], SLIPPER [11], all make different trade-offs in how they use these heuristic criteria for growing and pruning the rules.

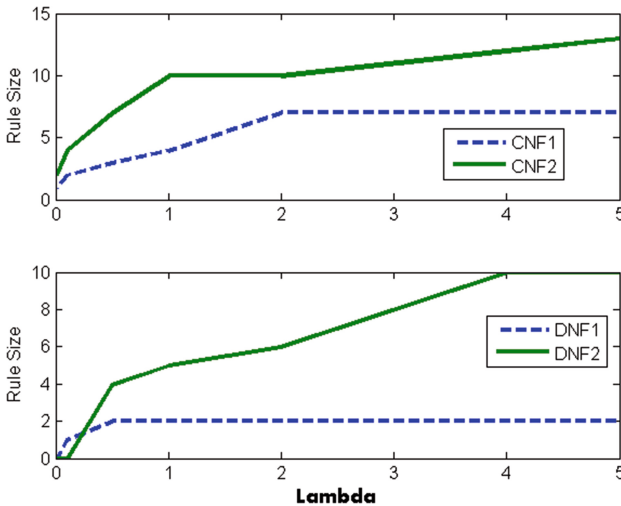


Fig. 3. Plot demonstrating behavior of rule size vs λ

Recent advances in large-scale optimization and scalable Bayesian inference gave rise to state-of-the-art black box models. However, many of the same advances can also be used in the context of interpretable machine learning models. Some of such recent proposals include Bayesian approaches [20, 30], constraint programming [2], integer programming approaches to learn decision trees [4], quadratic programming relaxation with a variance-penalized margin

objective [26]. Greedy approaches are used with a principled objective function in ENDER [15] and Set covering machines [22]. [19] propose a hierarchical kernel learning approach and [18] use optimization to combine basic Boolean clauses obtained from decision trees. Linear Programming relaxations based on Boolean Compressed Sensing formulation have been used to learn sparse interpretable rules and checklists⁶ in [16, 21]. Prior work has considered applications of constraint programming to learning Bayesian networks [2] and itemset mining [14, 23]. In contrast, we focus on learning sparse interpretable classification rules allowing control of accuracy vs. interpretability.

6 Extensions

In the paper, we have focused on decision rules in the DNF or CNF form, which is among the most interpretable classification methods available. We now describe a few related classification formulations, which are also amenable to being learned from data using a SAT-based framework. A simple AND-clause can be considered as a requirement that all of the N literals in the clause are satisfied, while a simple OR-clause requires that at least 1 of the N literals are satisfied. A useful generalization is a “K-of-N” clause [12], which is true when at least K of the N literals are satisfied. In particular, it leads to a very popular decision rubric called *checklists* or *scorecards*, widely used in medicine and finance, where a questionnaire asks some questions (e.g., risk factors), and the total number of positive answers is compared to a pre-determined threshold. LP relaxations have been considered for learning scorecards from data [16], and our MaxSAT-based framework can be directly extended. In the case of multi-class classification, a decision rule may be ambiguous, as it does not specify what multi-class label to use when several contradictory clauses pointing to different labels are satisfied simultaneously. Decision lists [25] enforces an order of evaluation of the rules, resolving this ambiguity. Bayesian frameworks for learning decision lists have been considered recently [20]. Perhaps the most well known interpretable classification scheme is a decision tree, where literals are arranged as nodes in a binary tree, and a decision is made by following the path from the root node to one of the leafs. The decision tree can be converted to an equivalent set of classification rules which correspond to all the paths from the root to the leafs, a more expensive representation. On the other side, however, certain small decision rules can lead to very complex decision trees, for example, the “K-of-N” rule cannot be efficiently encoded using a decision tree. Recent work has considered combinatorial optimization to learn compact interpretable decision trees [4]. Beyond simple Boolean expressions, a variety of weighted classification methods can be used, for example, a weighted linear combination of simple AND clauses – for instance by using Boosting on a set of classifiers based on simple logical clauses. In future work, we plan to extend our MaxSAT-based framework for all these related interpretable classification approaches.

⁶ Note, however, that the objective functions for the integer program and the LP relaxation in these papers are not the same as sparsity-penalized cost-sensitive classification error.

7 Conclusion

We proposed a new approach to learn interpretable classification rules via reduction to (MaxSAT). Due to the impressive advances in MaxSAT-solving, our formulation can find optimal or near-optimal rules balancing accuracy and interpretability (sparsity) for large data-sets involving tens or hundreds of thousands of data points, and hundreds or thousands of features. Furthermore, the approach separates the modeling from the optimization, and this framework could be used to solve a wide variety of interpretable classification formulations, including decision lists, decision trees, and decision rules with different cost functions (including group-sparsity, sharing of the variables, and having prior knowledge on variable importance). Finally, we demonstrate on experiments that for many classification problems interpretability does not have to come at a high cost in terms of accuracy.

Furthermore, we hope to share our excitement with applications of constraint programming/MaxSAT in Machine Learning, and to encourage researchers in both interpretable classification and in the CSP/SAT communities to consider this topic further: both in developing new SAT-based formulations for interpretable ML, and in designing bespoke solvers attuned to the problem of interpretable ML.

Acknowledgements. This work was supported in part by NUS ODPRT Grant, R-252-000-685-133 and IBM PhD Fellowship. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore, <https://www.nscg.sg>.

References

1. Andrews, R., Diederich, J., Tickle, A.: Survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowl. Based Syst.* **8**(6), 373–389 (1995)
2. van Beek, P., Hoffmann, H.F.: Machine learning of Bayesian networks using constraint programming. In: *Proceedings of CP*, pp. 429–445 (2015)
3. Berg, J., Saikko, P., Järvisalo, M.: Improving the effectiveness of sat-based preprocessing for MaxSAT. In: *Proceedings of IJCAI* (2015)
4. Bertsimas, D., Chang, A., Rudin, C.: An integer optimization approach to associative classification. *Adv. Neur. Inf. Process. Syst.* **25**, 269–277 (2012)
5. Bessiere, C., Hebrard, E., O’Sullivan, B.: Minimising decision tree size as combinatorial optimisation. In: Gent, I.P. (ed.) *CP 2009*. LNCS, vol. 5732, pp. 173–187. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_16
6. Blake, C., Merz, C.J.: {UCI} repository of machine learning databases (1998)
7. Boros, E., Hammer, P., Ibaraki, T., Kogan, A., Mayoraz, E., Muchnik, I.: An implementation of logical analysis of data. *IEEE Trans. Knowl. Data Eng.* **12**(2), 292–306 (2000)
8. Breiman, L., Friedman, J., Stone, C., Olshen, R.: *Classification and Regression Trees*. CRC Press, Boca Raton (1984)
9. Clark, P., Niblett, T.: The CN2 induction algorithm. *Mach. Learn.* **3**(4), 261–283 (1989)

10. Cohen, W.W.: Fast effective rule induction. In: Proceedings of International Conference on Machine Learning, pp. 115–123. Tahoe City, CA, July 1995
11. Cohen, W.W., Singer, Y.: A simple, fast, and effective rule learner. In: Proceedings of National Conference on Artificial Intelligence, pp. 335–342, Orlando, FL. July 1999
12. Craven, M.W., Shavlik, J.W.: Extracting tree-structured representations of trained networks. In: Proceedings of NIPS, pp. 24–30 (1996)
13. Davies, J., Bacchus, F.: Solving MaxSAT by solving a sequence of simpler sat instances. In: Proceedings of CP, pp. 225–239 (2011)
14. De Raedt, L., Guns, T., Nijssen, S.: Constraint programming for itemset mining. In: Proceedings of KDD, pp. 204–212 (2008)
15. Dembczyński, K., Kotłowski, W., Słowiński, R.: Ender: a statistical framework for boosting decision rules. *Data Mining Knowl. Discov.* **21**(1), 52–90 (2010)
16. Emad, A., Varshney, K.R., Malioutov, D.M.: A semiquantitative group testing approach for learning interpretable clinical prediction rules. In: Proceedings of Signal Process. Adapt. Sparse Struct. Repr. Workshop, Cambridge, UK (2015)
17. Freitas, A.: Comprehensible classification models: a position paper. *ACM SIGKDD Explor. Newsl.* **15**(1), 1–10 (2014)
18. Friedman, J.H., Popescu, B.E.: Predictive learning via rule ensembles. *Ann. Appl. Stat.* **2**(3), 916–954 (2008)
19. Jawanpuria, P., Jagarlapudi, S.N., Ramakrishnan, G.: Efficient rule ensemble learning using hierarchical kernels. In: Proceedings of ICML (2011)
20. Letham, B., Rudin, C., McCormick, T.H., Madigan, D.: Building interpretable classifiers with rules using Bayesian analysis. Technical report 609, Department of Statistics. University of Washington, December 2012
21. Malioutov, D.M., Varshney, K.R.: Exact rule learning via Boolean compressed sensing. In: Proceedings of ICML, pp. 765–773 (2013)
22. Marchand, M., Shawe-Taylor, J.: The set covering machine. *J. Mach. Learn. Res.* **3**(Dec), 723–746 (2002)
23. Nijssen, S., Guns, T., De Raedt, L.: Correlated itemset mining in ROC space: a constraint programming approach. In: KDD, pp. 647–656. ACM (2009)
24. Quinlan, J.R.: *C4.5: Programming for Machine Learning*, p. 38. Morgan Kaufmann, San Francisco (1993)
25. Rivest, R.L.: Learning decision lists. *Mach. Learn.* **2**(3), 229–246 (1987)
26. Rückert, U., Kramer, S.: Margin-based first-order rule learning. *Mach. Learn.* **70**(2–3), 189–206 (2008)
27. Valiant, L.G.: Learning disjunctions of conjunctions. In: Proceedings of International Joint Conference on Artificial Intelligence, pp. 560–566. Los Angeles, CA, August 1985
28. Varshney, K.R.: Data science of the people, for the people, by the people: a viewpoint on an emerging dichotomy. In: Proceedings of Data for Good Exchange Conference (2015)
29. Wang, T., Rudin, C., Doshi-Velez, F., Liu, Y., Klampfl, E., MacNeille, P.: Or’s of And’s for interpretable classification, with application to context-aware recommender systems. arXiv preprint [arXiv:1504.07614](https://arxiv.org/abs/1504.07614) (2015)
30. Wang, T., Rudin, C., Liu, Y., Klampfl, E., MacNeille, P.: Bayesian Or’s of And’s for interpretable classification with application to context aware recommender systems (2015)



Objective as a Feature for Robust Search Strategies

Anthony Palmieri^{1,2} and Guillaume Perez³(✉)

¹ Huawei Technologies Ltd., French Research Center, Paris, France
anthony.palmieri@huawei.com

² Université de Caen - Normandie, GREYC, Caen, France

³ Department of Computer Science, Cornell University, Ithaca, NY 14850, USA
guillaume.perez06@gmail.com

Abstract. In constraint programming the search strategy entirely guides the solving process, and drastically affects the running time for solving particular problem instances. Many features have been defined so far for the design of efficient and robust search strategies, such as variables' domains, constraint graph, or even the constraints triggering fails. In this paper, we propose to use the objective functions of constraint optimization problems as a feature to guide search strategies. We define an objective-based function, to monitor the objective bounds modifications and to extract information. This function is the main feature to design a new variable selection heuristic, whose results validate human intuitions about the objective modifications. Finally, we introduce a simple but efficient combination of features, to incorporate the objective in the state-of-the-art search strategies. We illustrate this new method by testing it on several classic optimization problems, showing that the new feature often yields to a better running time and finds better solutions in the given time.

1 Introduction

Solving combinatorial optimization problems is known to be a hard task, but constraint programming (CP) enables tackling several of them [22, 26]. One of the CP strength leans on an efficient search for a solution in the variables' domain space. The resolution of industrial problems often relies on dedicated knowledge experts to build a good search strategy (SS) [23, 25]. But such information, while appealing, is not always available nor possible. That is one of the main motivations for the development of black-box constraint solvers, where the only user's concern is to build an efficient model. Black-box solvers need robust and efficient SSs, and many researches have been done [5, 11, 16, 18, 28]. Notably, in Constraint Programming, activity-based search (ABS) [14], impact-based search (IBS) [20] and weighted degrees (Wdeg) [1] are well known state-of-the-art search strategies for combinatorial problems.

In CP a search heuristic usually consists of choosing a pair (*variable, value*), called a decision. Then, a binary search tree is built to explore the search space.

The solving time is highly correlated with the size of the search tree. A shorter run time is usually expected when a smaller tree is explored. Since the search strategy determines how to build the search tree, the solving time is strongly impacted by the search strategy, and can differ by order of magnitude.

Most search strategies use the first fail principle [7, 24]. This principle tries to fail as soon as possible, in order to reduce the search tree size. The first fail principle is very efficient in practice for constraint satisfaction problems (CSPs), for example SSs such as IBS and ABS consider the variables' domains as feature to make decisions, and try to find variables having the potential to reduce the other variables domains, while WDeg uses fail counters and the constraint graph.

A constraint optimization problem (COP) can be seen as a CSP with an objective function to optimize. Solvers often have a variable representing the possible values of the objective function. When a new solution is found, a constraint is added, requiring the next one to be better. Once the best solution is found, the next step is to prove its optimality. In other words, solving a COP includes: finding the best solution, and proving that no better solution exists. In practice, it is unknown whether the current solution is the best one until the exploration of the search tree is completed.

Search strategies are mostly designed to reduce the search space by focusing on constraint satisfaction, but with COPs, the objective value can additionally be used to reduce the search space. Two different solutions might prune the search space, depending on their respective objective values. In COP, finding a good solution can drastically reduce the search space, by avoiding the exploration of less promising parts of the search tree, with respect to the objective. This implies that the order in which solutions are found has a strong impact on run-time for the complete space exploration, unlike for CSPs.

This observation is one of the main motivations of this paper. Our idea, inspired by integer programming [4], is to extract good features from the objective variable to make good decisions. A recent CP work started exploring this area by using the objective to design a value selector in order to find a first good solution [3]. An advantage of using the objective as a feature is its ability to both optimize the objective value and to reduce the search tree at the same time. Such information, as will be shown in the experimental section, can drastically help SSs to make better decisions.

In the following, we design a function (Δ_O) monitoring the objective bounds modifications along the solving process. This function is one possible implementation of an objective-based feature extractor ($\widetilde{\Delta}_O$). We then define a variable selector based only on $\widetilde{\Delta}_O$, named Objective-Based Selector (OBS), which selects the variable maximizing $\widetilde{\Delta}_O$. Lastly, in order to take advantage of this new objective feature and the many existing ones, we propose a simple but efficient hybrid method to combine search strategies, such as IBS, ABS etc., to take into account the newly introduced objective feature. Finally, we show the efficacy of these new hybrid strategies compared to the original strategies on all the optimization problems from the Minizinc challenge library [15].

2 Preliminaries

2.1 Constraint Satisfaction Problem (CSP)

A *CSP* is a pair $P = (X, C)$ where $X = \{x_1, x_2, \dots, x_n\}$ is a set of variables and $C = \{C_1, C_2, \dots, C_m\}$ is a set of constraints. A variable x_i is associated with a domain $D(x_i)$, representing all of its possible values. A constraint C_i contains a set of all its allowed tuples defined over a subset $S_{C_i} \subseteq X$ of variables.

A solution is a tuple of values (a_1, a_2, \dots, a_n) such that the assignments $x_1 = a_1, x_2 = a_2 \dots, x_n = a_n$ respect all the constraints. The solving process for a CSP generally involves a depth first search algorithm with backtracks in a decision tree. At each node of the tree, a propagation algorithm is run, which iteratively uses a dedicated filtering algorithm to check the validity of each constraint. Each filtering algorithm reduces the search space by removing the values that cannot belong to a solution. Finally, a solution is found when all variables are instantiated to a value.

A constraint optimization problem (*COP*) is a pair (P, F_O) , where P is a *CSP* and F_O is an objective function that has to be optimized. Without loss of generality, we consider here only minimization problems. All solutions to a *COP* are not equivalent, as their overall quality is determined by the objective value $F_O(sol)$. The solving process of a *COP* is analogous to a *CSP*, except that it contains an objective constraint. This constraint ensures that the next solution found will be better. This paper aims to use this constraint in order to reduce the search space.

2.2 Search Strategies

A search strategy (SS) for constraint programming (CP) determines how the search tree is built during the solving process. At each node of the search tree, the SS chooses a non-assigned variable and a value belonging to its domain. A decision often corresponds to a pair (*variable, value*) which can be seen as a backtrackable constraint *variable = value*. Search strategies are crucially important to find good solutions, to reduce the search space, and even to quickly find an initial feasible or good solution [3].

We briefly describe three state-of-the-art SSs. For a more complete description please refer to their original publications.

Impact Based Search (IBS) [20] selects the variable whose choice is expected to provide the largest search space reduction. To do so, *IBS* considers the cardinality reduction of the Cartesian product of the domains (called the impact). Thus the main feature of this SS uses variables' domains.

More formally, let x be a variable, and v be a value belonging to the current domain $D(x)$. Let P_{before} (resp. P_{after}) be the cardinality of the Cartesian product of the domains before (resp. after) the application of the decision $x = v$. The impact of a decision is:

$$I(x = v) = 1 - \frac{P_{after}}{P_{before}}$$

Let $\bar{I}(x = v)$ be the average impact of the decision $x = v$. Then, this impact of a variable x with current domain $D(x)$ is computed by the following formula:

$$\bar{I}_x = \sum_{v \in D_x} 1 - \bar{I}(x = v)$$

At each node the free variable having the largest impact is assigned to its value having the smallest impact. Note that this search is an adaptation of pseudo-cost-based search from mixed integer programming.

Activity Based Search (ABS) [14] selects the most active variable per domain value. A variable's activity is measured by counting how often its domain is reduced during the search. Thus, once again, the feature of this SS uses the domains of the variables. More formally, the number of modified variables is monitored and stored in $A(x)$, which is updated after each decision with the following rule:

$$\begin{aligned} \forall x \in X_{s.t.} |D(x)| > 1 : A(x) &= A(x) \times \gamma \\ \forall x \in X_0 : A(x) &= A(x) + 1 \end{aligned}$$

X_0 represents the set of variables reduced by the decision and $\gamma \in [0, 1]$ is the decay parameter. *ABS* maintains an exponential moving average of activities by variables' value. At each node, *ABS* selects the variable with the highest activity and the value with the least activity.

Weighted Degree (WDeg) [1] uses the constraint graph to make decisions. *WDeg* counts the number of failures ω_c for each constraint c . *WDeg* features are the constraint graph and the fail counters. *WDeg* first computes, for each variable x , the value $wdeg(x)$, which is the weighted (ω) sum of the constraints involving at least one non-assigned variable. *WDeg* then, selects the variable having the highest ratio $\frac{|D(x)|}{wdeg(x)}$.

3 Objective Function and Search Strategy

Search strategies aim to reduce the search space, but additionally aim to find good solutions as quickly as possible. Most SSs choose the hardest variables to satisfy first, the main challenge being to find such variables. While most SSs decisions were based on variables domains, the constraint graph, etc., objective-value based decisions are rarely done in CP. One of the reasons is that, in CP, we cannot easily back-propagate the objective to the variables to make decisions as done in Mixed Integer Programming. But even if we can not have such exact information, not taking into account the variables impacting the objective value can lead to an exponential loss in time. This is shown by the following synthetic example.

Example. Consider a COP having $n + m$ variables and whose objective is the sum of the last m variables. This problem has an AllDifferent constraint [21] over all the variables. Ignoring the objective value can lead to the search tree shown in Fig. 1 (left). In this example, the strategy focuses only on other features, without taking the objective into account. Whereas a strategy that considers the objective detects variables having high impact on the objective, and consider them earlier enabling a potential reduction of the search tree.

Moreover, we can find high quality solutions earlier and these solutions prune the search space more efficiently. As we can see, the processing of the m variables is repeated an exponential number of times (d^n). This is because the variables that impact the objective are chosen too late leading to a bigger search tree.

The search tree using the objective value as a feature is shown in Fig. 1 (right). The last m variables are selected higher in the search tree, yielding better solutions faster and allowing to close the search using the objective sooner. Finally, by using the objective value, we obtain a smaller search tree.

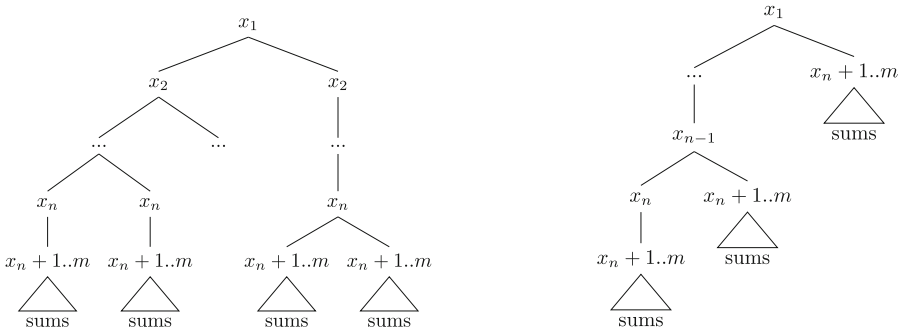


Fig. 1. (left) A search failing to consider the objective value. (right) An objective based search.

This simple example shows that the objective value allows to assign variables having high influence on the objective earlier and thus can help the solver to avoid considering useless parts of the tree. The idea is to consider as soon as possible the variables impacting the objective. We now define a new feature based on the objective, which we will use to define an objective-based search strategy.

3.1 Objective Modifications as a Feature

The proposed feature focuses on the objective bounds modifications by using a function Δ_O . The upper and lower bounds are separately considered as two different pieces of information. Let O be the objective variable to optimize. Let s and $s - 1$ be respectively the current and the previous node of the search tree. Let $\overline{\Delta}_O$ (resp. $\underline{\Delta}_O$) be the upper (resp. lower) bounds difference between its value before and after the decision propagation. The function is defined as follows:

$$\Delta_O(s) = a \times \underline{\Delta}_O + b \times \overline{\Delta}_O$$

We choose to consider the upper and lower bounds separately. The choice of the parameters a and b defines the function behavior. The coefficients can take any value and correspond to the importance (positive or negative) given to each bound. For instance, in minimization problems, the coefficient a of lower bound modification corresponds to the weight for the consideration of removing the best potential solutions. While, the upper bound modification coefficient b , represents the weight to consider the deletion of the worst potential solutions.

Note that this function has a more fine-grained description of the objective than usual measures used in search strategies. Classic SSs monitor the modifications of the decision variables, but in general, treating differently the lower and upper bounds, has no meaning for such variables.

3.2 Objective-Based Selector (OBS)

We propose a new variable selector based on the Δ_O function: *OBS*. *OBS* first selects the variables having the highest impact on the objective with regard to the Δ_O function. To do so, the weighted sum of the Δ_O function values for each $x \in X$ is monitored through $\widetilde{\Delta}_O(x)$, and updated after each decision involving the variable x . The parameter γ is the degree of weighting decrease of the exponential moving average. The updated value $\widetilde{\Delta}_O'(x)$ is processed as follow:

$$\widetilde{\Delta}_O'(x) = \frac{\widetilde{\Delta}_O(x) * (1 - \gamma) + \gamma * \Delta_O(x)}{\gamma}$$

At each decision, *OBS* selects the variable $x \in X$ such that $\forall y \in X, \widetilde{\Delta}_O(x) \geq \widetilde{\Delta}_O(y)$.

Example. Consider the didactic COP defined by the variables (x_1, x_2, x_3, x_4) having each as domain $D = [1, 4]$ and an *AllDifferent* constraint on the 4 variables. The COP's objective is $\min x_3 + x_4$. We use the parameters $(a = -1, b = 1)$ for the Δ_O function, in order to penalize lower bound modifications and reward upper bound modification.

The tree search from Fig. 2 shows the application of the objective based search strategy versus a lexicographic search. In this example, when a variable is selected, it is assigned to its minimum value. The lexicographic search on the left has more decisions than *OBS* on the right because it cannot identify which variable are important to satisfy the constraints and improve the objective.

At the beginning of the exploration, in the right tree showing *OBS* search, the variables x_1, x_2 and x_3 are selected and set to their minimum values. Each of these assignments has an effect on the objective's bounds and thus modifies Δ_O . When the decision $x_1 = 1$ is propagated, $\Delta_O(x_1)$ is set to -2 because of the changes of the objective domain from $[2, 8]$ to $[4, 8]$. The propagation of $x_2 = 2$ reduces the objective's domain from $[4, 8]$ to $[6, 8]$ implying $\Delta_O(x_2) = -2$. When the variable x_3 is selected, the objective is instantiated to 7. This implies

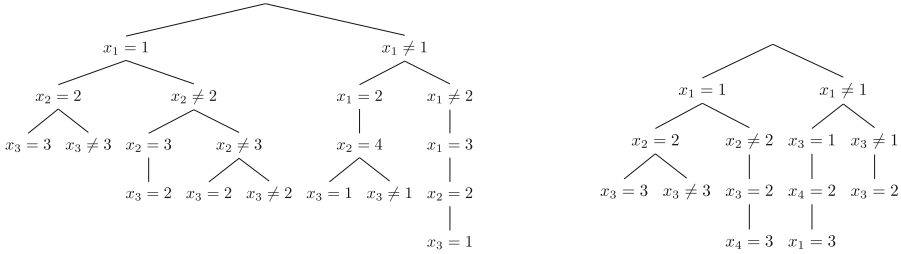


Fig. 2. Comparison of the search tree by a lexicographic search (left) and an objective based search (right)

a $\Delta_O(x_3) = 0$. A solution is found with the value 7, so the next solution has to be smaller than 7. Afterwards the decision $x_3 = 3$ is refuted and the search tree is backtracked to the decision $x_2 = 2$ which is refuted, implying $x_2 \neq 2$. Then x_3 which has the highest Δ_O value is selected and instantiated to its domain's minimal value: 2. Then x_4 is the next free variable with the highest Δ_O value, 0. We thus select x_4 and assign it to its smallest value, 3. We find a solution equal to 5. Finally, when this branch is closed, the decision $x_1 = 1$ is refuted and by applying the *OBS* selection, the branch leading to the best solution is explored. An important aspect of this search is that it is close to human intuition to choose first variables x_3 and x_4 since they belong to the objective.

Note that the maintenance of $\widetilde{\Delta}_O$ values and the selection process are simple and not intrusive in solvers. Moreover, *OBS* does not need to change the constraints implementations.

3.3 Hybridization of Search Strategies

In this section we show how the objective and classical features can be combined together, to benefit from both. But most strategies should not be directly combined due to the range differences of their feature. For example, the *IBS* strategy has a value range between $[0, 1]$, while *ABS* one is between $[0, n]$. We propose to normalize all these values to fit in the interval $[0, 1]$ in order to combine them. Note that this applies to the Δ_O function as well.

Let $\widetilde{S}^n(x)$ be the normalized value of a search strategy S based on a classical feature. And let $\widetilde{\Delta}_O^n(x)$ be the normalized values for *OBS*. We combine the two pieces of information with the following formula:

$$S_O(x) = \alpha * \widetilde{S}^n(x) + (1 - \alpha) * \widetilde{\Delta}_O^n(x)$$

The hybrid search strategy selects the variable maximizing S_O . The values α and $(1 - \alpha)$ represent the importance given to each feature. Note that α is in $[0, 1]$.

Example with ABS_O : While the ABS strategy uses the \widetilde{A} values, storing the activities involved by the variables, our modification of the value associated with each variable is the sum:

$$ABS_O(x) = \alpha * \widetilde{A}^n(x) + (1 - \alpha) * \widetilde{\Delta}_O^n(x)$$

This $ABS_O(x)$ value contains both pieces of information: the activity and the objective modifications.

Remarks: The hybridization of many others strategies is as simple as for ABS . For the following sections, we respectively denote the hybridized versions of ABS , IBS and $WDeg$ by ABS_O , IBS_O and $WDeg_O$.

4 Experiments

4.1 The Experimental Setting

Configurations. All experiments were done on a Dell machine having four Intel E7-4870 Intel processors and 256 GB of memory, running Scientific Linux. We implemented these new strategies in the Choco 4 CP solver [19]. The code can be found on our GitHub¹. Each run used a time limit of 30 min. The strategies were warmed up with a diving step, using up to 1000 restarts, or by ensuring a certain number of decisions. The same warm up (method and seed) was used for all the methods, in order to avoid any bias.

Benchmarks. The experimental evaluation used on the MiniZinc Benchmark library [15], with benchmarks that have been widely studied, often by different communities, including *template design*, *still life*, *RCPSP*, *golomb ruler*, etc. Many problem specifications can be found in [6]. Every class of optimization problems from the MiniZinc library has been considered. Since the number of instances per family is huge, and has a large variance between families, we have randomly selected up to 10 instances per family. Such subset selection preserves the diversity of instances, and do not favor a specific kind of family in plots. The problems have been translated into the FlatZinc format, using the MiniZinc global constraints library provided by Choco-solver, which preserves the global constraints.

Plots. The scatters and curves presented in this section are in log scale. A scatter plot shows the comparison of two strategies instance by instance. The diagonal separates the instances where each method has performed better than the other. The points above (resp under) the line correspond to the instances where the ordinate (resp the abscissa) strategy is less efficient. Larger is the gap between the axis line and the point, bigger is the difference between the strategies. Extreme points above and on the right correspond to the timeouts.

¹ Link.

Terminology. An instance is said to be *solved*, when the best solution has been found and its value proved to be optimal. The term *solution quality* is used when the search is incomplete, and only the best found solution can be judged.

4.2 OBS Evaluation

Once again, the *OBS* selector is highly configurable: each bound can have its own coefficient impacting the selection process. The running time of several configurations with different bounds importance have been profiled. The values -1 , 1 and 0 have been tested to respectively give: negative, positive, or no importance to the considered bound. All possible pairs of (a, b) from $(-1, 0, 1) \times (-1, 0, 1) \setminus \{(0, 0)\}$ have been tested.

The performance of different *OBS* parameters are shown in Fig. 3. This cumulative plot shows how many instances can be solved by each method, for a given time limit. This plot shows that a negative cost to the lower bound outperforms zero or positive cost, regardless of the upper bound.

Configurations weighting the lower bound negatively solve approximately 50 more instances than the alternatives. The solution quality has been compared as well: Fig. 4 shows how many time a search has found the best solution (not necessarily optimal) compared to its alternatives. Once again, the searches weighting the lower bound negatively show better results. One intuitive explanation is that choosing the variables impacting the less the bound which has to be optimized,

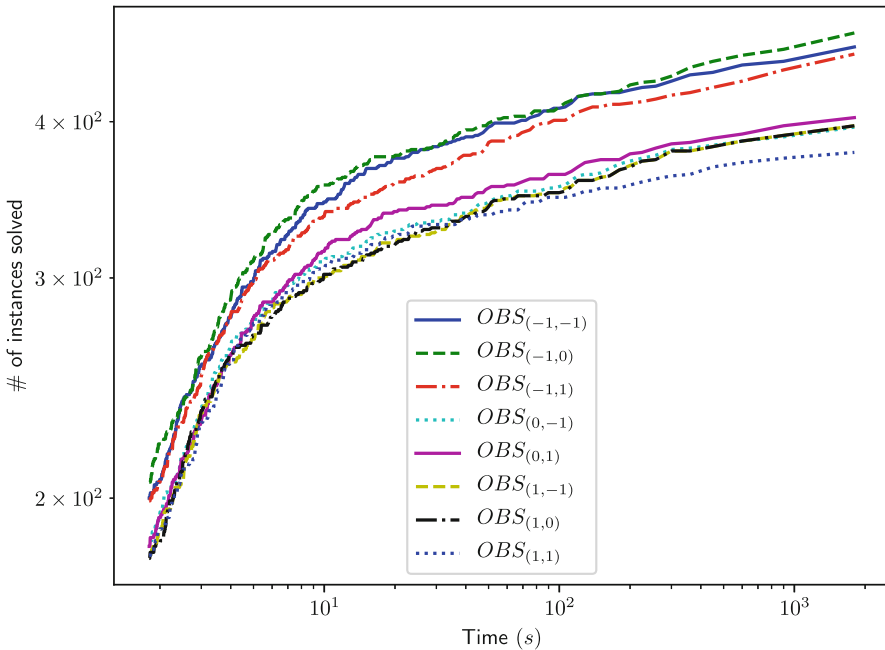


Fig. 3. Comparison of the number of solved instances for different *OBS* configuration.

concentrates the search into the most promising parts and like shown in Example 2 helps to back-propagate the objective to prune the tree search. Furthermore, the upper bound in optimization problems (here minimization, without loss of generality) does not have a big impact on resolution time. In addition to our previous intuitions, the upper bound seems to be very sensitive to initialization and to propagation. For instance in some constraints such as *sum*, which often determines the objective value, no arc consistency can be achieved in polynomial time. But, often, only the bounds are filtered, making less consistent the variations of this variable. Based on different OBS experiments, the configurations $(a = -1, b = 0)$ and $(a = -1, b = -1)$ seem to be the most promising.

4.3 Evaluation of Hybrid Strategies

We tried the hybridization with all the *OBS* configurations in order to select the most promising one. The configuration $(a = -1, b = 0)$ got better results within the hybridization, both in run-times and best objectives, which confirms our previous results. In the following, when no configuration is specified for *OBS*, then it means that the configuration $(a = -1, b = 0)$ has been used. Like *OBS*, the hybridization method is configurable in different ways. More or less importance can be given to the objective, or to the classical feature. In order to find the best parameter α , different experiments have been done. Figure 5 shows the comparison of different values of α on *ABS*. When *OBS* and *ABS* are not hybridized ($\alpha = 1$ and $\alpha = 0$), they clearly show orthogonal behaviors: the timeout are observed on different instances. By looking at the Fig. 5, it can be seen that $ABS_O(0.5)$ dominates the others: less timeout and better run-times are observed. Only a full comparison of *ABS* is presented here. We intentionally omit the remaining combinations to preserve clarity, but similar results are observed with the others hybridized strategies. The best combinations are reached when $\alpha = 0.5$. Thus in the following, when we are going to talk about a hybridized version, it will be always with $\alpha = 0.5$.

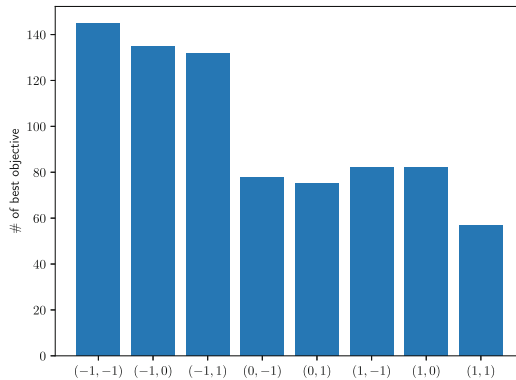


Fig. 4. Comparison of the objective quality between different OBS configurations.

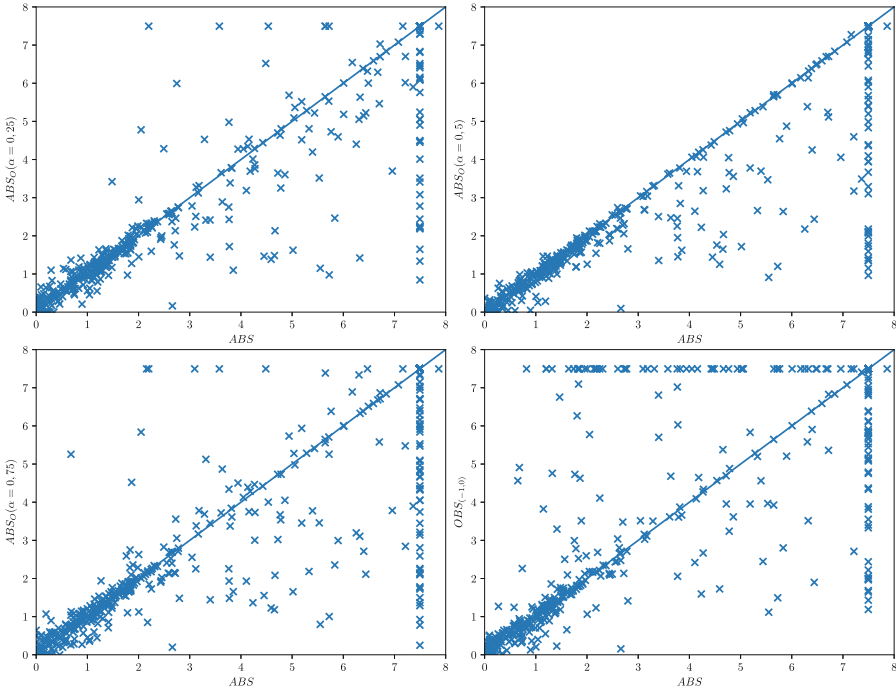


Fig. 5. Running time comparisons of the hybrid strategies, with respect to the hybridization parameters. From left to right and the top to the bottom, the configurations of ABS with (0.25) , (0.5) , (0.75) and (0) .

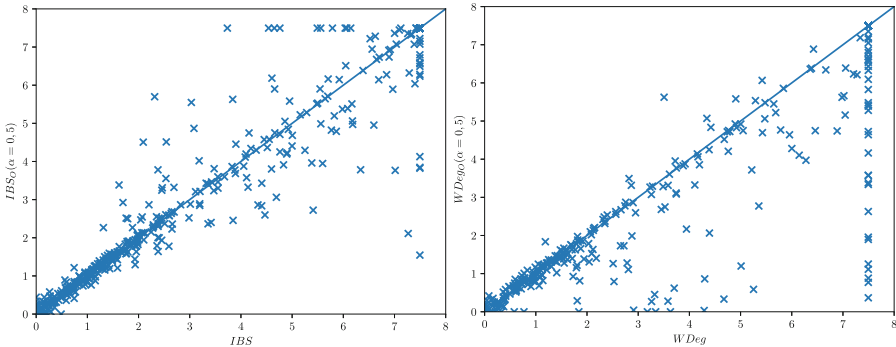


Fig. 6. Comparison of the original search against their hybridized version

The run-time and timeout comparisons between the others searches and their hybridized version are shown in Fig. 6. It is import to remark that $WDeg_O$ seems to outperform its original version, unlike IBS_O which has an orthogonal behavior. The Fig. 7 shows how the objective feature impacts the search to find

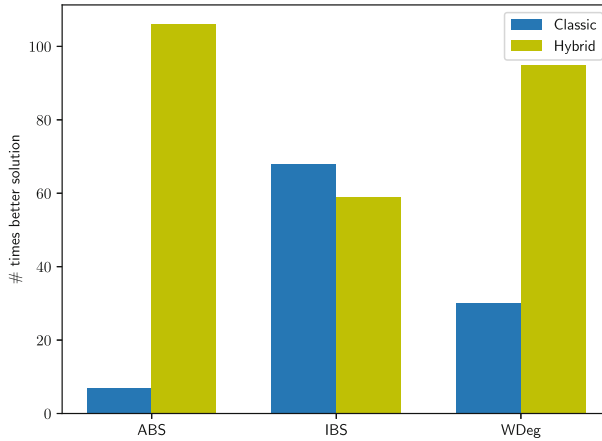


Fig. 7. Comparison of the objective quality between search strategies and their hybridized versions.

Table 1. Number of timeout in some families of instances.

Family	<i>OBS</i>	<i>ABS</i>	<i>WDeg</i>	<i>IBS</i>	<i>ABS_O</i>	<i>WDeg_O</i>	<i>IBS_O</i>
tdtsp	2	5	5	5	5	5	5
prize-collecting	2	7	7	7	7	2	8
2DBinPacking	7	8	8	8	6	8	8
mrcpsmm	0	3	1	1	0	0	0
mario	0	4	2	0	4	0	3
tpp	7	7	10	10	5	10	10
depot placement	3	7	7	1	4	6	4
p1f	2	1	7	1	2	7	3
table-layout	0	0	10	4	0	3	5
filters	2	1	5	1	1	5	1
amaze	4	3	5	4	3	5	4
open stack	8	5	10	7	4	9	6
talent scheduling	8	7	8	7	4	8	7

good solutions. It compares the number of times a search against its hybridized version has found a better solution. Unlike *IBS*, *ABS* and *WDeg* seem to benefit from the objective features, since their hybridized versions often find better solution than the original ones. For instance, the classical *ABS* find less than 10 better solution compared to its hybridized version which find more than 100 times.

To support again the interest of the hybridization method, we have extracted some interesting problem families in the Table 1. In this Table, even if *OBS* is not the best strategy, it is often able to solve problems where classical strategies do not. Furthermore, this table shows the interest of the hybridization, which most of the time takes advantage and improve the search considering only one feature. A good example is talent scheduling problem, *OBS* has 8 timeouts and *ABS* 7, but the hybridized version have only 4.

From the different plots and table presented, we remark that *IBS* is an exception because neither the original nor the hybridized version dominates each other and thus does not benefit as much as other search strategies from the hybridization. Actually, *IBS* contains already some information about the objective bounds modifications. The impact is computed over all variables including the objective. This is why the combination of the two features does not lead to a domination, but only an improvement in several problems and a decrease in some others. The resulting search is an orthogonal search to *IBS*.

4.4 Overall Evaluation

Figure 9 shows how many instances were solved as a function of time over all strategies. Without any hybridization *IBS* is the best strategy. However, with the hybridization, *ABS* shows the best improvements and so *ABS_O* become the best strategy. *ABS_O* has the largest number of solved instances under the allotted time. Furthermore, the hybridized versions are very competitive and improve the number of solved instances. Such a result confirms that using the objective as feature leads to strong improvement in solving time.

Most of the time, in real life problems, the optimal solution cannot be found or proved due to time limits. That is why we now compare the capabilities

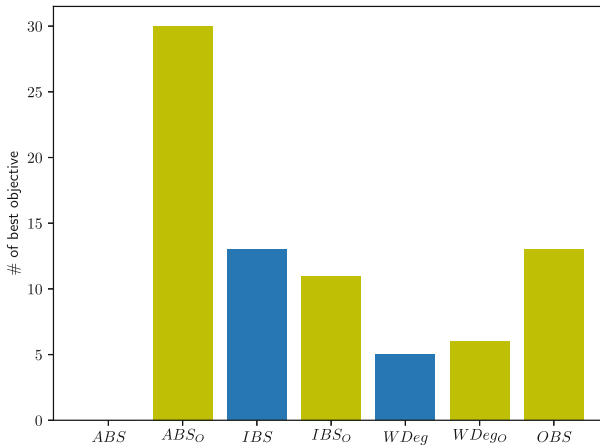


Fig. 8. Number of instances where each search strategy has found a strictly better objective compared to all the other. (Color figure online)

of *OBS* and the hybridized versions to find good solutions under an allotted time. The new hybrid strategies are very competitive in finding good solutions under a given amount of time as well. Figure 8 shows how many times a search strategy has found a strictly better solution than all the others. Searches using the objective feature are depicted in yellow and the others in blue.

ABS_O surpass the others and was able to find 30 times a strictly better objective than the others, while its original version *ABS* never finds a better solution. *IBS* and *OBS* seem to be the second best search strategies in terms of score. The hybridization shows again its advantages since ABS_O is strictly better than *ABS*. $WDeg_O$ slightly dominates *WDeg* and *OBS* has a good rank.

Miscellaneous Discussions. The objective can be monitor in many different ways. The $\Delta_O(t)$ was not our only trial, we tried to monitor the changes through a qualitative function counting how many times a variable modifies either the lower or the upper bounds. On the Minizinc Library, the qualitative function was dominated by the quantitative one.

Furthermore the $\Delta_O(t)$ function was used to designed a value selector. Different variants have been tried: first to select the value minimizing $\Delta_O(t)$, with possibly different values for a and b . Second to select the value using the new value heuristic from [3]. However, even if on some instances such as *ghoulomb* or *openstack* these selector showed a real improvement, they seem to globally

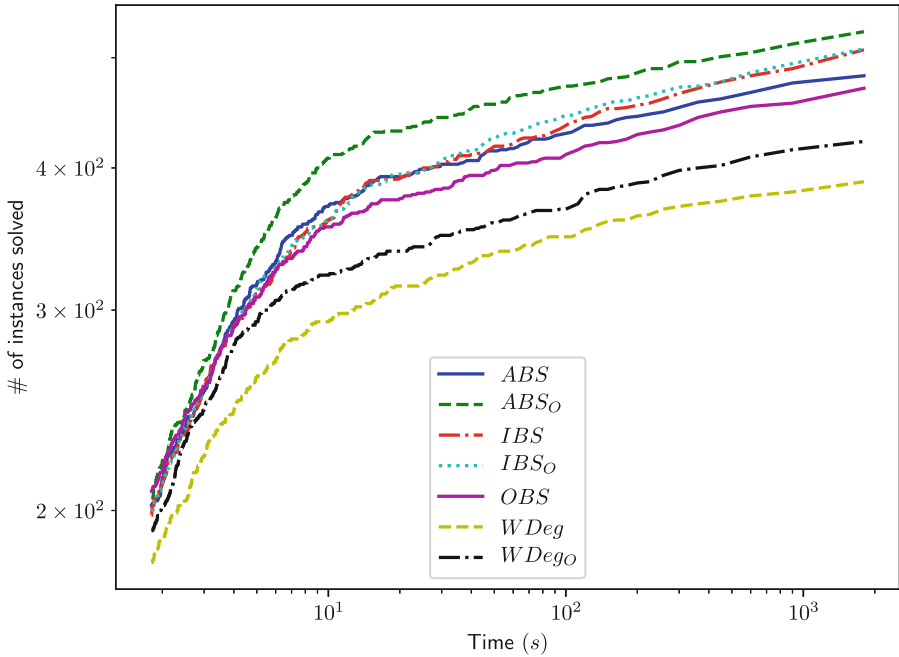


Fig. 9. Comparison of the number of instances solved by the different strategies as a function of time.

be dominated in the Minizinc problems set by *minVal*. The definition of a good value heuristic seems to still be a challenge to solve.

Our experimental section shows that combining classic search strategies with our objective-based feature leads to better performances and the ability to solve new problems. It shows that for *ABS* and *WDeg* adding an objective-based feature seems to dominate their performance. Finally it shows that the objective as a feature can play an important role in finding a good solution faster, as already claimed [3].

5 Related Work

The objective variable in COPs has already been considered in other fields such as max-SAT [8] to choose which literal to select, or in Soft-CSP for the decision value [10]. Large Neighborhood Search (LNS) framework also consider the objective: for example by changing the term of the weighted sum to minimize [13]. In constraint programming, the objective information is not yet well used. In [12], the authors propose a heuristic for weighted constraint satisfaction problems based on the solution quality to guide the value selection during the search. In [2], the authors propose a machine learning approach to learn the objective function from the variables' values, but not directly on the variables themselves.

More recently, counting based search has been adapted for optimization problems [17], the main idea being to consider objective-based solution density instead of a simple solution density. This is done by adding to each objective-based constraint an additional algorithm processing these values. Also, in MIP, the objective is widely used in the heuristic [4]: the variables having the best impact on the objective value of the relaxed problem are selected first. This approach differs from our, since CP does not have good relaxation as MIP and we consider the hybridization of the search strategies. A recent work [3] uses the objective information in order to select the variable value, leaving the variable selection to another strategy. Our method differs from [3] since we propose a variables selector, while [3] proposed a value selector. Secondly, we are trying to learn on-the-fly all along the search tree which variable seems to be the most promising, unlike [3]. In [3] the value is selected by testing all the possible assignments of the variable's domains to determine after the propagation which value is the best. Moreover, our feature is more fine-grained because it can be determined how strongly to emphasize bound modifications, using positive or negative parameters. In addition, in this paper we propose an hybridization of existing searches with the objective feature. More particularly our new strategies can be added into the set of available strategies to choose to solve a problem, even in online fashion [27].

6 Conclusion

In this paper we have demonstrated the need for using the objective variable as a feature for decisions within search strategies in constraint programming. We have defined a fine grain feature based on objective bound modifications. By using

this new feature, we have designed a new variable selector named *OBS*. This new variable selector is not the most efficient, but it is able to overpass the existing ones on some class of problems. Moreover, we have proposed a hybridization method to combine our proposed objective-based feature with many existing search strategies. Our evaluation has shown that the hybridized searches give great results and are better than the original strategy in term of run time and solution quality. Some searches are dominated by their hybrid versions. Through this new perspective, we have shown that using the objective as a feature to make decisions can lead to strong results. In addition, further work can be done, for example, with non valued SSs like the ones using a ranking criteria such as COS [5]. Directly applying this work on such SSs is not trivial, and should be a next step.

For both the $\Delta_O(t)$ function and the hybridization, we consider here only a linear combination of the values. More complex combination scheme can be considered. For example, non linear function or ranking function could be studied.

Finally, parameter optimization methods [9] could be used in order to find the best values of a , b and α for a given family of problem while solving it.

References

1. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI, vol. 16, p. 146 (2004)
2. Chu, G., Stuckey, P.J.: Learning value heuristics for constraint programming. In: Michel, L. (ed.) CPAIOR 2015. LNCS, vol. 9075, pp. 108–123. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18008-3_8
3. Fages, J.-G., Prud'Homme, C.: Making the first solution good! In: ICTAI 2017 29th IEEE International Conference on Tools with Artificial Intelligence (2017)
4. Gauthier, J.-M., Ribière, G.: Experiments in mixed-integer linear programming using pseudo-costs. *Math. Program.* **12**(1), 26–47 (1977)
5. Gay, S., Hartert, R., Lecoutre, C., Schaus, P.: Conflict ordering search for scheduling problems. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 140–148. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_10
6. Gent, I.P., Walsh, T.: CSPLib: a benchmark library for constraints. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 480–481. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-540-48085-3_36
7. Haralick, R.M., Elliott, G.L.: Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.* **14**(3), 263–313 (1980)
8. Heras, F., Larrosa, J.: New inference rules for efficient max-sat solving. In: AAI, pp. 68–73 (2006)
9. Hutter, F., Hoos, H., Leyton-Brown, K.: An evaluation of sequential model-based optimization for expensive blackbox functions. In: Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation, pp. 1209–1216. ACM (2013)
10. Larrosa, J., Schiex, T.: Solving weighted CSP by maintaining arc consistency. *Artif. Intell.* **159**(1–2), 1–26 (2004)
11. Lecoutre, C., Saïs, L., Tabary, S., Vidal, V.: Reasoning from last conflict(s) in constraint programming. *Artif. Intell.* **173**(18), 1592–1614 (2009)

12. Levasseur, N., Boizumault, P., Loudni, S.: A value ordering heuristic for weighted CSP. In: 19th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2007, vol. 1, pp. 259–262. IEEE (2007)
13. Lombardi, M., Schaus, P.: Cost impact guided LNS. In: Simonis, H. (ed.) CPAIOR 2014. LNCS, vol. 8451, pp. 293–300. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07046-9_21
14. Michel, L., Van Hentenryck, P.: Activity-based search for black-box constraint programming solvers. In: Beldiceanu, N., Jussien, N., Pinson, É. (eds.) CPAIOR 2012. LNCS, vol. 7298, pp. 228–243. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29828-8_15
15. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_38
16. Palmieri, A., Régin, J.-C., Schaus, P.: Parallel strategies selection. In: Rueher, M. (ed.) CP 2016. LNCS, vol. 9892, pp. 388–404. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44953-1_25
17. Pesant, G.: Counting-based search for constraint optimization problems. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, Arizona, 12–17 February 2016, USA, pp. 3441–3448 (2016)
18. Pesant, G., Quimper, C.-G., Zanarini, A.: Counting-based search: branching heuristics for constraint satisfaction problems. *J. Artif. Intell. Res. (JAIR)* **43**, 173–210 (2012)
19. Prud’homme, C., Fages, J.-G., Lorca, X.: Choco Documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S (2016)
20. Refalo, P.: Impact-based search strategies for constraint programming. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 557–571. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30201-8_41
21. Régin, J.-C.: A Filtering algorithm for constraints of difference in CSPs. In: AAAI, vol. 94, pp. 362–367 (1994)
22. Schaus, P., Van Hentenryck, P., Régin, J.-C.: Scalable load balancing in nurse to patient assignment problems. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 248–262. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01929-6_19
23. Simonis, H., O’Sullivan, B.: Search strategies for rectangle packing. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 52–66. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85958-1_4
24. Smith, B.M., Grant, S.A.: Trying harder to fail first. Research Report Series-University of Leeds School of Computer Studies LU SCS RR (1997)
25. Vilfm, P., Laborie, P., Shaw, P.: Failure-directed search for constraint-based scheduling. In: Michel, L. (ed.) CPAIOR 2015. LNCS, vol. 9075, pp. 437–453. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18008-3_30
26. Wallace, M.: Practical applications of constraint programming. *Constraints* **1**(1–2), 139–168 (1996)
27. Xia, W., Yap, R.H.C.: Learning robust search strategies using a bandit-based approach. In: AAAI Conference on Artificial Intelligence (2018)
28. Zitoun, H., Michel, C., Rueher, M., Michel, L.: Search strategies for floating point constraint systems. In: Beck, J.C. (ed.) CP 2017. LNCS, vol. 10416, pp. 707–722. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66158-2_45



PW-CT: Extending Compact-Table to Enforce Pairwise Consistency on Table Constraints

Anthony Schneider^(✉) and Berthe Y. Choueiry^(✉)

Constraint Systems Laboratory, University of Nebraska-Lincoln, Lincoln, USA
{aschneid,choueiry}@cse.unl.edu

Abstract. The Compact-Table (CT) algorithm is the current state-of-the-art algorithm for enforcing Generalized Arc Consistency (GAC) on table constraints during search. Recently, algorithms for enforcing Pairwise Consistency (PWC), which is strictly stronger than GAC, were shown to be advantageous for solving difficult problems. However, PWC algorithms can be costly in terms of CPU time and memory consumption. As a result, their overhead may offset the savings of search-space reduction. In this paper, we introduce PW-CT, an algorithm that modifies CT to enforce full PWC. We show that PW-CT avoids the high memory requirements of prior PWC algorithms and significantly reduces the time required to enforce PWC.

1 Introduction

Consistency properties and algorithms for enforcing them on a Constraint Satisfaction Problem (CSP) are one of the most intensively studied topics in Constraint Programming (CP). Consistency algorithms are used for inference and effectively reduce the search space of solving a CSP. In particular, Generalized Arc Consistency (GAC) has recently been the focus of extensive research for a good reason: it lends itself towards simple yet highly effective algorithms. Indeed, the low cost and effectiveness of GAC algorithms when paired with an ordering heuristic like dom/wdeg have made them the de facto baseline for research. The current state-of-the-art in GAC algorithms are Compact-Table (CT) [8] and STRBit [27], both of which use bitsets to quickly check for supports and perform tabular reduction – the process of removing invalid tuples from constraints.

Algorithms that enforce pairwise-consistency (PWC) have received a relatively modest amount of attention [23]. Recent algorithms have shown promise on some benchmarks [15–17, 20], at times considerably reducing the size of the search space. However, enforcing GAC with either CT or STRBit outperforms

The original version of this chapter was revised: The title has been corrected. The correction to this chapter is available at https://doi.org/10.1007/978-3-319-98334-9_48

Supported by NSF Grant No. RI-1619344. Work completed utilizing the Holland Computing Center of the University of Nebraska, which receives support from the Nebraska Research Initiative. We thank the reviewers for constructive feedback.

these PWC algorithms due to the latter’s initialization overhead, memory usage, and computational cost.

In this paper, we introduce PW-CT, an algorithm for enforcing full PWC, as an extension of CT [8]. PW-CT requires few modifications to the original CT structures, exploits mechanisms from existing PWC algorithms, and integrates additional improvements discussed in this paper. More specifically, PW-CT uses CT (i.e., GAC) as much as possible to avoid costly PWC checks in two ways: by ensuring the problem is GAC before resorting to any PWC checks and by identifying situations where GAC guarantees PWC. Finally, it exploits properties of the dual CSP to speed-up processing and reduce memory consumption. We compare the performance of PW-CT to that of state-of-the-art GAC and full PWC algorithms. We show that PW-CT dominates the latter by a large margin and outperforms both STRBit and CT on several benchmarks.

This paper is structured as follows. Section 2 provides background information. Section 3 reviews the state of the art. Section 4 identifies directions to improve PWC algorithms. Section 5 discusses PW-CT. Section 6 discusses our experiments. Finally, Sect. 7 concludes this paper.

2 Background

A Constraint Satisfaction Problem (CSP) is defined as $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, where \mathcal{X} is a set of variables, \mathcal{D} the set of their domain values, and \mathcal{C} is a set of constraints $c_i = \langle R_i, scope(c_i) \rangle$, where R_i is a relation defined on the variables in the scope of the constraint, $scope(c_i) \subseteq \mathcal{X}$, restricting the combination of values that the variables can take at the same time. The arity of a constraint is the cardinality of its scope. Solving a CSP requires assigning to each variable a value from its domain such that all the constraints are satisfied. In this paper, we consider table constraints, where relations are given by their allowed tuples. We use the relational projection operator π to restrict a tuple to a set of variables.

In the *hypergraph* representation of a CSP, the vertices represent CSP variables. Hyperedges represent constraints and connect the variables in the scope of the constraints. In the *dual graph* representation, the vertices represent the CSP constraints and edges connect vertices that share variables. Figure 1 shows the dual graph of a CSP with four nonbinary constraints. The edges are equality constraints forcing the shared variables to agree on the assigned values.

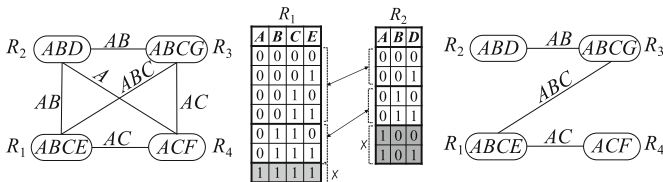


Fig. 1. Dual graph (left), subsopes and blocks (center), a minimal dual graph (right)

We designate by *subscope* the set of CSP variables shared by two constraints, $subscope(c_i, c_j) = scope(c_i) \cap scope(c_j)$. Multiple edges in the dual graph may be

labeled by the same subscope. In Fig. 1, each of the subscoopes AB and AC label two edges and each of the subscoopes A and ABC label one edge. The equality constraints of the dual graph are binary and *piecewise functional* [10,23]. A binary constraint is said to be piecewise functional if the domains of the variables in its scope can be partitioned such that a set from one variable is supported by at most one set in the other and vice versa. Because of the piecewise functionality of the constraints of the dual graph, each subscope partitions the tuples of a constraint into sets of equivalent tuples, which we call *blocks*. In Fig. 1, the subscope AB partitions each of the two relations R_1 and R_2 into three blocks. We define the *signature* of a block as the set of variable-value pairs of the inducing subscope (e.g., $\{\langle A, 0 \rangle, \langle B, 0 \rangle\}$). Thus, a signature is uniquely determined by a combination of a constraint, subscope, and tuple. Janssen et al. [11] and Dechter [7] observed that, in the dual graph, an edge between two vertices is *redundant* if there exists an alternate path between the two vertices such that the shared variables appear in every vertex in the path. Redundant edges can be removed without affecting the set of solutions. Janssen et al. [11] introduced an efficient algorithm for computing the *minimal dual graph* by removing redundant edges. Many minimal graphs may exist, but all are guaranteed to have the same number of edges. Figure 1 shows an example of a minimal dual graph.

In this paper, we exploit the following two consistency properties:

Definition 1. Generalized Arc Consistency (GAC) [18,26]: *A constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is GAC iff, for every constraint $c_i \in \mathcal{C}$, and $\forall x_j \in \text{scope}(c_i)$, every value $v \in D(x_j)$ is consistent with c_i (i.e., appears in some support of c_i).*

Definition 2. Pairwise Consistency (PWC) [9]: *A constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is PWC iff, for every tuple t_i in every constraint c_i there is a tuple t_j in every constraint c_j such that $\pi_{\text{subscope}(c_i, c_j)}(t_i) = \pi_{\text{subscope}(c_i, c_j)}(t_j)$, t_j is called a PW-support of t_i in c_j . A CSP that is both PWC and GAC is said to be full PWC (fPWC).*

3 Related Work

The CT algorithm [8] is the current state-of-the-art algorithm for enforcing GAC.¹ It makes heavy use of a data structure called an `RSparseBitSet`, which is similar to the sparse set structure [5,6]. In this paper, we extend the definition of the `RSparseBitSet` to facilitate PWC operations.

`STRBit` [27] is a GAC algorithm similar to CT in that it operates on bitsets. `STRBit` differs from CT in its data structures and how it propagates changes. To our knowledge, a comparison of the performance of these two algorithms has not yet appeared in the literature.

Samaras and Stergiou introduced PW-AC [23], an algorithm for enforcing PWC. PW-AC operates on the dual graph of the CSP, taking advantage of the piecewise functionality of the equality constraints. It propagates deletions

¹ In this paper, we consider that a GAC algorithm applies tabular reduction.

of blocks of tuples, rather than individual tuples, and maintains counts of the living tuples in each block. Notably, it iterates over *every pair* of constraints with a non-empty subscope.

The parametrized algorithms PerTuple [12, 13] and PerFB [24] enforce m -wise consistency, which generalizes pairwise consistency to every set of m constraints. Both algorithms use data structures that group the equivalent tuples in a constraint based on the piecewise-functionality property.

Algorithm eSTR [15], an extension of the STR family of algorithms for GAC [14, 25], enforces fPWC. It maintains PWC by tracking the counts of PW-supports of each tuple in a constraint relative to all other constraints in the problem and verifying a valid PW-support for each tuple as it is processed by STR. Algorithm eSTR^w is a modification of eSTR and enforces a weakened version of fPWC by not re-queuing a constraint after a PW-support is lost.

Algorithms HOSTR and MaxRPWC+r [20] enforce consistency properties that are weaker than PWC and incomparable to each other. fHOSTR, a variant of HOSTR, enforces full PWC, but was found by its authors to be too expensive relative to its weakened version. Paparrizou and Stergiou show that HOSTR and MaxRPWC+r outperform STR2 [14] on certain benchmarks.

Some approaches for enforcing higher order consistencies apply GAC after reformulating the CSP with new constraints or variables. Algorithm DkWC [19] enforces k -wise consistency by adding new hybrid constraints to the problem. The Factor Encoding (FE) enforces fPWC by adding new variables to the problem, thereby increasing the arity of constraints [16]. A decomposition of the FE lessens the imposed arity increases from FE while still enforcing fPWC [17].

4 Improving PWC Algorithms

Below, we describe four distinct techniques to improve the performance of PWC algorithms. These methods can be combined or exploited in isolation.

4.1 Piecewise Functionality

As mentioned in Sect. 2, Samaras and Stergiou [23] exploit the piecewise-functional property of the equality constraints of the dual graph to infer the blocks of equivalent tuples of two constraints with shared variables. If a tuple τ in a constraint c_i does not have a PW-support in another constraint c_j , all tuples in the block induced by $\pi_{\text{subscope}(c_i, c_j)}(\tau)$ on c_i can be immediately removed. Further, all other *blocks* of tuples that are PW-supported by τ in all other neighboring constraints (i.e., have the same signature) must also be deleted. This operation is in stark contrast with most GAC-based algorithms that search for supports one tuple at a time (except, of course, AC-5 [10]).

4.2 Pairwise Vs Subscope Reasoning

Algorithms for enforcing pairwise consistency usually operate on *every pair* of constraints with overlapping subsopes (e.g., PW-AC partitions relations pairwise, eSTR counts supports pairwise, etc.). Karakashian et al. [13] and Schneider

et al. [24] exploit the fact that, for a given subscope, any number of relations induce on a relation R_i the same unique partition. For example, in Fig. 1, the blocks induced by subscope $\{A, B\}$ on relation R_1 are the same for *any* relation R_j such that $\text{subscope}(R_1, R_j) = \{A, B\}$.

Consequently, identifying and storing a relation's partitions based on unique subscoopes rather than by the degree of a vertex in the dual graph can significantly reduce the memory requirements of algorithms that exploit the pairwise functionality of the equality constraints of the dual graph.

4.3 Minimal Dual Graph

As stated in Sect. 2, we can remove redundant edges in the dual graph of a CSP without affecting the set of solutions. In fact, Janssen et al. [11] show that enforcing PWC on a dual graph is equivalent to enforcing PWC on any of its minimal dual graphs. Importantly, removing redundant edges can reduce not only the degree of the graph (thus reducing the number of pairs of constraints over which a PWC algorithm must iterate) but also the number of unique subscoopes that a PWC algorithm must take into consideration. For instance, in the example shown in Fig. 1, removing redundant edges eliminates: (1) The need to compute and store the partitions of R_1 for the subscope $\{A, B\}$ and the partitions of R_3 for the subscope $\{A, C\}$ and (2) The subscope $\{A\}$ and the partition it induces on each of R_2 and R_4 . Consequently, a minimal dual graph can reduce the number of neighbors of a constraint in the problem, the number of unique subscoopes incident to a constraint, and may entirely eliminate some subscoopes from the problem. We conclude that a PWC algorithm that operates on a minimal dual graph may reduce its memory requirements and increase its propagation speed because of the reduced number of subscoopes to consider per constraint and the total number of unique subscoopes.

4.4 Determining When GAC Is Enough to Enforce PWC

In some situations, GAC is enough to enforce PWC between constraints. The algorithm eSTR, for example, only checks for PW-supports over “non-trivial” subscoopes, which are subscoopes with a cardinality strictly greater than one [15]. In fact, the particularity of constraints intersecting on at most one variable is discussed by Bessiere et al. [3] but PWC is unexplainably excluded from the corresponding theorem. Below, we restate this property and give a proof:

Proposition 1. *GAC is sufficient to enforce PWC over trivial subscoopes.*

Proof. Consider the CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$. If a subscope is trivial (e.g., $\text{subscope} = \{x\} \subset \mathcal{X}$) the signature of each block induced by this subscope is one variable-value pair (e.g., $\langle x, a \rangle$). Thus, the block loses all PW-supports only if $\langle x, a \rangle$ is removed from the problem. If $\langle x, a \rangle$ is deleted, a tabular-reduction algorithm necessarily removes all tuples with $\langle x, a \rangle$ from the problem. On the other hand, if $\langle x, a \rangle$ is alive after enforcing GAC, then, by definition, $\forall c_i \in \mathcal{C}$ such that

$x \in \text{scope}(c_i)$, there is at least one living tuple τ in the relation of c_i such that $\pi_x(\tau) = a$. \square

We elucidate a particular situation, which arises during search, in which the above property holds even for non-trivial subsopes as long as GAC is enforced on a constraint prior to running a PWC algorithm:

Proposition 2. *GAC is sufficient to enforce PWC on a block induced by a non-trivial subscope whose signature includes a deleted variable-value pair.*

Proof. This proposition follows from Proposition 1. Consider a block b_i induced by a non-trivial subscope σ_i on the constraint c_i . If a dead variable-value pair $\langle x, a \rangle$ is in the block's signature, a tabular-reduction GAC algorithm removes all tuples with $\langle x, a \rangle$ from the problem, and as a result, it removes all the PW-supports of b_i from the relations of neighboring constraints because they necessarily also contain $\langle x, a \rangle$ in their signatures. \square

Algorithm eSTR [15] implicitly applies this principle by ensuring that all the variable-value pairs of a tuple are alive before checking whether or not the tuple has PW-supports in neighboring constraints. We exploit Proposition 2 in PWC algorithms in a slightly more efficient manner. Assume a CSP is already PWC, after a variable is instantiated, we run an STR-based GAC, which may delete tuples from constraints. We now need to process these deleted tuples because some of them may be the sole tuples of some blocks that were the PW-support of other blocks in other constraints. In the case that a variable-value pair deleted by GAC appears in the signature of a block in which one of these deleted tuples appears, we can safely skip the processing we intended to do because its result is ensured by GAC. This operation is implemented in function ENFORCEPWC (Algorithm 4) in Sect. 5.2. In summary, Algorithm eSTR exploits the property by checking first whether the tuple is GAC and we exploit the property by avoiding checking PWC on blocks that we know are dead.

5 PW-CT

We now introduce PW-CT, an fPWC algorithm that exploits the mechanisms presented in Sect. 4. First, we describe how we modify the RSParseBitSet class of the CT algorithm [8] and the additional data structures required for enforcing PWC. Then, we provide the pseudocode of PW-CT.

5.1 Data Structures

PW-CT exploits the functions and data structures CT [8]. Below, we review the CT data structures and the additions required for PW-CT.

Support Structures. Both CT and PW-CT represent the living tuples in a constraint as an `RSparseBitSet`. The `RSparseBitSet` stores four members: an array of reversible 64-bit integers called `words`,² a reversible integer called `limit` that represents the number of non-zero integers in `words`, an array called `index` that stores the position of all non-zero integers in `words` in locations less-than or equal-to `limit`, and an array called `mask` used to modify the set. Demeulenaere et al. [8] introduce member functions of the `RSparseBitSet` used by PW-CT which we briefly review: function `addToMask` takes an array and alters `mask` to be the bitwise OR of the array and the current `mask`, function `intersectWithMask` alters `words` to be the bitwise AND of the current `words` and `mask`, and function `clearMask` sets the integers in `mask` to 0.

The `RSparseBitSet` for a constraint c_i is denoted as `living(c_i)`. The data structure `supports[c_i, x, a]` is a static array of bits corresponding to the tuples of a constraint c_i that have the value a for variable x .³ To improve performance of various functions in PW-CT, we introduce a structure `indices[c_i, x, a]`, which is an `RSparseBitSet` that stores the positions in `supports[c_i, x, a]` that are non-zero.

PW-CT uses two maps. The first, `incidentCons[σ]`, gives the list of constraints incident to a non-trivial subscope σ . The second, `incidentSubscopes[c_i]`, gives the list of non-trivial subsopes incident to a constraint c_i . We can optionally use the minimal dual graph to reduce the number of generated subsopes in each map without affecting the level of consistency enforced (see Sect. 4). Importantly, all these support structures are created at initialization.

Blocks. We represent a block as a simple structure with a member `sets`, which is a vector of pointers to `supports[c_i, x, a]` representing the signature of the block, and a member `commonIndices`, which is an `RSparseBitSet` of the indices shared by all of the supports in `sets`. Performing an intersection of the `sets` in a block computes the set of tuples with the signature corresponding to `sets`. In PW-CT, blocks are never stored but always computed dynamically during search.

The function `CREATEBLOCK` (Algorithm 1) takes as input a constraint, tuple, and subscope and returns a block structure, which can be used to dynamically compute the partition of tuples of the constraint with the corresponding signature. The `RSparseBitSet` `commonIndices` improves performance of some operations of the methods listed in Algorithm 2. Note that the method `initIntersection` called in Line 7 is defined in Algorithm 2 and makes use of the call `swap` in Line 5.

Additional Methods for the `RSparseBitSet` Class. Algorithm 2 introduces additional methods for the `RSparseBitSet` class for use in PW-CT. The method `initIntersection` is used in `CREATEBLOCK` to initialize the `RSparseBitSet` with the indices common to a collection of `RSparseBitSets`.

² 64-bit on most current architectures.

³ Note that we have added the additional parameter c_i to `supports[]` to uniquely determine the constraint's supports we are referring to in the pseudocode.

Algorithm 1. CREATEBLOCK(c_i, τ, σ)

Input: A constraint c_i , a tuple τ , and a subscope σ **Output:** A block b

```

1  $j \leftarrow 0$ 
2 foreach variable  $x \in \sigma$  do
3    $b.\text{sets}[j] \leftarrow \text{supports}[c_i][x][\tau[x]]$  //  $\tau[x]$  is the value for  $x$  in tuple  $\tau$ 
4    $\text{ind}[j] \leftarrow \text{indices}[c_i][x][\tau[x]]$ 
5   if  $\text{ind}[j].\text{limit} < \text{ind}[0].\text{limit}$  then  $\text{swap}(\text{ind}[j], \text{ind}[0])$ 
6    $j \leftarrow j + 1$ 
7  $b.\text{commonIndices}.\text{initIntersection}(\text{ind})$ 
8 return  $b$ 

```

We overload the original RSparesBitSet method `intersectIndex` to operate on blocks. It is similar in behavior to the original `intersectIndex`, differing in that it determines if a *block* of tuples has a support in the set, rather than a single variable-value pair. The method `removeBlock` computes the set-difference between the RSparesBitSet and a block of tuples.

Now, we list functions omitted from Algorithm 2 for brevity. The methods `save` and `restore` respectively save and restore the state of the reversible elements in the RSparesBitSet. We maintain the number of living bits when altering the set and `numSet` returns this value.⁴ PW-CT relies on the ability to discover the tuples removed between two points in time (the delta of the set). To this end, method `computeDelta` returns an RSparesBitSet containing the bits removed between the current state of the RSparesBitSet and the last stored state. Method `clearDelta` readies the set to track the next set of removed tuples, but does not alter the currently set bits. These were implemented using the method `save` and comparing the reversible primitives of the current state of the set and its previously saved state. Method `addBlockToMask` behaves like the original `addToMask`, but adds to the mask only those bits common to all bit-sets in the block. Its implementation follows from `addToMask` and `intersectIndex`. We also assume that the bits in the RSparesBitSets are iterable and treat the bits and the tuples they represent interchangeably in our pseudocode for simplicity.

5.2 Enforcing PW-CT

Roughly speaking, PW-CT has two main phases: a GAC phase, in which CT is executed until quiescence, and a PWC phase that performs a *single* pass over the tuples deleted by CT to uncover new non-PWC blocks. PW-CT maintains two queues: CTQueue tracks constraints that must be ‘checked for GAC’ and PWCQueue tracks constraints that have lost tuples thus threatening the PW-consistency of blocks in other constraints. Both queues are sets.

Function LOOKAHEAD (Algorithm 3) is the entry point for PW-CT. Lines 4 to 7 run CT until quiescence and enqueues constraints modified by GAC into

⁴ This can be done efficiently in C++ with Clang/GCC’s `__builtin_popcountll`.

Algorithm 2. Additional algorithms required for `RSparseBitSet`

```

1 Method initIntersection(sets: A vector of RSparseBitSets):
2   limit  $\leftarrow$  -1
3   index  $\leftarrow$   $\emptyset$ 
4   Expand words and index to size of sets[0].words
5   foreach  $i \leftarrow 0$  to sets[0].limit do
6     offset  $\leftarrow$  sets[0].index[i]
7     bits  $\leftarrow$  sets[0].words[offset]
8     for set  $\in$  sets and bits  $\neq 0$  do
9       bits  $\leftarrow$  bits & set.words[offset]           // Bitwise AND
10      if bits  $\neq 0$  then
11        words[offset]  $\leftarrow$  bits
12        limit  $\leftarrow$  limit + 1
13        index[limit]  $\leftarrow$  offset
14 Method intersectIndex(block: A Block created by CREATEBLOCK):
15   // If limit < block.commonIndices.numSet(), iterate from 0 to limit
16   for offset  $\in$  block.commonIndices do
17     intersection  $\leftarrow$  words[offset]
18     for set  $\in$  block.sets and intersection  $\neq 0$  do
19       intersection  $\leftarrow$  intersection & set.words[offset]           // Bitwise AND
20       if intersection  $\neq 0$  then return offset
21   return -1
22 Method removeBlock(block: A Block created by CREATEBLOCK):
23   for  $i \leftarrow$  limit to 0 do
24     offset  $\leftarrow$  index[i]
25     if offset  $\in$  block.commonIndices then
26       b  $\leftarrow$  64-bit Integer with all bits set
27       for set  $\in$  block.sets and b  $\neq 0$  do
28         b  $\leftarrow$  b & set.words[offset]           // Bitwise AND
29         words[offset]  $\leftarrow$  words[offset] &  $\sim$ b           // Bitwise NOT
30         if words[offset] = 0 then
31           index[i]  $\leftarrow$  index[limit]
32           index[limit]  $\leftarrow$  offset
33           limit  $\leftarrow$  limit - 1

```

`PWCQueue`. `CT` enqueues constraints with modified variables into `CTQueue`, thus, when execution hits Line 9, the problem is `GAC` but not necessarily `PWC`. Lines 9 to 13 call function `ENFORCEPWC` (Algorithm 4) on modified constraints to determine if the removal of tuples in each constraint c_i in the queue causes the loss of a `PW`-support in another constraint.

`ENFORCEPWC` iterates over all subscopes incident to a constraint and the constraint's most recently removed tuples, checking whether the block induced

Algorithm 3. LOOKAHEAD(\mathcal{P})	Enforces PWC on a CSP \mathcal{P}
Input: A CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$	
Output: Whether the current problem is consistent	
<pre> 1 consistent \leftarrow true 2 if \mathcal{P} has not been preprocessed then 3 \leftarrow consistent \leftarrow PreProcess(\mathcal{C}) 4 while consistent and not empty(CTQueue) do 5 $c_i \leftarrow$ pop(CTQueue) 6 consistent \leftarrow CompactTable(c_i) 7 if c_i was modified then push(c_i, PWCQueue) 8 if consistent and empty(CTQueue) then 9 mCons \leftarrow PWCQueue 10 for $c_i \in$ mCons and consistent do 11 consistent \leftarrow EnforcePWC(c_i) 12 living(c_i).clearDelta() 13 PWCQueue \leftarrow PWCQueue \setminus $\{c_i\}$ 14 return consistent </pre>	

by the combination of each subscope and tuple is empty. As discussed in Sect. 4.4, any blocks whose signatures have variable-value pairs removed by GAC necessarily have had all of their supporting blocks in neighboring constraints removed as well. The loop beginning at Line 4 in ENFORCEPWC (Algorithm 4) takes advantage of this insight by discarding blocks of tuples from consideration for PW-support checks, skipping unnecessary calls to REVISEBLOCK (Algorithm 5). It uses a mechanism similar to the incremental and reset-based updates [22], where Δ_x is the set of values of variable x removed by the previous call to CT.

Lines 16 to 19 check the block induced by each removed tuple for the current subscope for validity by calling function REVISEBLOCK (Algorithm 5). If no other tuples in the induced block are alive in the constraint, REVISEBLOCK removes the piecewise-functional blocks from all other constraints incident to the current subscope, and enqueues the constraints modified during this process. Multiple tuples in the set of removed tuples may belong to the same block for a given subscope, so, Line 19 removes all other tuples from that block from the set of tuples to check (as successive calls for the same block would be redundant).

It is advantageous to interleave CT and ENFORCEPWC calls because tuples removed by ENFORCEPWC may enable value deletions that can be propagated quickly by CT. To prevent running ENFORCEPWC until quiescence on the first pass, a copy of the queue is created in Line 9 of LOOKAHEAD (Algorithm 3). As a result, each modified constraint is processed at most once at each PWC pass.

Proposition 3. *If the CSP is initially PWC, LOOKAHEAD guarantees fPWC.*

Proof. Consider a constraint c_i altered by CT. Because the problem was PWC prior to running CT, the only ‘endangered’ blocks in c_i have tuples deleted by CT. To enforce PWC, we need to check if any block b_i whose signature is a

Algorithm 4. ENFORCEPWC(c_i)Propagates invalid blocks of c_i

Input: Constraint c_i that has been modified by CT
Output: Whether the current problem is consistent

```

1 tupsToCheck  $\leftarrow$  living( $c_i$ ).computeDelta()
2 for  $\sigma \in$  incidentSubscopes[ $c_i$ ] do
3   tupsToCheck.save()
4   for variable  $x \in \sigma$  s.t.  $x$  was modified on previous call to CT do
5     if  $|\mathcal{D}(x)| < |\Delta_x|$  then
6       tupsToCheck.clearMask()
7       for value  $a \in \mathcal{D}(x)$  do
8         tupsToCheck.addToMask(supports[ $c_i$ ][ $x$ ][ $a$ ])
9       tupsToCheck.intersectWithMask()
10    else
11      for value  $a \in \Delta_x$  do
12         $b \leftarrow$  an empty block
13         $b$ .sets  $\leftarrow$  supports[ $c_i$ ][ $x$ ][ $a$ ]
14         $b$ .indices  $\leftarrow$  indices[ $c_i$ ][ $x$ ][ $a$ ]
15        tupsToCheck.removeBlock( $b$ )
16  for  $\tau \in$  tupsToCheck do
17    consistent  $\leftarrow$  ReviseBlock( $c_i, \sigma, \tau$ )
18    if not consistent then return false
19    tupsToCheck.removeBlock(CreateBlock( $c_i, \sigma, \tau$ ))
20  tupsToCheck.restore()
21 return true

```

Algorithm 5. REVISEBLOCK(c_i, σ, τ)

Removes supports of empty block

Input: A constraint c_i , a subscope σ , and a tuple τ
Output: Whether the current problem is consistent

```

1 if living( $c_i$ ).intersectIndex(CreateBlock( $c_i, \sigma, \tau$ )) = -1 then
2   for  $c_j \in$  incidentCons[ $\sigma$ ] s.t.  $c_i \neq c_j$  do
3     living( $c_j$ ).removeBlock(CreateBlock( $c_j, \sigma, \tau$ ))
4     if living( $c_j$ ) was modified then
5       if living( $c_j$ ).numSet() = 0 then return false
6       push( $c_j, \text{PWCQueue}$ )
7       push( $c_j, \text{CTQueue}$ )
8 return true

```

combination of a deleted tuple τ of c_i , a subscope σ_i incident to c_i and c_i is empty as a result of CT. If we find a block b_i to be empty, we can remove the blocks that are PW-supports of b_i from all constraints c_j incident to σ_i . Because each c_j modified in REVISEBLOCK is added to the PWCQueue (Line 6), the removal of any tuple in c_j by REVISEBLOCK that emptied a block induced on any subscope

σ_j is necessarily detected by the next call to ENFORCEPWC(c_j). Running CT in between calls to ENFORCEPWC on any modified constraint ensures that the domains of the variables in the scope of the constraint are ‘synced’ with the constraint’s relation, thus, ensuring fPWC. \square

Proposition 4. *The time complexity of calling ENFORCEPWC on a constraint is $O((|\mathcal{C}| \cdot t) \cdot (\lceil \frac{t}{64} \rceil \cdot |\mathcal{C}| + |\sigma|))$, where t is the number of tuples in the largest constraint and σ the largest subscope.*

Proof. REVISEBLOCK iterates over the constraints incident to a subscope, which in the worst case is $|\mathcal{C}| - 1$. Each constraint may need to call removeBlock, which requires iterating over $\lceil t/64 \rceil$ elements. Creating the block requires iterating over σ . The only tuples evaluated by REVISEBLOCK are those that have been removed from a constraint, and at most t tuples can be removed. A removed tuple can be revised for each of its constraint’s incident subsopes. In the worst case, a constraint has $|\mathcal{C}| - 1$ neighbors in the dual graph, and each neighbor induces a unique subscope. Therefore, each tuple in the problem may cause REVISEBLOCK to be called $O((|\mathcal{C}| \cdot t) \cdot (\lceil \frac{t}{64} \rceil \cdot |\mathcal{C}| + |\sigma|))$ times. \square

Algorithm 6. PREPROCESS(C)	Runs CT and removes non-PWC tuples
<hr/>	
Input: A set of constraints C	
Output: Whether the current problem is consistent	
1 Run CT until quiescence	
2 if consistent then	
3	consistent \leftarrow InitPWC(C)
4	if consistent then
5	forall $c_i \in C$ do living(c_i).clearDelta()
6	consistent \leftarrow InitPWC(C)
7 return consistent	

PW-CT requires an additional initialization step to guarantee that preprocessing enforces fPWC. Consider the tuple $\tau = \{\langle A, 1 \rangle, \langle B, 1 \rangle, \langle C, 1 \rangle, \langle E, 1 \rangle\}$ in R_1 in Fig. 1. Each variable-value pair in τ has a GAC support in R_2 , but no PW-support. ENFORCEPWC operates on deleted tuples in order to propagate PW-support removals, but because τ ’s variable-value pairs are GAC, τ is not deleted by CT. Therefore, ENFORCEPWC is not called. To remedy this situation, all blocks that initially lack PW-supports need first to be removed from the problem. Once this removal is done, ENFORCEPWC can then evaluate the deleted tuples and propagate any other blocks that are emptied by their removal. Function PREPROCESS (Algorithm 6) accomplishes this operation by first enforcing GAC with CT and then calling function INITPWC (Algorithm 7).

Function INITPWC considers each subscope σ in the problem. It begins by finding the constraint c_s with the smallest number of living tuples incident to σ .

Algorithm 7. INITPWC(C) Partially enforces PWC on the constraints

Input: A set of constraints C **Output:** Whether the problem is consistent at preprocessing

```

1 for  $\sigma \in$  Subscopes do
2    $c_s \leftarrow$  constraint with fewest living tuples  $\in$  incidentCons[ $\sigma$ ]
3   foreach  $c_i \in$  incidentCons[ $\sigma$ ] do living( $c_i$ ).clearMask()
4   toCheck  $\leftarrow$  living( $c_s$ ) // Makes a copy
5   for  $\tau \in$  toCheck do
6     tuplePWC  $\leftarrow$  true
7     foreach  $c_i \in$  incidentCons[ $\sigma$ ] and tuplePWC do
8       if living( $c_i$ ).intersectIndex(CreateBlock( $c_i, \tau, \sigma$ )) = -1 then
9         tuplePWC  $\leftarrow$  false
10      if tuplePWC then
11        foreach  $c_i \in$  incidentCons[ $\sigma$ ] do
12          living( $c_i$ ).addBlockToMask(CreateBlock( $c_i, \tau, \sigma$ ));
13      toCheck.removeBlock(CreateBlock( $c_s, \tau, \sigma$ ))
14      foreach  $c_i \in$  incidentCons[ $\sigma$ ] do
15        living( $c_i$ ).intersectWithMask()
16        if living( $c_i$ ) was modified then
17          push( $c_i$ , PWCQueue)
18          push( $c_i$ , CTQueue)
19        if living( $c_i$ ).numSet = 0 then return false
20
21 return true

```

The algorithm checks if the blocks induced by the subscope σ for each tuple τ in living(c_s) has a PW-support in all constraints incident to σ (Lines 6 to 9). If the block is supported, then, for each constraint c_j incident to σ , we add the block of tuples induced by τ to the mask of living(c_j) (Line 12). After all blocks of c_s are processed, the masks of each RSparseBitSet living(c_j) contain only PW-supported tuples. Line 15 removes non-PWC tuples by calling the `intersectWithMask` method for each living(c_j). In practice, we found that in some problems the number of initially non-PWC tuples can be extremely large causing significant slowdown in the first call to `ENFORCEPWC` after `PREPROCESS`. To alleviate some of this burden, Function `PREPROCESS` runs twice `INITPWC`. The second call to `INITPWC` tends to remove fewer tuples than the first and guarantees the removal of any block that lost PW-supports due to the first call. Note that only the first call is strictly necessary for correctness.

6 Experiments

Our experiments run on 72 benchmarks of non-binary CSPs.⁵ Instances with intension and global constraints are converted to positive table constraints, resulting in a total of 2,210 instances. We evaluate and compare a total of 11 algorithms (Table 1) starting with the following: STR2 [14], STRBit [27], and CT [8] enforce GAC by table reduction; eSTR2 [15] enforces fPWC; eSTR2^w [15] enforces a weakened version of fPWC; STRBit+PW-AC is an unpublished hybrid algorithm that runs STRBit until quiescence before running PW-AC (greatly enhancing PW-AC’s effectiveness); and PW-CT enforces fPWC. Additionally, we evaluate eSTR2^m, eSTR2^{wm}, and PW-CT^m as variants of eSTR2, eSTR2^w, and PW-CT respectively, obtained on a minimal dual graph. Finally, Algorithm STRBit+PW-AC^{ms} is a variant of STRBit+PW-AC that operates on a minimal dual graph and propagates via subscopes. We test ordering heuristics dom/ddeg [2] and dom/wdeg [4],⁶ finding the first consistent solution with our custom solver. We limit each run to 7,200 s and 8GB of memory. When an algorithm times out, we add 7,200 s to the CPU time and indicate with a > sign that the time reported is a lower bound.

We perform a paired t-test between all algorithms with a significance level $\alpha = .05$. Under dom/ddeg, CT is the best algorithm with statistical significance. Most notably, we find that both PW-CT and PW-CT^m outperform all other eight algorithms, *including both* STR2 and STRBit. To our knowledge, PW-CT is the first fPWC algorithm to dominate a recent GAC algorithm. Exploiting a minimal dual graph improves, with statistical significance, *every* PWC algorithm tested. One exception occurs in the benchmark bddLarge, which times out while computing the minimal dual graph. Indeed, the number of unique, non-trivial subscopes in the problem is extremely large, averaging 1,123,484 over its instances.⁷ Surprisingly, PW-CT completes every instance in the benchmark, while all other PWC algorithms fail to complete even one instance. Not including bddLarge, we are able to load 1,707 instances of which 854 have non-trivial subscopes. The average number of unique subscopes on the full graph is 399.5, and 264.3 on the minimal. The remainder of the section discusses only the variants of the PWC algorithms exploiting a minimal dual graph given their superiority.

Table 1 compares the performance of the 11 algorithms. It provides the number of instances completed (#Cmpltd), the total CPU time (Σ CPU) over instances completed by at least one algorithm, the number of memouts (#MO), and the average number of node visits for instances completed by all algorithms (#NV). We exclude instances not solved by any algorithm.

PW-CT has almost the same number of memouts as GAC algorithms, testifying to its low memory consumption relative to all other PWC algorithms (34

⁵ <http://www.cril.univ-artois.fr/CPAI08/>.

⁶ Although dom/wdeg is generally more effective than dom/ddeg, the decisions made by dom/wdeg are considered too unstable to objectively allow comparing algorithms’ performance. Researchers studying the performance of HLC during search typically use dom/ddeg in their experiments [1, 20, 21].

⁷ Because bddLarge is an extreme outlier, we omit it from the results.

Table 1. Summary statistics of all tested instances

		dom/ddeg				dom/wdeg			
		#Cmpltd	Σ CPU (sec)	#MO	#NV	#Cmpltd	Σ CPU (sec)	#MO	#NV
71 Benchmarks tested with 2,210 total instances total									
GAC	CT	1,411	>449,421	65	2.90M	1,474	>338,246	65	1.65M
	STR2	1,284	>1,327,706	64	2.90M	1,355	>1,164,208	64	1.65M
	STRBit	1,370	>765,923	64	2.90M	1,445	>600,089	65	1.65M
fPWC	PW-CT	1,403	>579,112	65	0.99M	1,428	>715,885	65	0.82M
	PW-CT ^m	1,403	>567,500	65	0.99M	1,431	>696,622	65	0.82M
	STRBit+PW-AC	1,213	>1,738,628	137	0.99M	1,263	>1,752,986	139	0.84M
	STRBit+PW-AC ^{m,s}	1,247	>1,472,804	113	0.99M	1,290	>1,527,538	113	0.84M
	eSTR2	1,231	>1,750,990	102	0.99M	1,282	>1,769,454	102	0.83M
	eSTR2 ^m	1,248	>1,588,847	99	0.99M	1,295	>1,627,281	99	0.83M
wPWC	eSTR2 ^w	1,227	>1,784,866	102	1.01M	1,280	>1,769,529	102	1.1M
	eSTR2 ^{w,m}	1,243	>1,629,846	99	1.01M	1,294	>1,622,247	99	1.1M

to 72 fewer memouts). PWC algorithms that exploit a minimal dual graph incur fewer memouts than their original versions, testifying to the importance of using a minimal dual graph for PWC algorithms. When using dom/wdeg, the results of the pairwise t-tests are identical to dom/ddeg with the exception of STRBit and PW-CT. STRBit improves dramatically, beating both variants of PW-CT. Even so, it is clear from Table 1 that the performance of PW-CT is substantially closer to state-of-the-art GAC than to other algorithms used to enforce fPWC.

Figure 2 shows cumulative graphs of the number of instances completed for a given time using dom/ddeg: it compares the performance of PW-CT^m with the other GAC algorithms (left) and PWC algorithms (right). While CT wins, PW-CT^m is a close second, dominating the other two GAC algorithms. PW-CT^m clearly dominates all PWC algorithms by a large margin.

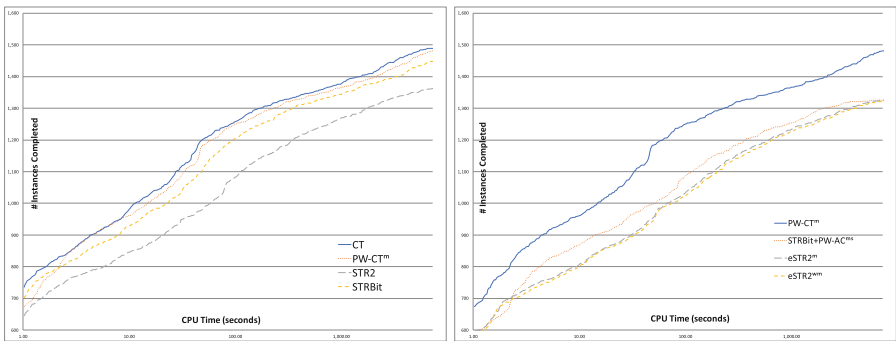
**Fig. 2.** PW-CT^m vs. GAC-based (left) and PWC algorithms (right) with dom/ddeg

Table 2 shows select benchmarks where PW-CT outperforms CT. These benchmarks were previously shown to benefit from enforcing PWC [20]. It is

Table 2. Select benchmarks using dom/ddeg (#I is the number of instances completed by at least one algorithm, and σ is the average number of non-trivial subsopes)

Benchmarks	#I	# σ	Σ CPU (sec)						
			CT	STR2	STRBit	PW-CT ^m	STRBit+PW-AC ^{ms}	eSTR2 ^m	eSTR2 ^{wm}
aim-100	24	146.9	>46,305	>54,997	>51,105	10,766	>19,078	>24,030	>37,687
aim-200	12	218.5	>34,208	>40,218	>35,594	69	121	253	>16,773
dubois	10	2.0	>29,801	>32,711	>31,363	10,798	>17,833	>20,061	>22,621
modRenault	50	61.6	2,553	>12,520	6,036	440	1,517	2,037	2,446
radar-8-24-3-2	1	113	2,621	3,450	3,452	2,612	>7,200	3,475	3,475

thus not surprising that those results hold for PW-CT. Notably, with the exception of the radar benchmark, all PWC algorithms outperform CT, emphasizing the usefulness of enforcing fPWC on difficult problems.

7 Conclusion

In this paper, we show that all PWC algorithms benefit from using a minimal dual graph to improve time and space cost and that the performance of PW-CT, our new algorithm for fPWC, is second only to CT, the best GAC algorithm.

References

- Balafrej, A., Bessière, C., Paparrizou, A.: Multi-armed bandits for adaptive constraint propagation. In: Proceedings of IJCAI 2015, pp. 290–296 (2015)
- Bessière, C., Régim, J.-C.: MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In: Freuder, E.C. (ed.) CP 1996. LNCS, vol. 1118, pp. 61–75. Springer, Heidelberg (1996). <https://doi.org/10.1007/3-540-61551-2-66>
- Bessière, C., Stergiou, K., Walsh, T.: Domain filtering consistencies for non-binary constraints. *Artif. Intell.* **172**, 800–822 (2008)
- Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: Proceedings of ECAI 2004, pp. 146–150 (2004)
- Briggs, P., Torczon, L.: An efficient representation for sparse sets. *ACM Lett. Program. Lang. Syst.* **2**(1–4), 59–69 (1993)
- le Clément, V., Schaus, P., Solnon, C., Lecoutre, C.: Sparse-sets for domain implementation. In: Proceedings of the CP Workshop on TRICS 2013 (2013)
- Dechter, R.: *Constraint Processing*. Morgan Kaufmann, Burlington (2003)
- Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régim, J.-C., Schaus, P.: Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In: Rueher, M. (ed.) CP 2016. LNCS, vol. 9892, pp. 207–223. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44953-1_14
- Gyssens, M.: On the complexity of join dependencies. *ACM Trans. Database Syst.* **11**(1), 81–108 (1986)
- Hentenryck, P.V., Deville, Y., Teng, C.M.: A generic arc consistency algorithm and its specializations. *Artif. Intell.* **57**, 291–321 (1992)
- Janssen, P., Jégou, P., Nougier, B., Vilarem, M.: A filtering process for general constraint-satisfaction problems: achieving pairwise-consistency using an associated binary representation. In: IEEE Workshop on Tools for AI, pp. 420–427 (1989)

12. Karakashian, S., Woodward, R., Choueiry, B.Y.: Improving the performance of consistency algorithms by localizing and bolstering propagation in a tree decomposition. In: Proceedings of AAAI 2013, pp. 466–473 (2013)
13. Karakashian, S., Woodward, R., Reeson, C., Choueiry, B.Y., Bessiere, C.: A first practical algorithm for high levels of relational consistency. In: Proceedings of AAAI 2010, pp. 101–107 (2010)
14. Lecoutre, C.: STR2: optimized simple tabular reduction for table constraints. *Constraints* **16**(4), 341–371 (2011)
15. Lecoutre, C., Paparrizou, A., Stergiou, K.: Extending STR to a higher-order consistency. In: Proceedings of AAAI 2013, pp. 576–582 (2013)
16. Likitvivanavong, C., Xia, W., Yap, R.H.C.: Higher-order consistencies through GAC on factor variables. In: O’Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 497–513. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_37
17. Likitvivanavong, C., Xia, W., Yap, R.: Decomposition of the factor encoding for CSPs. In: Proceedings of IJCAI 2015, pp. 353–359 (2015)
18. Mackworth, A.K.: Consistency in networks of relations. *Artif. Intell.* **8**, 99–118 (1977)
19. Mairy, J.-B., Deville, Y., Lecoutre, C.: Domain k-wise consistency made as simple as generalized arc consistency. In: Simonis, H. (ed.) CPAIOR 2014. LNCS, vol. 8451, pp. 235–250. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07046-9_17
20. Paparrizou, A., Stergiou, K.: Strong local consistency algorithms for table constraints. *Constraints* **21**(2), 163–197 (2016)
21. Paparrizou, A., Stergiou, K.: On neighborhood singleton consistencies. In: Proceedings of IJCAI 2017, pp. 736–742 (2017)
22. Perez, G., Régin, J.-C.: Improving GAC-4 for table and MDD constraints. In: O’Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 606–621. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_44
23. Samaras, N., Stergiou, K.: Binary encodings of non-binary constraint satisfaction problems: algorithms and experimental results. In: JAIR vol. 24, pp. 641–684 (2005)
24. Schneider, A., Woodward, R.J., Choueiry, B.Y., Bessiere, C.: Improving relational consistency algorithms using dynamic relation partitioning. In: O’Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 688–704. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_50
25. Ullmann, J.R.: Partition search for non-binary constraint satisfaction. *Inf. Sci.* **177**(18), 3639–3678 (2007)
26. Waltz, D.: Understanding line drawings of scenes with shadows. In: Winston, P. (ed.) *The Psychology of Computer Vision*, pp. 19–91. McGraw-Hill Inc., New York City (1975)
27. Wang, R., Xia, W., Yap, R.H.C., Li, Z.: Optimizing simple tabular reduction with a bitwise representation. In: Proceedings of IJCAI 2016, pp. 787–793 (2016)



Automatic Generation and Selection of Streamlined Constraint Models via Monte Carlo Search on a Model Lattice

Patrick Spracklen, Özgür Akgün^(✉), and Ian Miguel

School of Computer Science, University of St Andrews, St Andrews, UK
{j1ps,ozgur.akgun,ijm}@st-andrews.ac.uk

Abstract. Streamlined constraint reasoning is the addition of uninferred constraints to a constraint model to reduce the search space, while retaining at least one solution. Previously it has been established that it is possible to generate streamliners automatically from abstract constraint specifications in ESSENCE and that effective combinations of streamliners can allow instances of much larger scale to be solved. A shortcoming of the previous approach was the crude exploration of the power set of all combinations using depth and breadth first search. We present a new approach based on Monte Carlo search over the lattice of streamlined models, which efficiently identifies effective streamliner combinations.

1 Introduction and Background

If the performance of a constraint model is found to be inadequate, a natural step is to consider adding constraints to the model in order to assist the constraint solver in detecting dead ends in search and therefore reducing overall search effort. One approach is to add *implied* constraints, which can be inferred from the initial model and are therefore guaranteed to be sound. Effective implied constraints have been found both by hand [17, 18] and via automated methods [10, 11, 19]. If implied constraints cannot be found, or improve performance insufficiently, for satisfiable problems¹ a more aggressive step is to add *streamliner* constraints [22], which are not guaranteed to be sound but are designed to reduce significantly the search space while permitting at least one solution. For several problem classes, effective streamliners have been found by hand by looking for patterns in the solutions of small instances of those classes [22, 24, 26, 27].

Wetter et al. [40] demonstrated how to generate effective streamliners automatically from the specification of a constraint problem class in the abstract constraint specification language ESSENCE [14–16]. This method, which we expand upon, exploits the structure apparent in an ESSENCE specification, such as that of the Progressive Party Problem (Fig. 1), to produce candidate streamliners via

¹ Streamlining is unsuitable for unsatisfiable problems: streamliners are not necessarily sound, so exhausting the search space does not prove unsatisfiability (a case split approach is possible: a sub-problem with a streamliner, another with its negation).

```

1  language Essence 1.3
2  given n_boats, n_periods : int(1..)
3  letting Boat be domain int(1..n_boats)
4
5  find hosts : set (minSize 1) of Boat,
6     sched : set (size n_periods) of function (total) Boat --> Boat
7  minimising |hosts|
8
9  such that
10 $ Hosts remain the same throughout the schedule
11 forAll p in sched . range(p) = hosts,
12 $ Hosts stay on their own boat
13 forAll p in sched . forAll h in hosts . p(h) = h,
14 $ Hosts have the capacity to support the visiting crews
15 forAll p in sched . forAll h in hosts .
16     (sum b in preImage(p,h) . crew(b)) <= capacity(h),
17 $ No two crews are at the same party more than once
18 forAll b1,b2 : Boat .
19     b1 < b2 -> (sum p in sched . toInt(p(b1) = p(b2))) <= 1

```

Fig. 1. The Progressive Party Problem [36] in ESSENCE. There are two abstract decision variables, a set of host boats, and a set of functions from boats to boats representing the assignment of guests to hosts in each period. From this very concise, structured statement of the problem, 160 candidate streamliners can be generated by our system.

a set of streamliner generation rules. An effective streamliner that we automatically generate for this problem class constrains approximately half of the entries in the `sched` set variable to be monotonically increasing functions.

Using training instances drawn from the problem class under consideration, streamliner candidates are evaluated via a toolchain consisting of the automated constraint modelling tools CONJURE [1–4] and SAVILE ROW [31–33], and the constraint solver MINION [21]. Promising candidates, which retain at least one solution to the training instances while significantly reducing search, are used to solve more difficult instances from the same problem class. Candidate streamliners are often most effectively used in combination. For example, Wetter et al. presented an effective combination of three streamliners for the Van der Waerden numbers problem. Hence, the space of streamlined models forms a lattice where the root is the original ESSENCE specification and an edge represents the addition of a streamliner to the combination associated with the parent node.

A shortcoming of Wetter et al.’s work is the uninformed manner in which the streamliner lattice is explored using depth- or breadth-first search. Our principal contribution is a new method for exploring the lattice via Monte Carlo-style search, allowing more effective streamlined models to be found in a time budget. A second contribution is a set of new streamliner generator rules for sequence and matrix ESSENCE type constructors to complement those presented by Wetter et al. Finally, we demonstrate the efficacy of our approach on a variety of problems.

2 Essence Specifications and Streamliner Generators

An ESSENCE specification such as that presented in Fig. 1 identifies: the input parameters of the problem class (**given**), whose values define an instance; the combinatorial objects to be found (**find**); the constraints the objects must satisfy (**such that**); identifiers declared (**letting**); and an (optional) objective function (**min/maximising**). The key feature of the language is its support for abstract decision variables, such as set, multiset, relation and function, as well as *nested* types, such as the set of functions found in Fig. 1.

A concise, structured specification of a problem class directly supports the generation of powerful candidate streamliners: in the example, it is readily apparent that the problem requires us to find a set of boats and a set of functions assigning guest boat crews to hosts. Hence, streamliners related to sets and functions, such as that given in the introduction, can be generated straightforwardly. In contrast an equivalent constraint model in, for example, MiniZinc [30] or ESSENCE PRIME [32] has to represent these abstract decision variables with constrained collections of more primitive (e.g. integer domain) variables, such as the matrix model [12, 13] proposed by Smith et al. [36]. In this context, it is significantly more difficult to recognise the structure (i.e. the set and set of functions) in the problem and generate the equivalent streamliners.

Wetter et al. present a set of streamliner generation rules for the ESSENCE type constructors set, function and partition, as well as simple integer domains [40]. Our first contribution is to extend this set also to cover sequence and matrix type constructors. These are summarised in Fig. 2. For decision variables with

Name	matrix all
Param	R (another rule)
Input	X: matrix indexed by [I] of _
Output	forall i : I . R(X[i])
Name	matrix most (similarly for matrix half and matrix approximately half)
Param	R (another rule)
Input	X: matrix indexed by [I] of _
Output	I / 2 <= sum i : I . toInt (R(X[i]))
Name	sequence monotonically increasing (similarly for monotonically decreasing)
Input	X: sequence of _
Output	forall i, j in defined (X) . i < j -> X(i) <= X(j)
Name	sequence smallest first (similarly for largest first)
Input	X: sequence of _
Output	forall i in defined (X) . X(min (defined (X))) <= X(i)
Name	sequence on defined (similarly for range)
Param	R (another rule)
Input	X: sequence of _
Output	R(defined (X))

Fig. 2. Streamliner generators for sequence and matrix domains.

matrix domains, one generator (matrix all) takes another streamliner generator as a parameter (R , as a simple example: constrain an integer variable to take an even value) and *lifts* it to operate on all entries in a matrix. This rule can be applied to higher dimensional matrix domains as well, in which case the multi-dimensional matrix domain is interpreted in the same way as a series of nested one-dimensional matrix domains. The generators matrix most (and matrix half and matrix approximately half) operate in a similar way. In contrast to the matrix all streamliner generator, these generators first reify the result of applying R , and then restrict the number of places the constraint must hold.

For sequence domains, we present two sets of first-order streamliner generators: monotonically increasing (or decreasing) and smallest (or largest) first. These do not take another generator as a parameter but directly post constraints on the sequence decision variable. The sequence on range and sequence on defined generators take an existing streamliner generator as a parameter and lift it to work on the range or the defined set of the sequence domain respectively.

3 Monte Carlo Search for Streamliner Combinations

Le Bras et al. [25] and Wetter et al. [40] both observed that by applying several streamlining constraints to a model simultaneously the search required for finding the first solution can be reduced further than by applying the streamliners in isolation. Finding an effective streamliner combination involves searching the streamliner lattice, the size of which is determined by the power set of all candidate streamliners for a given problem class. Table 1 presents the number of candidate streamliners our current set of generation rules produces for a number of problem classes. In some cases the number of candidates generated is small. However, the cost of evaluating each combination on a set of test instances means that it is typically not feasible to evaluate all possible streamliner combinations.

Wetter et al. employed breadth-first and depth-first search to explore the streamliner lattice in an uninformed manner. The motivation for our work is the hypothesis that a best-first search can allow more effective streamliner combinations to be discovered within a given time budget. Our approach is to focus the search onto areas of the lattice where the streamliners combine to give the greatest reduction in search while retaining at least one solution.

For a given problem class, we have no prior knowledge of the performance of the set of candidate streamliners, either individually or in combination. This raises the issue of the exploration/exploitation problem: if we can identify a combination of streamliners that performs well, should we try and exploit that combination further by evaluating the addition of further streamliners, or should we explore other parts of the lattice that may at present seem less promising?

The exploration/exploitation tradeoff can be formalised in several reinforcement learning variants, including via markov decision processes [7]. We model this situation as a multi-armed bandit problem [5], which allows us to employ regret minimising algorithms to deal with the exploration/exploitation dilemma. The multi-armed bandit can be seen as a set of real distributions, each distribution being associated with the rewards delivered by one of the K levers.

Since the multi-armed bandit problem assumes that each lever corresponds to an independent action, in order to use it directly we would have to associate a lever with each point in the streamliner lattice, which is infeasible in general. Instead, we use the bandit algorithm to guide the exploration of the lattice in a process reminiscent of Monte Carlo Tree Search (MCTS) [8], as described below.

3.1 Algorithm Outline

Our algorithm has the same four basic steps as MCTS. It uses Upper Confidence bound applied to Trees (UCT) [8] to balance exploration and exploitation.

1. **Selection:** Starting at the root node, the UCT policy is recursively applied to traverse the explored part of the lattice until an unexpanded, but expandable node is reached. A node is expandable if it has at least one child that is not marked as pruned (Sect. 3.5). A child node is selected to maximise:

$$UCT = X_j + 2C_p \sqrt{\frac{2 \ln n}{n_j}}$$

where n is the number of times the current (parent) node in the lattice has been visited, n_j the number of times child j has been visited, X_j is the mean reward associated with child j and $C_p > 0$ is a constant [8].

2. **Expansion:** Enumerate the children of the Selected node and choose one to expand according to the heuristic explained in Sect. 3.4.
3. **Simulation:** The collection of streamliners associated with the expanded node are evaluated using the CONJURE, SAVILE ROW, and MINION toolchain.
4. **BackPropagation:** The result of the evaluation is propagated back up through the lattice to update reward values, as explained below.

3.2 Back Propagation

Since our search is operating over a lattice, a node may have multiple parents. This requires an alteration to the back propagation employed in MCTS: when we perform back propagation that reward value is back propagated up all paths from that node to the root. To illustrate consider a problem with two streamliners {A,B} and we are back propagating from a node in the lattice representing the combination {AB}. There are two paths by which this node could have been reached, {root → A → B} and {root → B → A}. Although the algorithm will have only descended one of these paths, because the reward value of a node in the lattice is representative of the ability of the streamliner combination represented by that node to combine and produce effective reductions in search, the node in the lattice that represents streamliner combination {B} should also receive this reward. For this reason both paths are rewarded accordingly and the reward generated is back propagated up all paths from that node to the root. We also ensure that a node that lies on more than one such path is rewarded only once. The cost of back propagation thus grows exponentially with depth.

However, since each level of the lattice represents an additional constraint it is unlikely that satisfiability is maintained at great enough depths for this to become an issue. Empirically, the cost is insignificant relative to solving the training instances.

We must also consider the situation where a node in a path back to the root has not yet been expanded. If we ignore such nodes, their true reward is not reflected in their reward values because all reward values back propagated from child nodes prior to their creation are lost. Our approach is that when a node is expanded, it absorbs the reward value and visit count of its immediate children that have already been expanded. This avoids caching a potentially large set of values while maintaining reward values for nodes around the focus of our search.

3.3 Simulation Reward

The performance of our best first search algorithm is heavily reliant on how the reward is produced from the simulation step. Initially we assigned rewards as follows: if the majority of the instances evaluated are unsatisfiable a reward of -1 is back-propagated, otherwise a reward of one minus the average reduction in search space (expressed in search nodes) is returned. While this is valid, our initial experiments revealed that its effect was to produce a search strategy similar to breadth-first search - i.e. too strong a bias towards exploration.

The reason for this is that the penalty value is too punitive when evaluating larger streamliner combinations. Intuitively, the penalty should be sensitive to the depth we have travelled into the lattice: as we add streamliners we reduce the search space and we expect the probability of such failure to rise. Therefore we divide the penalty value by the depth of the node being evaluated, allowing the prolonged exploration of promising paths.

3.4 Expansion Heuristic

The order of expansion of child nodes is an important factor in performance. An expanded child consumes time to perform simulation and, because the simulation reward is back propagated, if a penalty is awarded it can affect the likelihood of the parent node being selected on the next iteration. During the expansion phase of our algorithm child nodes are expanded in descending likelihood of the application of the associated streamliner combination resulting in a satisfiable problem. In order to facilitate this, when a successful simulation is performed, for a representative instance the solution found is stored, along with the approximate size of its search space (via the product of the domains of the decision variables in the model) and the proportion of the space explored to find the solution.

Upon expansion all potential children are enumerated and for each we check whether the additional associated streamliner invalidates the solution stored at the expanding parent. If the solution remains valid then the child is preferred for expansion because we know pre-simulation that the associated streamliner combination satisfies at least one instance and the additional streamliner might reduce search. If the solution is invalidated then the search space explored by the child is smaller than the expanding parent. We use the proportion of search space explored to find the solution associated with the expanding parent to estimate the likelihood of a solution existing in that subspace. Intuitively, if the parent explored a large fraction of the space then it is less likely that a solution will be found when adding the streamliner associated with the child node.

3.5 Pruning the Streamliner Lattice

As per Sect. 3.2, when a simulation for a streamliner combination reveals that the majority of training instances are unsatisfiable, a penalty is back propagated up the lattice. We also mark the node associated with the simulation as pruned and never consider any of its children for expansion. In addition, we prune nodes whose additional streamliner is determined to be *redundant* in combination with those inherited from the expanding parent, in the sense that it causes no further reduction in search on the evaluated instances. Pruning the lattice by these two methods typically reduces the number of nodes to be expanded very significantly.

4 Empirical Evaluation

We evaluate two hypotheses empirically. First, that the best-first search is more effective in exploring the streamliner lattice than the simpler depth- and breadth-first search methods employed in [40]. Second, that our method is able to automatically find streamliner combinations that drastically reduce the search space across a variety of problem classes.

We experiment with thirteen problem classes, eight from Wetter et al. and the remainder selected for variety, particularly problem classes requiring significantly more instance data such as SONET [28]. Streamlining can aid in the search for feasible solutions of optimisation problems, but not the proof of optimality. Hence, in our experiments we transformed optimisation into decision problems by the standard method of bounding the objective and searching for a satisfying solution. The results we obtain are very positive, as presented in Table 1.

Table 1. For each of the thirteen problem classes, fifteen instances were split 70/30 into training and test sets. All three methods received the same training budget of six hours per problem class, a cost which is amortised over the entire problem class for which the streamliners are applicable. We record the number of candidate streamliners and the mean time to solution for the test instances using non-streamlined models in columns 2 and 3. For the substantial majority of the problem classes the most effective streamliner discovered is composed of a combination of individual streamlining constraints, as presented in columns 4, 6 and 8. In these cases the Monte Carlo search method is able to discover larger combinations that yield superior results. For the problem classes where a single streamliner was found to be the most effective all methods are equally effective, as through pruning (Sect. 3.5) we were able to exhaustively search the space of all streamliner combinations. Mean reduction for both time and search nodes is measured by comparing the search required to find the first solution on the non-streamlined model with the model streamlined with the most effective streamliner combination found using the respective search method. The selected streamliners for each problem class retained satisfiability on all test instances. Through the addition of streamlining constraints we obtain a uniformly vast reduction in both search nodes and time. All computational experiments were run on a 32-core AMD Opteron 6272 at 2.1 GHz and an 8 core Intel Core i7-6920HQ at 2.90 GHz. All experiments for an individual problem class were run on a single machine.

Experimental data, models, raw results, and source code can be downloaded from: <http://github.com/stacs-cp/cp2018-streamlining>

Problem Class	Number of Candidate Streamliners	Mean Instance Time (Seconds)	DFS		BFS		Monte Carlo		
			Combination size	Mean Reduction in Search Nodes	Combination size	Mean Reduction in Search Nodes	Combination size	Mean Reduction in Search Time	
Progressive Party [35]	160	76	1	48.2%	2	41.5%	4	93.2%	91.7%
MFB [8]	76	54	2	72.7%	2	83.1%	3	96.2%	93.5%
Schur's Lemma [38]	65	254	1	39.2%	1	81.4%	3	92.3%	89.8%
Quasigroup Existence [34]	17	337	1	83.0%	1	83.0%	1	83.0%	87.4%
Van Der Waerden Numbers [37]	65	142	3	84.4%	2	82.3%	4	96.2%	95.7%
Graceful Double Wheel Graphs [26]	72	572	2	65.6%	3	81.0%	4	94.2%	90.2%
Graceful Gears [28]	72	493	2	64.6%	2	84.9%	4	95.5%	89.1%
Graceful Helms [6]	72	598	2	85.4%	2	80.6%	3	91.8%	87.1%
Graceful Wheel Graphs [19]	72	12	2	69.5%	2	67.4%	4	88.3%	80.8%
Car Sequencing [33]	36	724	1	31.9%	1	31.9%	1	31.9%	37.9%
FFPA [22]	144	231	1	86.1%	1	86.1%	1	86.1%	85.7%
SONET [27]	64	678	1	72.0%	2	87.2%	3	94.6%	95.8%
CVRP [36]	84	817	1	93.0%	1	93.0%	1	93.0%	84.1%

5 Conclusion

We have presented a new method for the automated generation of streamlined constraint models from a large set of candidates via Monte Carlo search. Our method is efficient in searching the space of candidates, producing more effective streamlined models in less time than less informed approaches. Our empirical results demonstrate a vast reduction in search across a variety of benchmarks.

As part of future work we plan to explore the generation of streamlined versions of alternative models generated by CONJURE. We expect the utility of particular streamlining constraints to vary depending on the model.

Acknowledgements. This work was supported via EPSRC EP/P015638/1. We thank our anonymous reviewers for helpful comments.

References

1. Akgün, Ö.: Extensible automated constraint modelling via refinement of abstract problem specifications. Ph.D. thesis, University of St Andrews (2014)
2. Akgun, O., Frisch, A.M., Gent, I.P., Hussain, B.S., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: Automated symmetry breaking and model selection in CONJURE. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 107–116. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_11
3. Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Breaking conditional symmetry in automated constraint modelling with Conjure. In: ECAI, pp. 3–8 (2014)
4. Akgün, Ö., Miguel, I., Jefferson, C., Frisch, A.M., Hnich, B.: Extensible automated constraint modelling. In: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, pp. 4–11. AAAI Press (2011)
5. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.* **47**(2), 235–256 (2002). <https://doi.org/10.1023/A:1013689704352>
6. Ayel, J., Favaron, O.: Helms are graceful. In: Progress in Graph Theory (Waterloo, Ont., 1982), pp. 89–92. Academic Press, Toronto (1984)
7. Bellman, R.: Dynamic Programming and Markov Processes. JSTOR (1961)
8. Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P.I., Tavener, S., Perez, D., Samothrakis, S., Colton, S., et al.: A survey of Monte Carlo tree search methods. *IEEE Trans. Comput. Intell. AI* **4**(1), 1–43 (2012)
9. Čagalj, M., Hubaux, J.P., Enz, C.: Minimum-energy broadcast in all-wireless networks: Np-completeness and distribution issues. In: Proceedings of the 8th Annual International Conference on Mobile Computing and Networking, pp. 172–182. ACM (2002)
10. Charnley, J., Colton, S., Miguel, I.: Automatic generation of implied constraints. *ECAI* **141**, 73–77 (2006)
11. Colton, S., Miguel, I.: Constraint generation via automated theory formation. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 575–579. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45578-7_42
12. Flener, P., Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Matrix modelling. In: Proceedings of the CP-01 Workshop on Modelling and Problem Formulation, p. 223 (2001)

13. Flener, P., Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Matrix modelling: exploiting common patterns in constraint programming. In: Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems, pp. 27–41 (2002)
14. Frisch, A.M., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The essence of essence. In: Modelling and Reformulating Constraint Satisfaction Problems, p. 73 (2005)
15. Frisch, A.M., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The design of essence: a constraint language for specifying combinatorial problems. *IJCAI* **7**, 80–87 (2007)
16. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: a constraint language for specifying combinatorial problems. *Constraints* **13**(3), 268–306 (2008)
17. Frisch, A.M., Jefferson, C., Miguel, I.: Symmetry breaking as a prelude to implied constraints: a constraint modelling pattern. In: *ECAI*, vol. 16, p. 171 (2004)
18. Frisch, A.M., Miguel, I., Walsh, T.: Symmetry and implied constraints in the steel mill slab design problem. In: Proceedings of CP01 Workshop on Modelling and Problem Formulation (2001)
19. Frisch, A.M., Miguel, I., Walsh, T.: CGRASS: a system for transforming constraint satisfaction problems. In: O’Sullivan, B. (ed.) *CologNet 2002*. LNCS, vol. 2627, pp. 15–30. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36607-5_2
20. Frucht, R.: Graceful numbering of wheels and related graphs. *Ann. N. Y. Acad. Sci.* **319**(1), 219–229 (1979)
21. Gent, I.P., Jefferson, C., Miguel, I.: Minion: a fast scalable constraint solver. *ECAI* **141**, 98–102 (2006)
22. Gomes, C., Sellmann, M.: Streamlined constraint reasoning. In: Wallace, M. (ed.) *CP 2004*. LNCS, vol. 3258, pp. 274–289. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30201-8_22
23. Huczynska, S., McKay, P., Miguel, I., Nightingale, P.: Modelling equidistant frequency permutation arrays: an application of constraints to mathematics. In: Gent, I.P. (ed.) *CP 2009*. LNCS, vol. 5732, pp. 50–64. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_7
24. Kouril, M., Franco, J.: Resolution tunnels for improved SAT solver performance. In: Bacchus, F., Walsh, T. (eds.) *SAT 2005*. LNCS, vol. 3569, pp. 143–157. Springer, Heidelberg (2005). https://doi.org/10.1007/11499107_11
25. Le Bras, R., Gomes, C.P., Selman, B.: Double-wheel graphs are graceful. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, *IJCAI 2013*, pp. 587–593. AAAI Press (2013). <http://dl.acm.org/citation.cfm?id=2540128.2540214>
26. Le Bras, R., Gomes, C.P., Selman, B.: On the Erdős discrepancy problem. In: O’Sullivan, B. (ed.) *CP 2014*. LNCS, vol. 8656, pp. 440–448. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_33
27. LeBras, R., Gomes, C.P., Selman, B.: Double-wheel graphs are graceful. In: *IJCAI*, pp. 587–593 (2013)
28. Lee, Y., Sherali, H.D., Han, J., Kim, S.I.: A branch-and-cut algorithm for solving an intraring synchronous optical network design problem. *Networks* **35**(3), 223–232 (2000)
29. Ma, K., Feng, C.: On the gracefulness of gear graphs. *Math. Pract. Theor.* **4**, 72–73 (1984)

30. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_38
31. Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I.: Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. In: O’Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 590–605. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_43
32. Nightingale, P., Akgün, O., Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in Savile Row. *Artif. Intell.* **251**, 35–61 (2017). <https://doi.org/10.1016/j.artint.2017.07.001>
33. Nightingale, P., Spracklen, P., Miguel, I.: Automatically improving SAT encoding of constraint problems through common subexpression elimination in Savile Row. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 330–340. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_23
34. Parrello, B.D., Kabat, W.C., Wos, L.: Job-shop scheduling using automated reasoning: a case study of the car-sequencing problem. *J. Autom. Reason.* **2**(1), 1–42 (1986)
35. Slaney, J., Fujita, M., Stickel, M.: Automated reasoning and exhaustive search: quasigroup existence problems. *Comput. Math. Appl.* **29**(2), 115–132 (1995)
36. Smith, B.M., Brailsford, S.C., Hubbard, P.M., Williams, H.P.: The progressive party problem: integer linear programming and constraint programming compared. *Constraints* **1**(1–2), 119–138 (1996)
37. Toth, P., Vigo, D.: The vehicle routing problem. In: SIAM (2002)
38. van der Waerden, B.: Beweis einer Baudetschen Vermutung. *Nieuw Arch. Wisk.* **19**, 212–216 (1927)
39. Walsh, T.: CSPLib problem 015: Schur’s lemma. <http://www.csplib.org/Problems/prob015>
40. Wetter, J., Akgün, Ö., Miguel, I.: Automatically generating streamlined constraint models with ESSENCE and CONJURE. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 480–496. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_34



Efficient Methods for Constraint Acquisition

Dimosthenis C. Tsouros^(✉), Kostas Stergiou, and Panagiotis G. Sarigiannidis

Department of Informatics and Telecommunications Engineering,
University of Western Macedonia, Kozani, Greece
{dtsouros,kstergiou,psarigiannidis}@uowm.gr

Abstract. Constraint acquisition systems such as QuAcq and MultiAcq can assist non-expert users to model their problems as constraint networks by classifying (partial) examples as positive or negative. For each negative example, the former focuses on one constraint of the target network, while the latter can learn a maximum number of constraints. Two bottlenecks of the acquisition process where both these algorithms encounter problems are the large number of queries required to reach convergence, and the high cpu times needed to generate queries, especially near convergence. We propose methods that deal with both these issues. The first one is an algorithm that blends the main idea of MultiAcq into QuAcq resulting in a method that learns as many constraints as MultiAcq does after a negative example, but with a lower complexity. The second is a technique that helps reduce the number of queries significantly. The third is based on the use of partial queries to cut down the time required for convergence. Experiments demonstrate that our resulting algorithm, which integrates all the new techniques, does not only generate considerably fewer queries than QuAcq and MultiAcq, but it is also by far faster than both of them, both in average query generation time and in total run time.

Keywords: Constraint acquisition · Learning · Modeling

1 Introduction

Constraint programming (CP) has progressed significantly over the last decades, and is now considered as one of the foremost paradigms for solving combinatorial problems. However, a major bottleneck in the use of CP is modeling. Expressing a combinatorial problem as a constraint network requires considerable expertise [1]. To overcome this obstacle, several techniques have been proposed for modeling a constraint problem automatically [2]. Along these lines, an area of research that has started to attract a lot of attention is that of *constraint acquisition* where the model of a constraint problem is acquired (i.e. learned) using a set of examples that are posted to a human user or to a software system [3].

Constraint acquisition can come in various flavours depending on factors such as whether the learner can post queries to the user dynamically, and the type of queries that can be posted and answered. In *passive* acquisition, examples

of solutions and non-solutions are provided by the user. Based on these examples, the system learns a set of constraints that correctly classifies all the given examples [3–7]. In *active* acquisition, the learner interacts with the user dynamically while acquiring the constraint network [3, 8, 9]. In such a scenario, the acquisition system proposes examples to the user to classify as solutions or non-solutions. Such questions are called membership queries [10], and this is the class of queries that has received the most attention.

A state-of-the-art interactive acquisition algorithm is QuAcq [11]. QuAcq is able to ask the user to classify partial queries, which may be easier for the user to answer. Also, asking partial queries gives the system the capability to focus on the scope of a constraint that is violated and hence learn the constraint. If the answer to a membership query posted by QuAcq is positive, the system reduces the search space by removing the set of constraints violated by this example. If the answer is negative, QuAcq asks a series of partial queries to locate the scope of one of the violated constraints of the target network. An attempt to make QuAcq more efficient was presented by Arcangioli et al., with the MultiAcq algorithm [12]. Instead of focusing on the scope of only one constraint, MultiAcq finds all the scopes of constraints of the target network violated by a generated example that is classified as negative.

Active acquisition has several advantages. First of all, it decreases the number of examples necessary to converge to the target set of constraints. In addition, it does not require the existence of diverse examples of solutions or non-solutions to the problem. This is a significant advantage especially if the problem has not already been solved. Another advantage is that the user does not need to be human. It might be a previous system developed to solve the problem. For example, active learning to automatically acquire CSPs which model the elementary actions of a robot by asking queries to the simulator of the robot [13].

However, active learning still presents computational challenges regarding the number of queries required and the cpu time needed to generate them. It has been shown that the number of membership queries needed to converge can be exponentially large [14]. Furthermore, despite the good theoretical bound of QuAcq and QuAcq-like approaches in terms of number of queries, the generation of a membership query is an NP-hard problem [3]. Hence, it can be too time-consuming, and therefore annoying, if the system interacts with a human user. For example, QuAcq can take more than 30 min to generate a query for the model acquisition of Sudoku puzzles near convergence.

In this paper, we propose methods to deal with both of these challenges. We first introduce an algorithm, called MQuAcq, that blends the main idea of MultiAcq into QuAcq. This algorithm uses the reasoning of QuAcq when searching for constraints to learn once a negative query is encountered, but instead of focusing on one constraint, it learns a maximum number of constraints, just like MultiAcq does. But whereas MultiAcq learns constraints of the target network in a number of queries linear in the size of the example, our proposed approach finds constraints in a logarithmic number of queries. Experiments demonstrate that MQuAcq outperforms both QuAcq and MultiAcq.

Our second method is an optimization on the process of locating scopes that helps reduce the number of queries significantly. Our third method generalizes the idea of allowing partial queries to be posted to the user. Instead of using partial queries only when trying to focus on one or more constraints after a complete example has been classified as negative, we allow the generation of partial queries as examples to be posted to the user. We use an optimization method to generate such queries, and as experiments demonstrate, this can reduce the time needed for the system to converge, resulting in avoidance of premature convergence and reduced total run time.

Experimental results with benchmark problems demonstrate that the integration of our methods results in an algorithm that considerably outperforms both QuAcq and MultiAcq as it generates significantly fewer queries, it is up to one order of magnitude faster in average query generation time, and by far superior in total run time.

The rest of this paper is organized as follows. Section 2 presents the necessary background on constraint acquisition. In Sect. 3 we review the basics of QuAcq and MultiAcq. Section 4 describes the new methods that we propose. Experimental results are reported in Sect. 5. Section 6 concludes the paper.

2 Background

The *vocabulary* (X, D) is a finite set of n variables $X = \{X_1, \dots, X_n\}$ and a domain $D = \{D(x_1), \dots, D(x_n)\}$, where $D(X_i) \subset \mathbb{Z}$ is the finite set of values for X_i . The vocabulary is the common knowledge shared by the user and the constraint acquisition system.

A *constraint* c is a pair $(\text{rel}(c), \text{var}(c))$, where $\text{var}(c) \subseteq X$ is the *scope* of the constraint and $\text{rel}(c)$ is a relation between the variables in $\text{var}(c)$ that specifies which of their assignments are allowed. $|\text{var}(c)|$ is called the *arity* of the constraint. A *constraint network* is a set C of constraints on the vocabulary (X, D) . $C[Y]$, where $Y \subseteq X$, denotes the set of constraints from C whose scope is a subset of Y . Besides the vocabulary, the learner has a *language* Γ consisting of *bounded arity* constraints.

An example e_Y is an assignment on a set of variables $Y \subseteq X$. e_Y is rejected by a constraint c iff $\text{var}(c) \subseteq Y$ and the projection $e_{\text{var}(c)}$ of e_Y on the variables in the scope $\text{var}(c)$ of the constraint is not in $\text{rel}(c)$. An assignment e_Y is a partial solution iff it is accepted by all the constraints in $C[Y]$. A complete assignment that is accepted by all the constraints in C is a solution to the problem. $\text{sol}(C)$ denotes the set of solutions of C . A partial assignment e_Y which is accepted by $C[Y]$ is not necessarily part of a complete solution. A *redundant* or *implied* constraint c in C is a constraint that if removed from the constraint network the set of solutions $\text{sol}(C)$ remains the same. In other words, if all the other constraints in C are satisfied then c is also satisfied.

Using terminology from machine learning, a *concept* is a Boolean function over D^X , that assigns to each example $e \in D^X$ a value in $\{0, 1\}$, or in other words classifies it as positive or negative. The target concept f_T is a concept

that assigns 1 to e if e is a solution of the problem and 0 otherwise. In constraint acquisition, the target concept is the target constraint network C_T , such that $sol(C_T) = \{e \in D^X \mid f_T(e) = 1\}$. The *constraint bias* B is a set of constraints on the vocabulary (X, D) , built using the constraint language Γ , from which the system can learn the target constraint network. $\kappa_B(e_Y)$ represents the set of constraints in B that reject e_Y .

The classification question asking the user to determine if an example e_X is a solution to the problem that the user has in mind is called a *membership query* $ASK(e)$. The answer to a membership query is positive if $f_T(e) = 1$ and negative otherwise. A *partial query* $ASK(e_Y)$, with $Y \subseteq X$, asks the user to determine if e_Y , which is an assignment in D^Y , is a partial solution or not. A classified assignment e_Y is labelled as positive or negative depending on the answer of the user to $ASK(e_Y)$. Following the literature, we assume that all queries are answered correctly by the user.

In interactive constraint acquisition the system 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 labelled as positive in E and reject those labelled as negative. The *learned network* C_L has to agree with E .

A (complete or partial) query $q = e_Y$ is called *irredundant* (or *informative*) iff the answer to q is not predictable. That is, q is irredundant iff it is not classified as positive by all the constraints in the bias B , which means that $\kappa_B(e_Y)$ is not empty. At the same time, q should be accepted by the learned network C_L otherwise it will be classified as negative.

The acquisition process has *converged* on the learned network $C_L \subseteq B$ iff C_L agrees with E and for every other network $C \subseteq B$ that agrees with E , we have $sol(C) = sol(C_L)$. If the first condition is true (C_L agrees with E) but the second condition has not been proved, we have *premature convergence*. If there does not exist a constraint network $C \subseteq B$ such that C agrees with E then the acquisition has *collapsed*. This happens when the target constraint network is not included in the bias, i.e. $C_T \not\subseteq B$.

3 Algorithms for Constraint Acquisition

QuAcq (Algorithm 1) iteratively generates a complete assignment e which satisfies the currently built C_L and is rejected by at least one constraint in B (line 4). Then e is posted as a membership query to the user. If e is classified as positive then all constraints that violate it are removed from B (line 6). If e is negative, the algorithm tries to find one constraint that is violated by e to add to C_L by calling functions *FindC* and *FindScope* (line 8). Once the system learns the constraint (line 10), if no collapse occurs, it returns to the query generation step.

Once a generated example is classified as negative, QuAcq calls the recursive function *FindScope* (Algorithm 2) to discover the scope of one of the violated constraints. *FindScope* takes as parameters an example e that violates one or more constraints from the Bias, two sets of variables R and Y , initialized to the empty set and to X respectively, and a Boolean variable *ask_query*.

Algorithm 1. QuAcq: Quick Acquisition**Input:** B, X, D (B : the bias, X : the set of variables, D : the set of domains)**Output:** C_L : a constraint network

```

1:  $C_L \leftarrow \emptyset$ ;
2: while true do
3:   if  $sol(C_L) = \emptyset$  then return “collapse”;
4:   Generate  $e$  in  $D^X$  accepted by  $C_L$  and rejected by  $B$ ;
5:   if  $e = \text{nil}$  then return “ $C_L$  converged”;
6:   if  $ASK(e) = \text{yes}$  then  $B \leftarrow B \setminus \kappa_B(e)$ ;
7:   else
8:      $c \leftarrow FindC(e, FindScope(e, \emptyset, X, false))$ ;
9:     if  $c = \text{nil}$  then return “collapse”;
10:    else  $C_L \leftarrow C_L \cup \{c\}$ ;

```

An invariant of *FindScope* is that the example e violates at least one constraint whose scope is a subset of $R \cup Y$. If *FindScope* is called with $ask_query = \text{true}$ it asks the user if e_R is positive or not (line 3). If the answer is yes, it removes all the constraints from the bias that reject e_R . Otherwise, it returns the empty set (line 3). *FindScope* reaches line 5 only in the case where e_R does not violate any constraint. Hence, because as mentioned above e violates at least one constraint whose scope is a subset of $R \cup Y$, if Y is a singleton, the variable it contains surely belongs to the scope of a constraint that is violated. In this case the function returns Y .

If none of the return conditions is satisfied, the set Y is split in two balanced parts (line 6) and the algorithm searches recursively, in sets of variables built using these parts, for the scope of a violated constraint, in a logarithmic number of steps (lines 7–9).

Function *FindScope* posts a partial query to the user until it finds the scope of a constraint that is violated. A potential deficiency is the fact that if a question to the user violates, say 3 constraints, and the answer was negative, then after removing some variables from Y , if the partial query is still rejecting 3 constraints, *FindScope* will ask the user to classify the partial query again. However, there is no need for this because it is certain that the partial query will still be classified as negative. In Sect. 4.2 we propose a fix to this problem.

After the system has located the scope of a violated constraint, it calls function *FindC*, described in [11] but not included here due to space limitations, to find the violated constraint.

Given that the generation of an irredundant membership query e is an NP-hard problem, and that there may be several constraints from the target network that are violated by e , it is very likely that the system can learn more information from a negative query. This is what MultiAcq tries to do. MultiAcq generates an example like QuAcq and then the function *FindAllScopes* is called to learn a maximum number of constraints violated by the specific example [12].

Algorithm 2. *FindScope***Input:** e, R, Y, ask_query (e : the example, R, Y : sets of variables, ask_query : boolean)**Output:** *Scope* : a set of variables, the scope of a constraint in C_T

```

1: function FindScope( $e, R, Y, ask\_query$ )
2:   if  $ask\_query$  then
3:     if  $ASK(e_R) = \text{yes}$  then  $B \leftarrow B \setminus \kappa_B(e_R)$ ;
4:     else return  $\emptyset$ ;
5:   if  $|Y| = 1$  then return  $Y$ ;
6:   split  $Y$  into  $\langle Y_1, Y_2 \rangle$  such that  $|Y_1| = \lceil |Y|/2 \rceil$ ;
7:    $S_1 \leftarrow FindScope(e, R \cup Y_1, Y_2, true)$ ;
8:    $S_2 \leftarrow FindScope(e, R \cup S_1, Y_1, (S_1 \neq \emptyset))$ ;
9:   return  $S_1 \cup S_2$ ;

```

The recursive function *FindAllScopes* takes as input a complete example e and a subset of variables Y (X for the first call). It asks the user to classify the example and if the answer is “no”, there still exist violated constraints in Y . Therefore, *FindAllScopes* is called on each subset of Y built by removing one variable from Y . If in all of these calls the answer of the user is “yes” then Y is the scope of a violated constraint and it is added to the set of found scopes. Function *FindC* is then called to find the constraint, like in QuAcq.

In addition to the above, which are described in the relevant papers, QuAcq and MultiAcq take some extra steps, especially during query generation¹. Specifically, in line 4 of QuAcq (respectively in MultiAcq), when trying to find a solution of C_L violating at least one constraint from the bias, there is a cutoff imposed. If no example is found within this time limit then the system visits the constraints in B one by one, and for each constraint c it tries to solve the problem comprising $C_L \cup \neg c$. A second cutoff is then used for this process. This is done both in QuAcq and MultiAcq. This means that if the second cutoff is triggered for all constraints in B , premature convergence has occurred because for every other network $C \subseteq B$ that agrees with E , it has not been proved that $sol(C) = sol(C_L)$. Another optimization concerning MultiAcq that is not mentioned in [12] but has been implemented by the authors is that *FindC* is called on the fly each time a scope is found, whereas in [12] all scopes are first returned and then *FindC* is called.

4 Efficient Constraint Acquisition

4.1 Multi-QuAcq

We first describe a new algorithm, called Multi-QuAcq (MQuAcq for short) which takes logarithmic time to discover all the violated constraints, achieving the benefits of both QuAcq and MultiAcq. MQuAcq (Algorithm 3) is an active learning algorithm which is based on QuAcq and extends it by incorporating the

¹ Personal communication with the authors of the algorithms.

basic idea of MultiAcq. The main difference between QuAcq and MQuAcq is the fact that QuAcq finds one explanation (constraint) of why the user classified an example as negative, whereas MQuAcq tries to learn all the violated constraints. This is done by calling function *FindScope* (Algorithm 2) iteratively while reducing the search space, removing variables from the scopes already found. The main difference with MultiAcq is that the proposed approach uses the QuAcq search method to find each scope through function *FindScope*, and in this way avoids some redundant searches (which can be very time-consuming) and queries that MultiAcq makes with function *FindAllScopes*.

Algorithm 3. The MQuAcq Algorithm

Input: B, X, D (B : the bias, X : the set of variables, D : the set of domains)

Output: C_L : a constraint network

```

1:  $C_L \leftarrow \emptyset$ ;
2: while true do
3:   Scopes.clear();
4:   if  $\text{sol}(C_L) = \emptyset$  then return “collapse”;
5:   Generate  $e$  in  $D^X$  accepted by  $C_L$  and rejected by  $B$ ;
6:   if  $e = \text{nil}$  then return “ $C_L$  converged”;
7:   if  $\neg \text{findAllCons}(e, X, 0)$  then return “collapse”;

```

MQuAcq starts by initializing the C_L network to the empty set (line 1) and then it enters the main loop (line 2). The array *Scopes*, which is initialized to be empty in line 3, is used within function *FindAllCons* as explained below. If C_L is unsatisfiable, the algorithm collapses (line 4). Otherwise, a complete assignment e is generated (line 5), satisfying C_L and violating at least one constraint in B . If such an example does not exist then we have converged (line 6). Otherwise, function *findAllCons* is called to find all the constraints that are violated by the example e . If *findAllCons* return false then have we collapsed (line 7).

The recursive function *FindAllCons* is presented in Algorithm 4. It takes as parameters an example e , a set of variables Y and an integer variable s , which is a counter/identifier for the scopes. It returns false if collapse has occurred and true otherwise. *FindAllCons* adds to C_L all the constraints that are violated by e in Y . It uses the array *Scopes* to store all the scopes of the constraints that have been found in the current generated query. The main idea is to search for partial queries in the given example that do not contain any constraint already found, so that the answer will not be predictable. To achieve this, from each scope S already found we make $|S|$ partial examples, one for each variable $x_i \in S$, with each such example involving variables $Y' = Y \setminus \{x_i\}$. The identifier s helps us to decide on which scope we have to branch. When a partial example that violates no constraint already learned but at least one from B is found, *FindAllCons* uses *FindScope*, as in QuAcq, to learn a constraint from C_T .

In the first call to *FindAllCons* s is equal to 0 so that a query is immediately posted to the user in line 7. The condition of line 2 will be false because

Algorithm 4. findAllCons**Input:** e, Y, s (e : the example, Y : set of variables, s : scopes identifier)**Output:** *not_collapsed* : returns false if collapsed, true otherwise

```

1: function findAllCons( $e, Y, s$ )
2:   if  $\kappa_{B \setminus C_L}(e_Y) = \emptyset$  then return true;
3:   if  $s < |Scopes|$  then
4:     for  $x_i \in Scopes[s]$  do
5:       if  $\neg findAllCons(e, Y \setminus \{x_i\}, s + 1)$  then return false;
6:   else
7:     if  $ASK(e_Y) = \text{yes}$  then  $B \leftarrow B \setminus \kappa_B(e_Y)$ ;
8:     else
9:        $scope \leftarrow FindScope(e, \emptyset, Y, false)$ ;
10:       $c \leftarrow FindC(e, scope)$ ;
11:      if  $c = \text{nil}$  then return false;
12:      else  $C_L \leftarrow C_L \cup \{c\}$ ;
13:       $Scopes.push(scope)$ ;
14:      if  $\neg findAllCons(e, Y, s)$  then return false;
15:   return true;

```

e is generated in such a way that it violates at least one constraint in B (i.e. $|\kappa_{B/C_L}(e_Y)| > 0$). If the example is negative then a constraint is sought using *FindScope* and then the recursive calls to *FindAllCons* start. If the example is positive, the constraints in B that reject it are removed and true is returned.

In any subsequent call to *FindAllCons* we start by checking if there exists any violated constraint in B to learn, not already in C_L . If not, it is implied that $ASK(e_Y) = \text{yes}$ and the function returns true (line 2), because we assume that the bias is expressive enough to learn a C_L equivalent to the target network C_T . This check is important because as the recursive calls to *FindAllCons* remove variables from Y (as explained below), we may end up in a case where e_Y is surely positive and no search for a violated constraint is needed. This is because if $ASK(e_Y) = \text{yes}$ then for every $Y' \subseteq Y$ we surely have $ASK(e_{Y'}) = \text{yes}$. With this check the algorithm avoids a lot of redundant searches, reducing the number of nodes in the tree of recursive calls, and also avoids asking redundant queries.

After that, *FindAllCons* checks if s is smaller than the size of *Scopes*. As mentioned, s is a counter/identifier acting as an id of scopes already visited. If we have not branched on all the scopes already found ($s < |Scopes|$), it means that we still have a violating scope in the query and therefore we avoid asking a partial query which includes $Scopes[s]$ because we know that the answer will be “no”. So we call *FindAllCons* recursively on each subset of Y created by removing one of the variables of the scope at position s of *Scopes* (lines 4–5), and increasing s by 1 to continue with the next scope.

In the case that s is equal to the size of *Scopes* it means that we have finished with branching and we have a partial example that does not contain the scope of any violated constraint already learned. Hence, there must exist a partial query e_Y that violates at least one constraint of B (otherwise the algorithm

would have returned at line 2) and no violated constraint already found exists in Y . Therefore, the system asks again the user to classify the partial example (line 7). If the answer is positive then the constraints in B that reject e are removed. Otherwise, function *FindScope* is called to find the scope of one of the violated constraints (line 9). *FindC* will then select a constraint from B with the discovered scope that is violated by e (line 10). If no constraint is found then the algorithm collapses. Otherwise, the constraint returned by *FindC* is added to C_L (line 12) and its scope is added to the array of found scopes (line 13). Now, we call again *FindAllCons* to continue searching in the partial examples created by removing the variables of the scope the function has just found.

We now illustrate the behavior of *FindAllCons* with a simple example. We denote by c_{ij} the constraint between variables X_i and X_j .

Example 1. Consider a problem consisting of n variables and a complete example e generated in line 5 of MQuAcq. Suppose that the constraints from C_T that are violated by e and we have not already learned are c_{12} , c_{13} and c_{34} . In the first call (call 0) to *FindAllCons*, e will be posted as a query to the user. After the user answers “no”, the algorithm will find the constraint c_{12} using functions *FindScope* and *FindC*. Next, *FindAllCons* will be called to continue searching for the remaining constraints that violate e . In the next call (call 1) we have $Y = X$, $s = 0$ and $|Scopes| = 1$ (*Scopes* includes $\{X_1, X_2\}$). As $s < |Scopes|$, we know that the answer to $ASK(e_Y)$ will be “no”. So *FindAllCons* will be called recursively on each subset of Y built by removing one variable from the scope $Scopes[0]$ (i.e. $\{X_1, X_2\}$), and increasing s by 1.

In the first recursive call (call 1.1) we have $Y' = Y \setminus \{X_1\}$ and $s = 1$. As $s = |Scopes|$, it means that we have branched on all scopes found until now. Hence, the query $e_{Y'}$ will be posted to the user and the constraint c_{34} will be learned because it is the only constraint among the variables Y' that violates C_T . In the next call (call 2) of *FindAllCons* in line 14 no further constraint will be found. So we go back to the second call of line 5 (call 1.2). We have $Y' = Y \setminus \{X_2\}$ and $s = 1$. $|Scopes| = 2$, so we have another scope in which we have to branch. Hence, *FindAllCons* will be called recursively on each subset of Y' built by removing one variable from the scope $Scopes[1]$, and increasing s by 1. In call 1.2.1, we have $Y'' = Y' \setminus \{X_3\}$ and $s = 2$. Because no constraint from C_T is violated the answer from the user will be “yes” and true will be returned. In call 1.2.2, we have $Y'' = Y' \setminus \{X_4\}$ and $s = 2$. Because $s = |Scopes|$ the query will be asked to the user and then the constraint c_{13} will be learned.

MultiAcq learns a constraint of the target network in a number of queries linear in the size of the example whereas MQuAcq finds a constraint in a logarithmic number of queries, using the *FindScope* function as is QuAcq. We now analyse the complexity of MQuAcq in terms of the number of queries it asks to the user.

Theorem 1. *Given a bias B built from a language Γ , with bounded arity constraints, and a target network C_T , MQuAcq uses $O(|C_T| * (\log|X| + |\Gamma|))$ queries to find the target network or to collapse and $O(|B|)$ queries to prove convergence.*

Proof. (sketch) We know that a scope of a constraint from C_T is found in at most $|S| * \log|Y|$ queries with the function *FindScope*, with $|S|$ being the arity of the scope and $|Y|$ the size of the example given to the function [11]. As $Y \subseteq X$, *FindScope* needs at most $|S| * \log|X|$ queries to find a scope, because in MQuAcq, in the worst case, only one constraint from B will be violated by any complete example. Also, *FindC* needs at most $|\Gamma|$ queries to find a constraint from C_T in the scope it takes as parameter, if one exists [11]. If none exists, the system collapses. Hence, the number of queries necessary to find a constraint is $O(|S| * \log|X| + |\Gamma|)$, and the number of queries for finding all the constraints in C_T or collapsing is at most $C_T * (|S| * \log|X| + |\Gamma|)$ which is $O(C_T * (\log|X| + |\Gamma|))$ because $|S|$ is bounded. Convergence is proved when B is empty or contains only redundant constraints. Constraints are removed from B when the answer from the user is yes in a query. In the case that the example generated by the algorithm in line 5, contains only one constraint from B , it leads to at least one constraint removal in each query. This gives a total of $O(|B|)$ queries to converge. \square

The complexities of QuAcq and MultiAcq to find the target network are $O(|C_T| * (\log|X| + |\Gamma|))$ and $O(|C_T| * (|X| + |\Gamma|))$ respectively. Hence, we achieve the same bound as QuAcq but a better one than MultiAcq, while discovering all the violated constraints from a negative example.

4.2 FindScope-2

We now describe an optimization to function *FindScope*, aiming at asking fewer queries to the user. This results in a function we simply call *FindScope-2*, which can be used instead of *FindScope* either inside QuAcq or inside our new algorithm MQuAcq to cut down the number of generated queries.

Let us first consider the example illustrated in Table 1 for motivation. In this example we have a problem consisting of the set of variables X_1, \dots, X_5 , and the language $\Gamma = \{\neq\}$. The bias B contains all the possible \neq constraints between the five variables.

Table 1. Example illustrating a deficiency of *FindScope*

Call	R	Y	ASK
0	\emptyset	X_1, X_2, X_3, X_4, X_5	-
1	X_1, X_2, X_3	X_4, X_5	No

For the purpose of this example, the target network C_T is not important. Assume that QuAcq is applied to this problem and suppose that the first example, which was classified as negative by the user, is $e_1 = \{1, 1, 1, 4, 5\}$. The constraints from B that it violates are $X_1 \neq X_2, X_1 \neq X_3, X_2 \neq X_3$. After the first call to *FindScope*, R is equal to X_1, X_2, X_3 , so the partial example e_R that is then asked to the user is $e_2 = \{1, 1, 1\}$. But the constraints from B that are violated are still $X_1 \neq X_2, X_1 \neq X_3, X_2 \neq X_3$. Therefore, this partial example is negative, and there is no point in posting it to the user.

To avoid such redundant queries made by *FindScope*, we modify this function (see Algorithm 5) adding a check that inspects if the number of violated constraints from the bias is the same as in the last query asked. This is implemented using a global variable *rej* to store this number. This check is done in line 3. If this is the case, it is implied that the answer will still be no and therefore we return the empty set. Before the first call to *FindScope*, *rej* must be initialized to the number of constraints from *B* that are violated by the complete query.

As a further improvement to *FindScope*, given the assumption that the bias is expressive enough to learn C_T , in cases where $|\kappa_B(e_R)| = 0$ (i.e. there is no violated constraint in *B*), it is implied that $\text{ASK}(e_R) = \text{yes}$. So another check is performed in line 2. If the bias is not expressive enough to learn C_T , the system will collapse later, because it will not find any constraint to learn.

Algorithm 5. FindScope-2

Input: *e*, *R*, *Y*, *ask_query* (*e*: the example, *R*, *Y*: sets of variables, *ask_query*: boolean)

Output: *Scope* : a set of variables, the scope of a constraint in C_T

```

1: function FindScope-2(e, R, Y, ask_query)
2:   if ask_query AND  $|\kappa_B(e_R)| > 0$  then
3:     if rej  $\neq$   $|\kappa_B(e_R)|$  then
4:       if  $\text{ASK}(e_R) = \text{yes}$  then  $B \leftarrow B \setminus \kappa_B(e_R)$ ;
5:       else
6:          $rej \leftarrow |\kappa_B(e_R)|$ ;
7:       return  $\emptyset$ ;
8:     else return  $\emptyset$ ;
9:   if  $|Y| = 1$  then return Y;
10:  split Y into  $\langle Y_1, Y_2 \rangle$  such that  $|Y_1| = \lceil |Y|/2 \rceil$ ;
11:   $S_1 \leftarrow \text{FindScope}(e, R \cup Y_1, Y_2, \text{true})$ ;
12:   $S_2 \leftarrow \text{FindScope}(e, R \cup S_1, Y_1, (S_1 \neq \emptyset))$ ;
13:  return  $S_1 \cup S_2$ ;

```

4.3 Exploiting Partial Queries

Another step of the acquisition process that can be improved is the crucial initial query generation step, which is included in all of the proposed algorithms (e.g. line 4 in Algorithm 1 and line 5 in Algorithm 3). In this section we propose an approach to query generation that helps improve any acquisition algorithm.

Let us first note that although both QuAcq and MultiAcq allow for the use of partial queries to focus on the violated constraint(s) after an example has been classified as negative, they both always try to generate complete examples at the start of the query generation iteration (e.g. line 4 in Algorithm 1). However, generating a complete example requires finding a complete variable assignment that satisfies all constraints in C_L and violates at least one constraint in *B*. Given that solving C_L is in itself an NP-hard problem, the process can be very time-consuming.

Experimental results that we have obtained with both QuAcq and MultiAcq demonstrate that when no time limit to the query generation process is set then both algorithms can take several minutes (more than 30 min) to generate a query as convergence is approached. This of course is unacceptable from the user's point of view, and therefore a time limit is necessary for the practical application of the algorithms. However, setting any time limit to the query generation process means that the algorithm may reach *premature convergence* (e.g. line 5 of Algorithm 1).

Another relevant issue is that of proving convergence in problems that contain redundant constraints. As the system cannot always know beforehand if some of the constraints in the bias are redundant, proving that no solution of C_L violates at least one constraint in B can be very time-consuming in the presence of redundant constraints. This is because if near the end of the process B is left with redundant constraints only, no solution of C_L can violate any of these constraints, simply because these constraints, being implied, will be surely satisfied.

Given the importance of query generation in the acquisition process, it is of primary importance that it is executed as efficiently as possible, and in a way such that the problem of premature convergence is avoided as much as possible. Towards this, we propose to exploit partial queries at this step of the process. Both QuAcq and MultiAcq assume that the user, be it human or machine, is able to answer partial queries, so there is no reason to limit the use of partial queries to the case where a complete query has been classified as negative.

Our proposal is to model the query generation problem as an optimization problem in which we seek to find a (partial) assignment of the variables that maximizes the number of violated constraints in B . We call this heuristic max_B . This is related to but is not the same as the max heuristic that was used within QuAcq [11]. The max heuristic tries to generate a complete solution of C_L that violates a maximum number of constraints from B . Hence, given a time limit, which is necessary for any algorithm to run in reasonable times as explained above, max will focus on finding complete assignments that satisfy all the constraints in C_L and violate as many as possible from B , while max_B will focus on violating as many constraints as possible from B without necessarily building a complete variable assignment.

Although the difference between max_B and max may not seem substantial, experimental results given below show the use of max_B largely alleviates the danger of premature convergence and can have a significant impact on the total run time of the acquisition algorithm.

5 Experimental Evaluation

To evaluate our proposed methods, we ran some experiments on a system carrying an Intel(R) Core(TM) i5-4690K CPU @ 3.50 GHz with 8 Gb of RAM. We compared the proposed methods to both QuAcq and MultiAcq, which were implemented as efficiently as possible using the strategies described in [11, 12], as

well as the additional heuristics described to us through private communication with the authors. To be precise:

- We implemented the *max* and *max_B* heuristics, with dom/wdeg for variable ordering and random value ordering, for the generation of the queries. For both heuristics we set a cutoff of 1 s, returning the best example found within this time limit, if any.
- In all our methods, and also in QuAcq and MultiAcq, we used the additional cutoff of 5 s to the query generation step that was described in Sect. 3. That is, if no example is found within this time limit, the system takes one by one each constraint c in B and tries to solve $C_L \cup \neg c$ with a second cutoff of 5 s.
- To maximize the performance of MultiAcq we used the heuristic proposed in [12]: A cutoff of 5 s is used in function *FindAllScopes*. After triggering the cutoff for the first time, *FindAllScopes* is called again on the same complete example with a reverse order of the variables. If a second cutoff is triggered, we generate a new example and shuffle the variables' order.

We used the following benchmarks in our study:

Sudoku. The Sudoku puzzle is a 9×9 grid. It must be completed in such a way that all the rows, all the columns and the 9 non overlapping 3×3 squares contain the numbers 1 to 9. The *vocabulary* for this problem has 81 variables and domains of size 9. The target network has 810 binary \neq constraints on rows, columns and squares. The bias was initialized with 12.960 binary constraints from the language $\Gamma = \{=, \neq, >, <\}$.

Latin Square. The Latin square problem consists of an $n \times n$ table in which each element occurs once in every row and column. In this problem, we have 100 (i.e. $n = 10$) variables with domains of size 10. The target network has 900 binary \neq constraints on rows and columns. The system was initialized with a bias of 19.800 binary constraints created from the language $\Gamma = \{=, \neq, >, <\}$.

Zebra. The Zebra problem consists of 25 variables of domain size of 5. The target network contains 50 \neq constraints and 12 additional constraints given in the description of the problem. The bias was initialized with 1200 binary constraints from the language $\Gamma = \{=, \neq, >, <, x_i - x_j = 1, |x_i - x_j| = 1\}$.

Murder. This problem consists of 20 variables with domains of size 5. The target network contains 4 cliques of \neq constraints and 12 additional binary constraints. The bias was initialized with 760 constraints based on the language $\Gamma = \{=, \neq, >, <\}$.

5.1 Results

In our experiments we measure the size of the learned network C_L , the average waiting time \bar{T} (in secs) for the user, the total number of queries $\#q$, the average size \bar{q} of all queries, the number of complete queries $\#q_c$, the time $T_{queries}$ taken from the start of the process until the last query and the total time needed (to converge) T_{total} . The difference between T_{total} and $T_{queries}$ is the time needed to prove convergence or to reach premature convergence (because of the cutoffs). The size of C_L in some cases is smaller than the size of the target network C_T due

to the presence of redundant constraints that some methods learn and others do not. Finally, we counted the times each method triggers any of the two cutoffs.

To validate our results, we implemented the tested methods, including QuAcq and MultiAcq, using the Choco solver, as well as our own solver written in C++². Results were very similar, with our solver being slightly faster. Therefore, in the following we report results obtained using our solver. In Table 2 we evaluate our methods both individually and combined, and we compare them against QuAcq and MultiAcq. Hence, we give results from QuAcq, MultiAcq, MQuAcq, QuAcq with *FindScope-2* instead of *FindScope*, QuAcq with max_B instead of max , and MQuAcq with *FindScope-2* and max_B (i.e. all our methods combined). Each algorithm was run 10 times and the means are presented.

Table 2. Comparative results

Benchmark	Algorithm	$ C_L $	\bar{T}	$\#q$	\bar{q}	$\#q_c$	$T_{queries}$	T_{total}
Sudoku	QuAcq	648	0.063	11567	35	659	734.65	1555.34
	MultiAcq	797	0.050	14361	10	37	752.49	822.56
	QuAcq + FindScope 2	648	0.113	6054	43	658	686.26	1506.88
	QuAcq max_B	810	0.060	14240	31	534	865.35	865.41
	MQuAcq	800	0.008	14735	27	40	117.96	173.01
	MquAcq + FindScope 2 max_B	810	0.012	6713	32	15	84.99	85.00
Latin	QuAcq	855	0.062	15515	46	869	956.23	1186.46
	MultiAcq	898	0.119	20628	11	49	2471.12	2486.15
	QuAcq + FindScope 2	855	0.123	7993	56	870	983.13	1212.14
	QuAcq max_B	900	0.059	16226	44	789	96	956.01
	MQuAcq	898	0.008	17812	36	45	142.25	157.27
	MquAcq + FindScope 2 max_B	900	0.013	8401	45	19	113.77	113.78
Zebra	QuAcq	60	0.066	789	11	60	51.93	51.95
	MultiAcq	59	0.393	953	6	8	374.77	374.78
	QuAcq + FindScope 2	60	0.105	485	12	60	51.09	51.10
	QuAcq max_B	60	0.066	800	11	59	53.38	53.39
	MQuAcq	60	0.004	790	8	8	3.44	3.45
	MquAcq + FindScope 2 max_B	60	0.007	503	9	6	3.92	3.92
Murder	QuAcq	52	0.080	609	9	52	48.98	49.14
	MultiAcq	52	0.008	680	5	9	5.97	6.13
	QuAcq + FindScope 2	52	0.138	355	10	52	49.25	49.40
	QuAcq max_B	52	0.084	608	8	47	51.13	51.42
	MQuAcq	52	0.008	634	7	7	5.50	5.67
	MquAcq + FindScope 2 max_B	52	0.017	370	8	3	6.23	6.47

Looking at the performance of *FindScope-2* when used inside QuAcq, we can see that the number of queries asked were significantly lower compared to standard QuAcq with *FindScope* because the former avoids asking several redundant queries. In terms of number of queries *FindScope-2* gives a gain of

² The only exception is the max_B heuristic which was only implemented in our solver.

38% on the zebra problem, 42% on murder, and 48% on sudoku and latin square. Interestingly, it seems that the more variables are present in a problem, the bigger is the gain in avoided queries. As a downside, *FindScope-2* increases the average query generation time, but not the total time required to converge.

Focusing on the use of max_B , we observe that in small problems (murder and zebra) we have similar performance to max . This is because such problems are easy, meaning that in most cases both max_B and max can find complete solutions to C_L that violate many constraints in B within the time limit. In contrast, in sudoku and latin square, there are differences. As we can see from column $|C_L|$, max_B helps to not only learn the complete target network, but also redundant constraints. This results in more queries being asked and greater $T_{queries}$. But on the other hand, we gain in the total running time (T_{total}) because, having learned the redundant constraints during the process, B is empty in the end, and therefore the system does not have to prove that no solution of C_L violates them. In addition, with max_B we have fewer complete queries and smaller average query size because near convergence, when it is time-consuming to find a complete assignment, max_B returns a partial assignment.

Focusing on our main contribution, i.e. MQuAcq, and comparing it to QuAcq, we observe that the use of *FindAllCons* to learn all the violated constraints from a negative example reduces significantly the average waiting time per query and the total time of the execution. QuAcq is 9 times slower in sudoku, 7.5 times in latin square, 15 times in zebra and 8.5 times in murder. This is due to the fewer generations of new examples in line 5 of MQuAcq, because the algorithm is able to learn a maximum number of violated constraints from each negative example. This is validated by looking at column $\#q_c$, which shows that far fewer complete queries are generated. As a downside, MQuAcq requires more queries in total than QuAcq to converge. However, the average size of the queries is smaller.

Comparing MQuAcq to MultiAcq, it is clear that the redundant searches MultiAcq makes greatly affect the average time per query and total time needed. MQuAcq needs far less time to ask a query to the user, and requires posting fewer queries to converge, in most problems.

Regarding the cutoffs, neither of the two cutoffs was triggered by any method on zebra or murder. On sudoku (resp. latin square), QuAcq triggered the first cutoff twice on average (resp. once), and the second 170 times (resp. 46). The corresponding numbers for MultiAcq were 11 (resp. 16) and 29 (resp. 27), and for MQuAcq 10 (resp. 14) and 28 (resp. 25). Importantly, any method that used max_B never trigger a cutoff. Hence, max_B can help alleviate the problem of premature convergence.

Finally, the results obtained from MQuAcq with *FindScope-2* and max_B (i.e. integrating all our methods) confirm our intuitions. The integrated method is by far superior to both QuAcq and MultiAcq considering all the important metrics, especially on the harder problems. It cuts down the total number of queries by 50% or even more, and generates fewer complete queries than even MultiAcq, it significantly reduces the average query waiting time for the user, and it is up to one order of magnitude faster (or even more) in total run time.

6 Conclusion

We have presented new methods that can boost the performance of constraint acquisition systems. Our main algorithm, MQuAcq, extends QuAcq to discover all the violated constraints from a negative example, just like MultiAcq does, but with a better complexity bound. Our other methods help reduce the number of queries and the time required to reach convergence. Experimental results demonstrate that an algorithm which integrates all our methods significantly outperforms the state-of-the-art algorithms on all the important metrics.

References

1. Freuder, E.C.: Modeling: the final frontier. In: The First International Conference on The Practical Application of Constraint Technologies and Logic Programming (PACLP), London, pp. 15–21 (1999)
2. De Raedt, L., Passerini, A., Teso, S.: Learning constraints from examples. In: Proceedings in Thirty-Second AAAI Conference on Artificial Intelligence (2018)
3. Bessiere, C., Koriche, F., Lazaar, N., O’Sullivan, B.: Constraint acquisition. *Artif. Intell.* **244**, 315–342 (2017)
4. Bessiere, C., Coletta, R., Freuder, E.C., O’Sullivan, B.: Leveraging the learning power of examples in automated constraint acquisition. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 123–137. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30201-8_12
5. Bessiere, C., Coletta, R., Koriche, F., O’Sullivan, B.: A SAT-based version space algorithm for acquiring constraint satisfaction problems. In: Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M., Torgo, L. (eds.) ECML 2005. LNCS (LNAI), vol. 3720, pp. 23–34. Springer, Heidelberg (2005). https://doi.org/10.1007/11564096_8
6. Lallouet, A., Lopez, M., Martin, L., Vrain, C.: On learning constraint problems. In: 2010 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI), vol. 1, pp. 45–52. IEEE (2010)
7. Beldiceanu, N., Simonis, H.: A model seeker: extracting global constraint models from positive examples. In: Milano, M. (ed.) CP 2012. LNCS, pp. 141–157. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33558-7_13
8. Freuder, E.C., Wallace, R.J.: Suggestion strategies for constraint-based matchmaker agents. In: Maher, M., Puget, J.-F. (eds.) CP 1998. LNCS, vol. 1520, pp. 192–204. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49481-2_15
9. Bessiere, C., Coletta, R., O’Sullivan, B., Paulin, M., et al.: Query-driven constraint acquisition. *IJCAI* **7**, 50–55 (2007)
10. Angluin, D.: Queries and concept learning. *Mach. Learn.* **2**(4), 319–342 (1988)
11. Bessiere, C., Coletta, R., Hebrard, E., Katsirelos, G., Lazaar, N., Narodytska, N., Quimper, C.G., Walsh, T., et al.: Constraint acquisition via partial queries. *IJCAI* **13**, 475–481 (2013)
12. Arcangioli, R., Bessiere, C., Lazaar, N.: Multiple constraint acquisition. In: *IJCAI: International Joint Conference on Artificial Intelligence*, pp. 698–704 (2016)
13. Paulin, M., Bessiere, C., Sallantin, J.: Automatic design of robot behaviors through constraint network acquisition. In: 20th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2008, vol. 1, pp. 275–282. IEEE (2008)
14. Bessiere, C., Koriche, F.: Non learnability of constraint networks with membership queries. Technical report, Technical report, Coconut, Montpellier, France (2012)



A Circuit Constraint for Multiple Tours Problems

Philippe Vismara^{1,2(✉)} and Nicolas Briot¹

¹ LIRMM, Univ Montpellier, CNRS, Montpellier, France

² MISTEA, Montpellier SupAgro, INRA, Univ Montpellier, Montpellier, France
philippe.vismara@supagro.fr

Abstract. Routing problems appear in many practical applications. In the context of Constraint Programming, circuit constraints have been successfully developed to handle problems like the well-known Traveling Salesman Problem or the Vehicle Routing Problem. These kind of constraints are linked to the search for a Hamiltonian circuit in a graph. In this paper we consider a more general multiple tour problem that consists in covering a part of the graph with a set of minimal cost circuits. We define a new global constraint `WEIGHTEDSUBCIRCUITS` that generalizes the `WeightedCircuit` constraint by releasing the need to obtain a Hamiltonian circuit. It enforces multiple disjoint circuits of bounded total cost to partially cover a weighted graph, the subsets of vertices to be covered being induced by external constraints. We show that enforcing Bounds Consistency for `WEIGHTEDSUBCIRCUITS` is NP-hard. We propose an incomplete but polynomial filtering method based on the search for a lower bound of a weighted Steiner circuit.

1 Introduction

Many real problems can be modeled as a tour problem, the best known is the Travelling Salesman Problem (TSP). It consists in finding a Hamiltonian cycle (i.e., passing by each vertex of a graph once) with a minimum weight. Many works coming from Integer Linear Programming (ILP) or dynamic programming allow to quickly solve large instances of TSP. In addition, some variations of the TSP, such as the Vehicle Routing problem (VRP), have been the subject of numerous studies proposing effective solving methods [1].

In this context, Constraint Programming has for a long time offered its expressiveness to address variations of TSP. Initially limited to small instances, the most recent filtering algorithms allow to compete with ILP approaches on complex problems where complementary constraints restrict TSP solutions.

These good results are related to the definition of global constraints and the associated filtering algorithms: the constraint `CYCLE` (or `CIRCUIT`) enforces covering the graph with one circuit visiting all vertices once; the constraint `WEIGHTEDCIRCUIT` imposes, in addition, that the sum of the costs of the edges of the circuit is lower than a cost variable. As enforcing arc consistency for these constraints is generally NP-hard, the filterings used are inevitably incomplete but

in practice relatively efficient. For instance, the `WEIGHTEDCIRCUIT` constraint propagator can incorporate different methods based on TSP relaxation from literature.

However, all these constraints are linked to the search for a Hamiltonian cycle. Many real problems require searching for one or more cycles covering all or part of the vertices of the graph. These problems correspond to two kinds of relaxation in the definition of the Hamiltonian cycle. The classical relaxation is the VRP where some vertices (depots) can be visited several times. This problem is generally modeled by duplicating a few vertices in order to return to a Hamiltonian cycle. The second relaxation consists in not covering all the vertices of the graph, the set of the discarded vertices depending on external constraints. In this case, the `CYCLE` constraint can eventually be used by artificially adding the discarded vertices to the end of the Hamiltonian cycle. However, it becomes impossible to integrate them into a weighted cycle without disrupting the cost of the solution. It is then impossible to benefit from all the filtering power of constraint `WEIGHTEDCIRCUIT`.

In this paper, we aim to generalize the `WEIGHTEDCIRCUIT` constraint in case some vertices can be discarded. We consider a more general multiple tour problem that consists in covering a part of the graph by a set of minimal cost circuits. We define a new global constraint, called `WEIGHTEDSUBCIRCUITS`, that enforces multiple disjoint circuits of bounded cost to partially cover a weighted graph. This is a generalization of constraint `WEIGHTEDCIRCUIT` where the Hamiltonian circuit can be divided into several disjoint subcircuits, with an additional subset of discarded vertices.

The remainder of this paper is structured as follows. Section 2 surveys the necessary preliminaries. Section 3 covers related work on global constraints for tour problems. Section 4 gives the definition of `WEIGHTEDSUBCIRCUITS` constraint and proposes a decomposition of the constraint with standard constraints and a cycle constraint adapted to multiple tours. Section 5 deals with the filtering of this `NOSUBTOURS` constraint and that of the `WEIGHTEDSUBCIRCUITS` constraint. Finally, Sect. 6 presents some preliminary experimental results.

2 Preliminaries

We consider a weighted graph $G = (V, E, c)$ where V is a set of vertices, E a set of edges and a weight function $c : E \rightarrow \mathbb{Q}_+$.

When the graph G respects *triangle inequality*, the weight of any edge (i, j) is smaller or equal to the cost of any path from i to j .

Graphs are considered from an oriented point of view. In this context, the term circuit should be used rather than cycle. However, a cycle is generally described by a sequence of vertices or by defining, for each vertex i , the $Next_i$ vertex that follows i in the cycle. In both cases, the cycle is oriented and then it is a circuit.

An *elementary* circuit of G is a circuit where no vertex appears more than once. A *Hamiltonian* circuit is an elementary circuit of length $|V|$.

For any subset $W \subseteq V$, $G[W]$ is the subgraph of G induced by W .

For any set variable Set , the lower (respectively upper) bound of Set , denoted by $LB(Set)$ (respectively $UB(Set)$), is the set of required (respectively possible) values in Set .

In the following, we will note $OPT_{TSP}(G)$ the cost of an optimal solution for $TSP(G)$.

3 Related Work

The Constraint Programming research community has long been interested in the search for Hamiltonian circuits, which were already part of the Alice language constraints [2]. A configurable Cycle constraint was also part of the global constraints introduced in the CHIP solver [3].

The most common model is to define a variable $Next_i$ for each vertex i of the graph, where $Next_i$ represents the vertex which follows i in the Hamiltonian circuit. To ensure that each vertex is visited only once, an ALLDIFFERENT constraint can be posted on the $Next_i$ and enforcing GAC for this constraint is polynomial [4]. Conversely, checking that there is a Hamiltonian circuit is NP-complete. The filtering used in solvers for the CIRCUIT constraint is, therefore, naturally incomplete. Two subconstraints are mainly used for this filtering on the edges composing the circuit: the NOSUBTOUR constraint which prohibits the presence of subcircuits and the CONNECTED constraint which ensures the strong connectivity of the circuit. Other works have proposed a filtering based on graph separators [5] or investigated how to add explanations to the CIRCUIT constraint in a lazy clause generation solver [6].

The NOSUBTOUR constraint [7,8] is posted on graph $G = (V, E)$. It ensures that the $Next_i$ variables do not form a subtour of length strictly smaller than $|V|$. Combined with constraint ALLDIFFERENT, the NOSUBTOUR constraint enforces $Next_i$ variables to form a Hamiltonian circuit of G .

The filtering generally associated with the NOSUBTOUR constraint consists in removing from $Next_i$ any value that could close a path to form a subcircuit strictly smaller than $|V|$. This is based on the following rule (Fig. 1) applied when $Next_i$ is instantiated with j :

$$Next_i = j \wedge (\mathcal{L}(\beta(i)) + \mathcal{L}(j) + 1) < |V| - 1 \Rightarrow Next_{\varepsilon(j)} \neq \beta(i) \quad (1.1)$$

where, for any vertex z , $\beta(z)$, $\varepsilon(z)$ and $\mathcal{L}(z)$ are respectively the beginning, the end and the length of the path induced by the $Next_i$ variables and passing through z . Due to constraint ALLDIFFERENT, we must have $\varepsilon(i) = i$ and $\beta(j) = j$ when $Next_i$ is instantiated with j .

The values of $\beta(z)$, $\varepsilon(z)$ and $\mathcal{L}(z)$ can be easily managed with backtrackable variables and updated in $O(1)$ for each instantiation of $Next_z$.

The CONNECTED constraint has been less studied in literature. The simplest approach is to use a $O(|V| + |E|)$ search algorithm (like Tarjan's) to find strong connected components [9]. It is also possible to limit the number of searches by maintaining a spanning tree [10].

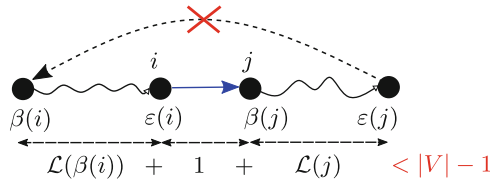


Fig. 1. Illustration of the filtering rule for the NoSubTour constraint.

Whatever the filtering used for the CIRCUIT constraint, it is only concerned with the connectivity of the graph without taking into account the edge weights. However, the cost of the circuit is a key element in the optimization of a TSP or VRP. Therefore, constraint WEIGHTEDCIRCUIT $(Next_1, \dots, Next_n, Cost)$ has been defined to enforce the rule that the circuit defined by the variables $Next_i$ has a lower cost than the variable $Cost$.

The filtering algorithms used for constraint WEIGHTEDCIRCUIT are based on TSP relaxations. This consists in reducing the TSP to a polynomial optimization problem whose optimal cost is a lower bound of the initial TSP cost. This bound can be used directly to update $LB(Cost)$. It also filters the $Next_i$ variables by eliminating the edges which are not part of the optimal solution and which, if they replace an edge of this solution, generate an additional cost beyond $UB(Cost)$.

Several relaxations can be used to filter the constraint WEIGHTEDCIRCUIT [7, 8, 11, 12] and most of them are incomparable [13]. The Minimum Spanning Tree (MST) and even better the Minimum Spanning Arborescence (MSA) directly provide a lower bound to the TSP. In Assignment Problem (AP) relaxation, the solution can be composed of several disjointed cycles. With Held and Karp 1-tree relaxation, the solution is a Minimum Spanning Tree of $G[V \setminus \{1\}]$ plus two edges connecting vertex 1 to this spanning tree.

4 The WeightedSubCircuits Constraint

The WEIGHTEDSUBCIRCUITS (WSC) constraint aims to generalize the WEIGHTEDCIRCUIT constraint. Instead of imposing a single Hamiltonian circuit on the whole graph, it enforces a Hamiltonian circuit for each subgraph induced by one or more subsets of vertices.

For the sake of simplicity, we assume that these subsets are defined by $K + 1$ set variables $Set_1, \dots, Set_K, Set_{dummy}$, the last subset being the set of discarded vertices. However, it is possible to adapt the definition with other representations of these subsets, for example with integer variables.

The WEIGHTEDSUBCIRCUITS constraint is intended to be combined with other constraints controlling the distribution of vertices in subsets $Set_1, \dots, Set_K, Set_{dummy}$ which must form a partition of the set of vertices. The number K of subsets is an upper bound as some subsets may be empty. In addition,

non-empty subsets must contain at least 2 vertices since the isolated vertices must belong to Set_{dummy} .

4.1 Definition

Definition 1. (WEIGHTEDSUBCIRCUITS)

Given a weighted graph $G = (V, E, c)$, the constraint

$$WSC[G]([Set_1, \dots, Set_K], Set_{dummy}, [Next_1, \dots, Next_n], [Cost_1, \dots, Cost_K], Z)$$

holds on the set variables $Set_1, \dots, Set_K, Set_{dummy}$ and the integer variables $Next_1, \dots, Next_n, Z$ and $Cost_1, \dots, Cost_K$ if and only if:

1. the subsets $Set_1, \dots, Set_K, Set_{dummy}$ form a partition of V ;
2. $\forall k \in 1..K$, the set $E_k = \{(i, Next_i) \mid i \in Set_k\}$ defines a Hamiltonian circuit of $G[Set_k]$ and $\sum_{(i,j) \in E_k} c(i, j) \leq Cost_k$;
3. $\forall i \in V, i \in Set_{dummy} \Leftrightarrow Next_i = i$;
4. $\sum_{k=1}^K Cost_k \leq Z$;

This definition is illustrated by Fig. 2. The set Set_{dummy} contains all discarded vertices. As in Minizinc [14] `subcircuit` constraint, we set $Next_i = i$ for all discarded vertices. This allows constraint `ALLDIFFERENT` to be applied to all $Next_i$ variables.

Because of the third rule, any subset Set_k must be empty or contains at least 2 vertices. When $K = 1$, the constraint can be simplified since $Set_{dummy} = V \setminus Set_1$ and Z is an upper bound of $Cost_1$.

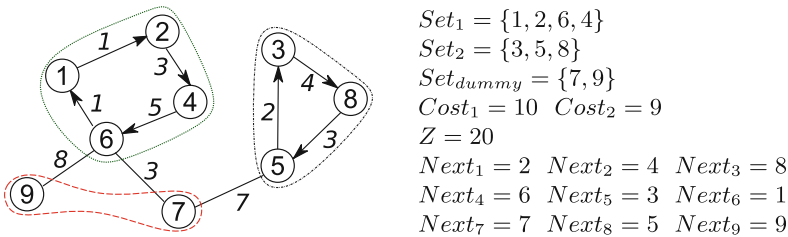


Fig. 2. An example of a weighted graph and a set of variables for which the constraint $WSC[G]([Set_1, Set_2], Set_{dummy}, [Next_1, \dots, Next_9], [Cost_1, Cost_2], Z)$ holds.

In the definition of the `WEIGHTEDSUBCIRCUITS` constraint we have assumed that the subsets are represented by set variables. This can facilitate the addition of external constraints on the subsets, such as a maximum cardinality to limit the number of vertices in each circuit. However, the constraint could also be defined with subsets represented by n membership integer variables

$\{member_i\}_{i \in V}$. Moreover, the two representations can be combined thanks to channeling constraints $\forall i \in V, \forall k \in 1..K, i \in Set_k \Leftrightarrow member_i = k$ and $i \in Set_{dummy} \Leftrightarrow member_i = K + 1$.

The WEIGHTEDSUBCIRCUITS constraint is clearly a generalization of the WEIGHTEDCIRCUIT constraint:

$$\text{WEIGHTEDCIRCUIT}[G](Next_1, \dots, Next_n, Z) \equiv \text{WSC}[G]([V], \emptyset, [Next_1, \dots, Next_n], [Z], Z)$$

Moreover, the WEIGHTEDSUBCIRCUITS constraint can also be used to implement some (but not all) variants of the generic Cycle constraint of CHIP [3]. For instance, one variant of the CHIP Cycle constraint holds for K cycles with p incompatible nodes ($p \leq K$) that must belong to disjoint cycles with total cost constrained by two bounds. This cycle constraint is equivalent to a WEIGHTEDSUBCIRCUITS constraint with additional unary constraints on the $Cost_k$ and Set_k variables.

Since WEIGHTEDSUBCIRCUITS is a generalisation of WEIGHTEDCIRCUIT, it is not surprising that filtering WEIGHTEDSUBCIRCUITS is NP-hard:

Theorem 1. *Achieving Bounds Consistency (BC) on WSC is NP-hard.*

Proof. Deciding if a graph $G = (V, E, c)$ has a Hamiltonian circuit of cost less or equal to a given value p is NP-complete. Consider the constraint

$$\text{WSC}[G]([Set_1], Set_{dummy}, [Next_1, \dots, Next_n], [Cost_1], Z)$$

where $\text{LB}(Set_1) = V$ and for each variable $Next_i, \text{dom}(Next_i) = \{j \mid (i, j) \in E\}$ and $D(Cost_1) = D(Z) = \{0, \dots, p\}$. BC empties the domain of Z if and only if G does not admit a Hamiltonian circuit of cost less or equal than p . \square

4.2 Decomposition

Before considering the development of specific propagators, we can try to decompose the WSC constraint into a set of standard constraints. Except for line 1.6, the WSC constraint can be decomposed into standard constraints as follows:

Proposition 1.

$$\text{WSC}[G](Set_1, \dots, Set_K, Set_{dummy}, Next_1, \dots, Next_n, Cost_1, \dots, Cost_K, Z) \Leftrightarrow$$

$$\text{ALLDIFFERENT}(Next_1, \dots, Next_n) \tag{1.2}$$

$$\wedge \text{PARTITION}(Set_1, \dots, Set_K, Set_{dummy}) \tag{1.3}$$

$$\wedge \forall i \in V, i \in Set_{dummy} \Leftrightarrow Next_i = i \tag{1.4}$$

$$\wedge \forall i \in V, \forall k = 1..K, i \in Set_k \Leftrightarrow Next_i \in Set_k \tag{1.5}$$

$$\wedge \text{NOSUBTOURS}(Set_1, \dots, Set_K, Next_1, \dots, Next_n) \tag{1.6}$$

$$\wedge \forall k = 1..K, \sum_{i \in Set_k} c(i, Next_i) \leq Cost_k \tag{1.7}$$

$$\wedge \sum_{k=1}^K Cost_k \leq Z \tag{1.8}$$

The ALLDIFFERENT constraint on the *Next* variables ensures that any vertex must belong to a cycle or must be isolated and then, thanks to line 1.4, must belong to Set_{dummy} . We assume that the PARTITION constraints allows empty sets. Thanks to line 1.5 each cycle must be included in a single subset Set_k . The NOSUBTOURS constraint enforces that such a cycle is an Hamiltonian cycle of the induced subgraph $G[Set_k]$. This is not a standard constraint but it is a generalization of the NOSUBTOUR constraint [7, 8]. The next section will discuss NOSUBTOURS filtering. The sum constraints (1.7) and (1.8) ensure that the cost of each cycle is greater or equal to the sum of the weights of its edges and that the total sum of $Cost_k$ variables is less or equal to Z .

5 Propagation

First we will look at the filtering of constraint NOSUBTOURS in order to be able to implement constraint WEIGHTEDSUBCIRCUITS thanks to its decomposition. We will then propose a specific additional filtering for constraint WEIGHTEDSUBCIRCUITS.

Since obtaining AC for these two constraints is NP-hard, the filtering algorithms that we will study in this section are necessarily incomplete.

5.1 NoSubTours

When $Next_i$ is instantiated with j , the filtering rule 1.1 used for the constraint NOSUBTOUR is dedicated to remove from the domain of $Next_{\varepsilon(j)}$ any value leading to a cycle of size less than $|V|$.

For the NOSUBTOURS constraints (1.6), the set $\{(i, Next_i), i \in Set_k\}$ must form a cycle of size $|Set_k|$ in $G[Set_k]$. Hence, if the path resulting from the instantiation $Next_i = j$ has a length smaller than the size of the lower bound of Set_k , this path cannot be closed at its ends to form a cycle. This corresponds to the following filtering rule:

$$\begin{aligned}
 Next_i = j \wedge i \in LB(Set_k) \wedge (\mathcal{L}(\beta(i)) + \mathcal{L}(j) + 1) < |LB(Set_k)| - 1 \\
 \Rightarrow Next_{\varepsilon(j)} \neq \beta(i) \quad (1.9)
 \end{aligned}$$

Conversely, if the resulting path passes through all vertices of the upper bound of Set_k , the cycle must be closed and Set_k is instantiated:

$$\begin{aligned}
 Next_i = j \wedge i \in LB(Set_k) \wedge (\mathcal{L}(\beta(i)) + \mathcal{L}(j) + 1) = |UB(Set_k)| - 1 \\
 \Rightarrow Next_{\varepsilon(j)} = \beta(i) \wedge Set_k = UB(Set_k) \quad (1.10)
 \end{aligned}$$

NOSUBTOURS filtering can also benefit from searching for connected components to ensure that $LB(Set_k)$ is included in a connected (via $Next_i$ variables) component of $UB(Set_k)$.

Finally, we can notice that backtractable variables like $\beta(i)$ can also be used to filter constraint (1.5) since all the vertices in the path passing through i and

connecting $\beta(i)$ to $\varepsilon(i)$ must be in the same Set_k than i . This corresponds to the following filtering rules:

$$i \in \text{LB}(Set_k) \Rightarrow \{\beta(i), \text{Next}_{\beta(i)}, \dots, i, \text{Next}_i, \dots, \varepsilon(i)\} \subseteq \text{LB}(Set_k) \quad (1.11)$$

and

$$i \notin \text{UB}(Set_k) \Rightarrow \{\beta(i), \text{Next}_{\beta(i)}, \dots, i, \text{Next}_i, \dots, \varepsilon(i)\} \cap \text{UB}(Set_k) = \emptyset \quad (1.12)$$

5.2 WeightedSubCircuits

Previous works on the WEIGHTEDCIRCUIT constraint have shown the benefit of dedicated filtering compared to separate filtering on the Circuit constraint and the cost of the circuit.

During the search, the vertices involved in the WEIGHTEDSUBCIRCUITS constraint can be divided in 4 categories (see Fig. 3):

- A. The vertices in $\text{LB}(Set_{dummy})$ will not be part of the circuits.
- B. The vertices in $\text{LB}(Set_k)$ will necessarily contribute to the value of $Cost_k$ and Z .
- C. The vertices in $V \setminus (\text{UB}(Set_{dummy}) \cup \bigcup_k \text{LB}(Set_k))$ cannot be excluded but are not yet assigned to a Set_k . They cannot yet contribute to the value of a specific $Cost_k$ but will necessarily contribute to the value of Z .
- D. For the vertices in $\text{UB}(Set_{dummy}) \setminus \text{LB}(Set_{dummy})$ it is still too early to know if they will be part of the circuits.

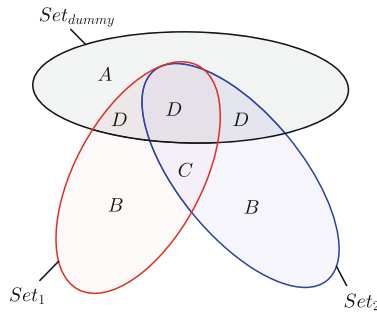


Fig. 3. Vertex distribution for constraint WEIGHTEDSUBCIRCUITS during the search: A vertices are definitely discarded and inserted in subset Set_{dummy} ; B vertices are definitely added to one subset Set_k ; C vertices can no longer be discarded but are not yet assigned to any subset S_k ; D vertices can belong to any subset.

Whether one considers a particular variable $Cost_k$ or the global variable Z , in both cases there is a subset of vertices that must be part of the solution and other vertices that may participate.

For instance, to find a lower bound for $\text{LB}(Cost_k)$, the subset of required vertices is equal to $\text{LB}(Set_k)$. It is unfortunate that it is not possible to consider only $G[\text{LB}(Set_k)]$ in order to find a lower bound for $\text{LB}(Cost_k)$:

Proposition 2. $OPT_{TSP}(G[LB(Set_k)])$ is not a lower bound for $LB(Cost_k)$

Proof. In the graph of Fig. 4, the subgraph induced by $LB(Set_k)$ is a cycle of weight 24. With additional vertices 5 and 6 added to Set_k , the induced subgraph includes a cycle of weight 21. Moreover, $G[LB(Set_k)]$ may not contain a cycle while $G[UB(Set_k)]$ does. \square

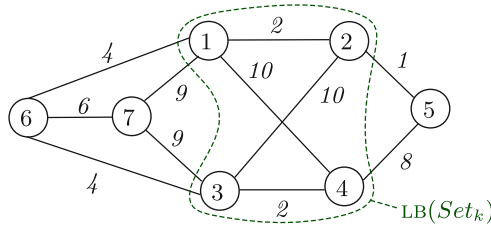


Fig. 4. The optimal value for $TSP(G[LB(Set_k)])$ is equal to 24 whereas $OPT_{TSP}(G[LB(Set_k) \cup \{5, 6\}])$ is equal to 21.

To find a lower bound for $Cost_k$, we must consider not only the mandatory vertices but also the potential vertices. This question can be reduced to the Steiner cycle problem, which is a generalization of the Steiner tree problem [15].

Definition 2 (Steiner Cycle Problem). Let $H = (V, E, c)$ a weighted graph and $V' \subseteq V$. The Steiner Cycle Problem $SCP(H, V')$ consists in finding an elementary cycle of minimum cost that contains all nodes in V' (but may include additional vertices). The cost of an optimal solution will be noted $OPT_{SCP}(H, V')$.

The TSP is a specific case of SCP where $V' = V$.

Proposition 3. $LB(Cost_k) \geq OPT_{SCP}(G[UB(Set_k)], LB(Set_k))$

Proof. $Cost_k$ is the cost of a Hamiltonian cycle in $G[Set_k]$. Thus, this cycle must necessarily pass through all the vertices of $LB(Set_k)$ and eventually through some vertices of $UB(Set_k) \setminus LB(Set_k)$. This is the exact definition of $SCP(G[UB(Set_k)], LB(Set_k))$. \square

Proposition 3 gives a way to filter $Cost_k$. Since computing an optimal Steiner cycle is NP-hard, a lower bound can be determined by relaxing a constraint of SCP as is done for TSP. To do this, we start by defining an extended subgraph:

Definition 3 (Extended subgraph). Given a weighted graph $H = (V, E, c)$ and a subset $V' \subseteq V$, the extended subgraph $\mathcal{G}(H, V')$ is obtained by adding to $H[V']$ new edges (i, j) , with weight $c(i, j) = \delta_{i,j}$, such that $(i, j) \notin E$ and there is a shortest path connecting i to j in $H[(V \setminus V') \cup \{i, j\}]$ whose cost is equal to $\delta_{i,j}$.

This definition is illustrated by Fig. 5. The edges added to $H[V']$ correspond to a shortest path outside V' and connecting two non-adjacent vertices of $H[V']$.

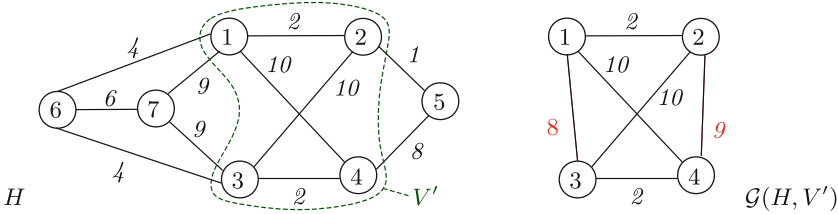


Fig. 5. Extended subgraph $\mathcal{G}(H, V')$ for a graph $H = (V, E, c)$ and a subset $V' \subseteq V$

Proposition 4. *Given a weighted graph $H = (V, E, c)$ that respects triangular inequality and $V' \subseteq V$ with $|V'| \geq 3$ we have*

$$OPT_{SCP}(H, V') \geq OPT_{TSP}(\mathcal{G}(H, V'))$$

Proof. Any Hamiltonian cycle \mathcal{C} that is solution of $SCP(H, V')$ is composed of paths included in V' and paths outside V' . Let $\mathcal{P}_{i,j} = \langle i, x_1, \dots, x_t, j \rangle$ be a sub-path of \mathcal{C} such that $i, j \in V'$ and $\forall p \in 1..t, x_p \notin V'$. If $(i, j) \in E$, the triangular inequality imposes that $c(i, j)$ is not greater than the cost of $\mathcal{P}_{i,j}$. If $(i, j) \notin E$, by construction of $\mathcal{G}(H, V')$, there exists in $\mathcal{G}(H, V')$ an edge (i, j) whose weight is lower or equal to the cost of $\mathcal{P}_{i,j}$. Thus, the cycle obtained by replacing in \mathcal{C} all paths $\mathcal{P}_{i,j}$ by edge (i, j) is a Hamiltonian cycle of $\mathcal{G}(H, V')$ whose cost is lower or equal to that of \mathcal{C} . \square

By combining Propositions 3 and 4 we obtain a filtering rule for $Cost_k$:

Corollary 1. $LB(Cost_k) \geq OPT_{TSP}(\mathcal{G}(G[\text{UB}(Set_k)], LB(Set_k)))$

and therefore, a filtering rule for Z :

Corollary 2. $LB(Z) \geq \sum_{k=1}^K OPT_{TSP}(\mathcal{G}(G[\text{UB}(Set_k)], LB(Set_k)))$

Since all Set_k must be disjointed, all graphs $\mathcal{G}(G[\text{UB}(Set_k)], LB(Set_k))$ are disjointed and so we have:

Proposition 5. $LB(Z) \geq OPT_{TSP}(\cup_{k=1}^K \mathcal{G}(G[\text{UB}(Set_k)], LB(Set_k)))$

Computing $\cup_{k=1}^K \mathcal{G}(G[\text{UB}(Set_k)], LB(Set_k))$ can be done in $O(|V|(|E| + |V| \log |V|))$ with at most $|V \setminus \text{UB}(Set_{dummy})|$ calls to Dijkstra’s algorithm. This is comparable to the complexity of some relaxation algorithms. For example, the Hungarian algorithm used for the Assignment Problem relaxation is in $O(|V|^3)$.

Thanks to Proposition 5, a lower bound of Z can be computed with a relaxation of the TSP, as in the case of constraint WEIGHTEDCIRCUIT.

This bound is directly related to the state of variables Set_k , which define the vertices of the extended subgraphs, and to the domains of variables $Next_i$, which fix adjacency in G . Depending on the relaxation used, it is also possible to take into account variable Set_{dummy} .

For example, suppose we use the relaxation corresponding to the Assignment Problem (AP). For any graph H , a solution of $AP(H)$ is a set of disjointed

minimum cost elementary circuits covering all the vertices of H . Applying AP to extended subgraphs results in a set of sub-cycles covering all type B vertices of Fig. 3. However, type C vertices are not covered by these sub-cycles even though they will necessarily be part of the final cycles. To take into account these vertices we can expand the extended subgraphs to vertices of type C .

Let $\mathcal{S}_B = \bigcup_{k=1}^K \text{LB}(\text{Set}_k)$ the set of type B vertices and \mathcal{S}_C the set of type C vertices. We have $\mathcal{S}_C = V \setminus (\text{UB}(\text{Set}_{dummy}) \cup \mathcal{S}_B)$.

We define the *global extended subgraph* \mathcal{G}^* as follow:

Definition 4 (*Global extended subgraph*). The global extended subgraph \mathcal{G}^* is an extended subgraph built on $G[\mathcal{S}_B \cup \mathcal{S}_C]$ by adding only edges between two vertices of the same $\text{LB}(\text{Set}_k)$ or between a vertex of \mathcal{S}_B and a vertex of \mathcal{S}_C .

This definition is illustrated by Fig. 6.

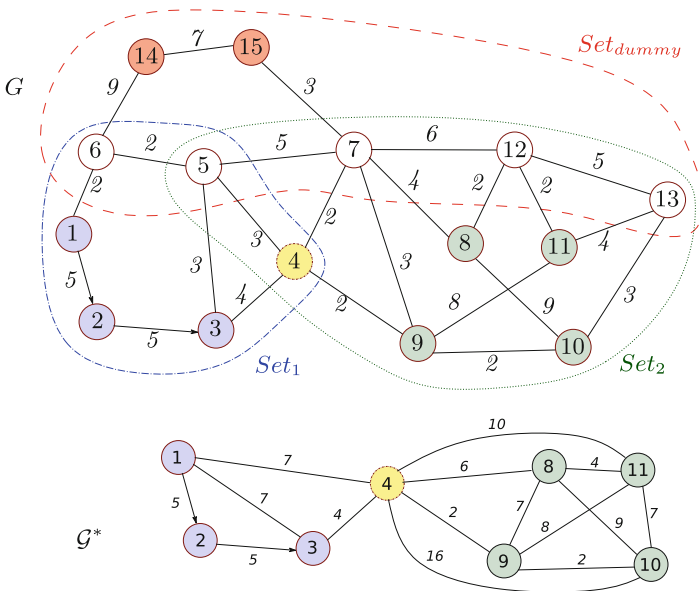


Fig. 6. En example of global extended subgraph \mathcal{G}^* , with $\mathcal{S}_C = \{4\}$ and $\mathcal{S}_B = \text{LB}(\text{Set}_1) \cup \text{LB}(\text{Set}_2) = \{1, 2, 3, 8, 9, 10, 11\}$ and $\text{Next}_1 = 2, \text{Next}_2 = 3$

Computing \mathcal{G}^* has the same complexity $O(|V|(|E| + |V| \log |V|))$ as computing all extended subgraphs.

Since \mathcal{G}^* is built from a subgraph that contains the C -type vertices, we have a new lower bound for Z :

Proposition 6

$$\text{LB}(Z) \geq \text{OPT}_{AP}(\mathcal{G}^*) \tag{1.13}$$

The proof is similar to that of Proposition 4.

For the graph in Fig. 6, the cycle set $\{\langle 1, 2, 3, 4, 1 \rangle, \langle 8, 11, 8 \rangle, \langle 9, 10, 9 \rangle\}$ is an optimal solution to $AP(\mathcal{G}^*)$. The cost (33) of this solution is a lower bound

of Z , whatever the vertices added to Set_1 or Set_2 . Moreover, according to the upper bound of Z , some edges can be discarded because they cannot be part of a solution. This filtering is based on computing a reduced cost for each arc (i, j) not in the solution of $AP(\mathcal{G}^*)$, i.e., the minimum increase of the overall cost for setting $Next_i$ to j (see [12, 13]). For instance, with $UB(Z) = 40$, edges $(8, 10)$ and $(9, 11)$ could be eliminated. This filtering concerns only the edges of \mathcal{G}^* that belong to G .

6 Experimental Results

This section presents some preliminary results to evaluate the benefits of the constraint WEIGHTEDSUBCIRCUITS. These preliminary experiments aim to measure the interest of a filtering based on Steiner cycles.

Rather than generating random data we consider the Balancing Bike Sharing Systems (BBSS). This problem is linked to the management of a shared bicycle fleet. The objective is to optimize a tour of the stations in order to remove bicycles from overfilled stations and refill empty stations. The capacity of the transport vehicle and the time available do not allow all stations to be optimized. We implemented the CSP model of [16, 17] which uses a CYCLE constraint. We simply modified it in order to make dummy vertices appear: to ensure that some stations will not be visited, we imposed that the demand for each visited station be fully satisfied rather than partially handle all stations. The benchmark is based on instances from the city of Vienna given by [16]. To generate additional instances from size 12 to 18, we extracted a subset of vertices from instances with 20 vertices. Unlike the initial article which was based on a Large Neighborhood Search (LNS) approach, we used a standard search procedure used with fixed ordering of variables. This limits the size of instances that can be processed.

We compared 4 models based on:

- a simple CYCLE constraint using the NOSUBTOUR filtering.
- the decomposition of the WEIGHTEDSUBCIRCUITS constraint presented in Sect. 5.1, with the filtering rules 1.9 to 1.12 for the NOSUBTOURS constraint
- the NOSUBTOURS constraint plus the filtering rule for Z based on $AP(\mathcal{G}^*)$ (Proposition 6)
- the previous filtering rules with additional filtering on the $Next_i$ variables using reduced costs from $AP(\mathcal{G}^*)$.

We implemented these models in the Java library Choco 4 [18]. All the experiments were executed on a Linux machine with Intel(R) Xeon(R) CPU E5-2680 (2.40 GHz). The time limit for each run was set to 2 h.

Table 1 summarizes the results obtained for series of 30 graphs of different sizes. When all the graphs in a series have been resolved, the CPU time and the number of nodes are an average over the 30 graphs. Otherwise, only the number of resolved instances is displayed.

These preliminary results show that the model based on the decomposition of the WEIGHTEDSUBCIRCUITS constraint allows to find an optimal solution for

Table 1. Average results on BBS instances for the WEIGHTEDSUBCIRCUITS.

V	CIRCUIT	Decomposition of WSC		WSC filtering on Z with $AP(\mathcal{G}^*)$ relaxation		WSC filtering on Z and $Next_i$	
		# solved	Time (s)	Nodes	Time (s)	Nodes	Time (s)
10	12/30	13	201,658	6	78,735	6	66,984
12	6/30	89	1,415,712	16	228,809	12	159,703
14	0/30	381	6,035,049	49	723,136	40	522,914
16		1515	23,729,425	181	2,480,824	121	1,489,637
18		26/30		526	6,480,702	369	4,017,667
20		19/30		1523	16,147,092	1097	10,475,711

instances up to 20 vertices while the model based on the CIRCUIT constraint of Choco (NoSubTour) reaches the time limit for several small instances. In addition, the filtering of the Z variable based on the $AP(\mathcal{G}^*)$ relaxation seems to be quite effective. The last columns show that the computation of \mathcal{G}^* is even more profitable if it is used to eliminate edges by filtering the $Next_i$ variables.

7 Conclusion

In this paper, we considered circuit constraints that allow the modeling of tour problems in a CP solver. We have proposed a new global constraint, named WEIGHTEDSUBCIRCUITS, that enforces multiple disjoint circuits of bounded total cost to partially cover a weighted graph. The constraint is posted on a family of subsets of vertices to obtain a Hamiltonian circuit in each subgraph induced by a subset. The WEIGHTEDSUBCIRCUITS constraint is intended to be combined with other constraints that control the composition of these subsets and the dummy set of discarded vertices.

We have shown that the WEIGHTEDSUBCIRCUITS constraint can improve filtering where the WeightedCircuit constraint cannot be used because of the dummy vertices. We have proposed an adaptation of the NOSUBTOUR constraint filtering that is compatible with discarded vertices. We have shown that computing a lower bound of the cost of each circuit can be reduced to a Steiner circuit problem. We demonstrated how to obtain a lower bound of the Steiner circuit by solving a TSP relaxation on an extended subgraph. To obtain a lower bound of the total cost of all circuits, we have shown that it is possible to take into account the required vertices that are not yet assigned to a subset, using AP relaxation. Preliminary experiments have shown the potential of this approach and encourage further exploration of filtering rules for the new constraint.

References

1. Toth, P., Vigo, D.: Vehicle routing: problems, methods, and applications. In: SIAM (2014)
2. Laurière, J.: A language and a program for stating and solving combinatorial problems. *Artif. Intell.* **10**(1), 29–127 (1978)
3. Beldiceanu, N., Contejean, E.: Introducing global constraints in chip. *Math. Comput. Model.* **20**(12), 97–123 (1994)
4. Régim, J.C.: A filtering algorithm for constraints of difference in CSPs. In: Proceedings of the National Conference on Artificial Intelligence, AAAI-94, pp. 362–367. Seattle (1994)
5. Kaya, L.G., Hooker, J.N.: A filter for the circuit constraint. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 706–710. Springer, Heidelberg (2006). https://doi.org/10.1007/11889205_55
6. Francis, K.G., Stuckey, P.J.: Explaining circuit propagation. *Constraints* **19**(1), 1–29 (2014)
7. Caseau, Y., Laburthe, F.: Solving small TSPs with constraints. In: Proceedings of the 14th International Conference on Logic Programming, pp. 316–330. MIT Press (1997)
8. Pesant, G., Gendreau, M., Potvin, J., Rousseau, J.: An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transp. Sci.* **32**(1), 12–29 (1998)
9. Dooms, G.: The CP (Graph) computation domain in constraint programming. Ph.D. thesis, Université catholique de Louvain (2006)
10. Prosser, P., Unsworth, C.: A connectivity constraint using bridges. In: ECAI 2006: 17th European Conference on Artificial Intelligence, Frontiers in Artificial Intelligence and Applications, vol. 141, pp. 707–708. IOS Press (2006)
11. Focacci, F., Lodi, A., Milano, M.: A hybrid exact algorithm for the TSPTW. *INFORMS J. Comput.* **14**(4), 403–417 (2002)
12. Benchimol, P., Hoeve, W.J.V., Régim, J.C., Rousseau, L.M., Rueher, M.: Improved filtering for weighted circuit constraints. *Constraints* **17**(3), 205–233 (2012)
13. Ducomman, S., Cambazard, H., Penz, B.: Alternative filtering for the weighted circuit constraint: comparing lower bounds for the TSP and solving TSPTW. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, pp. 3390–3396 (2016)
14. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessiere, C. (ed.) Principles and Practice of Constraint Programming - CP 2007, pp. 529–543. Springer, Berlin Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_38
15. Hwang, F.K., Richards, D.S., Winter, P.: The Steiner tree problem. In: *Annals of Discrete Mathematics*, vol. 53. Elsevier (1992)
16. Di Gaspero, L., Rendl, A., Urli, T.: Constraint-based approaches for balancing bike sharing systems. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 758–773. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_56
17. Di Gaspero, L., Rendl, A., Urli, T.: Balancing bike sharing systems with constraint programming. *Constraints* **21**(2), 318–348 (2016)
18. Prud’homme, C., Fages, J.G., Lorca, X.: Choco Documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. (2016). <http://www.choco-solver.org>



Towards Semi-Automatic Learning-Based Model Transformation

Kiana Zeighami¹(✉), Kevin Leo¹, Guido Tack^{1,2},
and Maria Garcia de la Banda^{1,2}

¹ Faculty of IT, Monash University, Melbourne, Australia
{kiana.zeighami,kevin.leo,guido.tack,maria.garciadelabanda}@monash.edu
² Data61/CSIRO, Melbourne, Australia

Abstract. Recently, [16] showed that the nogoods inferred by learning solvers can be used to improve a problem model, by detecting constraints that can be strengthened and new redundant constraints. However, the detection process was manual and required in-depth knowledge of both the learning solver and the model transformations performed by the compiler. In this paper we provide the first steps towards a (largely) automatic detection process. In particular, we discuss how nogoods can be automatically simplified, connected back to the constraints in the model, and grouped into more general “patterns” for which common facts might be found. These patterns are easier to understand and provide stronger evidence of the importance of particular constraints. We also show how nogoods generated by different search strategies and problem instances can increase our confidence in the usefulness of these patterns. Finally, we identify significant challenges and avenues for future research.

1 Introduction

Lazy Clause Generation (LCG) [3, 11] is a powerful solving technique that combines the strengths of Constraint Programming and SAT solving. It works by instrumenting finite domain propagation to record the reasons for each propagation step, thus creating an implication graph like the ones built by SAT solvers [9]. This graph is used to derive nogoods (i.e., reasons for failure) that are recorded as clausal propagators and propagated efficiently using SAT technology. The combination of constraint propagation and clause learning can dramatically reduce search and greatly improve performance (e.g., [14, 15]).

Shishmarev et al. [16] have shown that the nogoods inferred by a learning solver when executing a model may be used to design model transformations that improve its execution. In hindsight, this should have been expected: a nogood is introduced when the propagation achieved by the constraints in the model is not strong enough to avoid a failure during search. By identifying the nogoods that caused the biggest search reduction and connecting them back to the constraints in the model, Shishmarev et al. were able to improve propagation for two models. This was achieved by modifying the model constraint that generated the nogood, and by adding a new redundant constraint (suggested by the nogood).

Inferring *useful* redundant constraints for a given model is extremely difficult. Thus, using nogoods to achieve this and to strengthen the constraints already

in the model is an exciting new approach with significant potential. However, Shishmarev et al. used a manual method to infer model improvements based on the inferred nogoods. Further, the inference required in-depth knowledge of the learning solver and of the transformation performed (in this case by the MiniZinc [10] compiler) to the user’s model to become the input to the target solver. In this paper we perform the first steps towards a semi-automatic process. In particular, we show how the nogoods can be automatically renamed, simplified, connected back to the constraints in the model, and grouped into more general patterns for which common facts might be found. These patterns are easier to understand by (expert) modellers than the raw nogoods, and their associated facts can help these modellers find useful modifications to the model. Additionally, we show how the generation of nogoods using different search strategies and instances of a problem can be used to increase our confidence in the usefulness of these patterns. We show the potential of the approach with two case studies, and finish by identifying some of the significant challenges with which we are still faced, and some avenues for future research that may help resolve them.

2 Background

Constraint Programming: A *constraint problem* P is a tuple (C, D, f) , where C is a set of constraints, D a *domain* which maps each variable x appearing in C to a set of values $D(x)$, and f an (optional) objective function. Set C is logically interpreted as the conjunction of its elements, and D as the conjunction of unary constraints $x \in D(x)$, for each variable x appearing in C . A *literal* of P is a unary constraint whose variable appears in C . A constraint solver starts from a problem $P \equiv (C, D, f)$ and applies propagation to reduce domain D to D' as a fixpoint of all propagators for C . If D' is equivalent to *false* ($D'(x)$ is empty for some variable x), we say P is failed. If D' is not equivalent to *false* and fixes all variables, we have found a solution to P . Otherwise, the solver splits P into n subproblems $P_i \equiv (C \wedge c_i, D', f)$, $1 \leq i \leq n$ where $C \wedge D' \Rightarrow (c_1 \vee c_2 \vee \dots \vee c_n)$ and where c_i are literals (the *decisions*), and iteratively searches these.

The search proceeds making decisions until either (1) a solution is found, (2) a failure is detected, or (3) a restart event occurs. In case (1) the search either terminates if the model has no objective function f , or computes the value of f , sets a bound for the next value of f to be better (greater or smaller, depending on f) and continues the search for this better value. In case (2), the search usually backtracks to a previous point where a different decision can be made. In case (3) the search restarts, possibly incorporating new constraints previously learnt.

Lazy Clause Generation: LCG solvers [3, 11] extend CP solvers by instrumenting their propagators to explain domain changes in terms of *equality* ($x = d$ for $d \in D(x)$), *disequality* ($x \neq d$) or *inequality* ($x \geq d$ or $x \leq d$) literals. An *explanation* for literal ℓ is $S \rightarrow \ell$, where S is a set of literals (interpreted as a conjunction). For example, the explanation for the propagator of constraint $x \neq y$ inferring literal $y \neq 5$, given literal $x = 5$, is $\{x = 5\} \rightarrow y \neq 5$. Each new literal inferred by a propagator is recorded together with its explanation, forming an *implication graph*. Whenever the search reaches a failure, LCG solvers use the implication graph to compute a *nogood* N , that is, a set of literals (interpreted as

a conjunction) that represents the reason for the failure and cannot be extended to a solution. Then, they add its negation ($\neg N$, a set of literals interpreted as their disjunction) as a clausal propagator and backtrack, resuming the search. These learnt clauses ensure the search cannot fail again for the same reasons.

Modelling: We distinguish between a constraint problem *model*, where the input data is described in terms of parameters (i.e., variables that will be fixed before the search starts), and a particular model *instance*, where the values of the parameters are added to the model. Constraint models are usually defined in a high level language, such as Essence [4] or MiniZinc [10], and their instances are compiled into a *flattened* format, where loops are unrolled into the appropriate set of constraints, and global constraints are potentially decomposed into a representation suitable for the selected solver. Note that the compiler may introduce new variables and constraints during this flattening process, and the solver can also introduce further variables and constraints during its execution of the instance. The nogoods of an LCG solver will consist of literals that refer to this fully flattened and decomposed instance of the problem. This makes their analysis in terms of the high-level model particularly challenging.

Paths: We use the concepts of *variable paths* [6] and *constraint paths* [7], which assign a unique identifier to each variable and constraint appearing in a flattened instance. They allow us to connect the flattened variables and constraints to the model’s source code. Each identifier describes the path the compiler took when compiling a MiniZinc instance (that is, a MiniZinc model and its input data) to FlatZinc, from the actual model to the point where a new variable or constraint is introduced. For example, the following is a (simplified) constraint path:

```
14:12-15:61 forall:p1=1,p2=6 14:36-15:60 -> 14:37-14:74 /\ 14:37-14:74 clause
```

Each of its components has two parts: four numbers denoting the span of text in the MiniZinc model that the expression came from (with format from line:column-to line:column); and a textual description of what the expression represents. The above path represents a `clause` that was inserted into the final FlatZinc, as a result of encoding a negated (thus the `clause` part) conjunction `/\` appearing in the left hand side of the implication (`->`) that appears in the `forall` loop from lines 14 – 15, with index variables `p1=1` and `p2=6`.

3 Towards Automation: The freepizza case

Shishmarev et al. [16] demonstrated how the clauses learnt by an LCG solver can be used to improve a model. To achieve this, they executed the model using the LCG solver Chuffed [2], replayed its search decisions (in the same order) using the non-LCG solver Gecode [13], merged the two resulting search trees, counted the number of nodes explored by Gecode that were not explored by Chuffed, and assigned them to the learnt clause(s) that helped Chuffed fail before Gecode. This formed a ranking of all clauses based on the total number of search nodes avoided by each clause, with the top-ranked clauses being the most effective for reducing search. Finally, they manually inspected the 10 most effective clauses to learn information that could help improve the model.

This section introduces several techniques for automating part of the process described in [16] using MiniZinc (the techniques can, however, be applied in other

modelling languages). We use one of their case studies (free pizza) to illustrate the manual process on a hard problem (Chuffed only solved 1 of 5 instances in the 2015 MiniZinc Challenge [17]), and the issues faced when automating it. In this problem customers get pizzas either by paying for them or by using vouchers. Each voucher (a, b) allows customers to get b number of pizzas for free if they pay for a number of pizzas, and none of the b pizzas are more expensive than the a ones. A customer who has m vouchers and wants n pizzas aims to minimise the amount paid for the n pizzas. Their MiniZinc model (called `freepizza`) is:

```

1  int: n;      set of int: PIZZA = 1..n;    % number of pizzas wanted
2  array[PIZZA] of int: price;              % price of each pizza
3  int: m;      set of int: VOUCH = 1..m;   % number of vouchers
4  array[VOUCH] of int: buy;                % buy this many to use voucher
5  array[VOUCH] of int: free;               % get this many free
6
7  set of int: ASSIGN = -m .. m; % i -i 0 (free/paid with voucher i or not)
8  array[PIZZA] of var ASSIGN: how;
9  array[VOUCH] of var bool: used;
10
11 constraint forall(v in VOUCH)(used[v]<->sum(p in PIZZA)(how[p]=-v)>=buy[v]);
12 constraint forall(v in VOUCH)(sum(p in PIZZA)(how[p]=-v) <= used[v]*buy[v]);
13 constraint forall(v in VOUCH)(sum(p in PIZZA)(how[p]=v) <= used[v]*free[v]);
14 constraint forall(p1, p2 in PIZZA)((how[p1] < how[p2] /\ how[p1]= -how[p2])
15                                     -> price[p2] <= price[p1]);
16 int: total = sum(price);
17 var 0..total: objective = sum(p in PIZZA)((how[p] <= 0)*price[p]);

```

Lines 1–5 introduce the parameters: numbers n and m , an array for the pizzas' prices, and two arrays for vouchers s.t. $(\text{buy}[i], \text{free}[i])$ represents the i th voucher (a, b) . The next three lines define two arrays of decision variables: $\text{used}[v]$, which is true iff voucher v was used; and $\text{how}[p]$, which is v if pizza p was free thanks to voucher v , is 0 if p was paid for and not used in any voucher, and is $-v$ if p was paid for and used to get free pizzas with voucher v .

Constraints start in line 11, which states that if voucher v was used, then the total number of pizzas bought and assigned to v must be greater than or equal to the number of pizzas required by it. Line 12 states similar information but in the opposite direction: the total number of pizzas bought and assigned to voucher v must be less than or equal to $\text{used}[v] * \text{buy}[v]$. Together they constrain the total number of pizzas bought for v to be equal to $\text{buy}[v]$, if used. The constraint in line 13 states that the total number of free pizzas obtained thanks to voucher v must be smaller than or equal to the number of free pizzas allowed by v if used ($\text{used}[v] * \text{free}[v]$). The last constraint states that if there are two pizzas p_1 and p_2 assigned to the same voucher with p_2 being free and p_1 being paid for (given $\text{how}[p_1] < \text{how}[p_2]$ and $\text{how}[p_1] = -\text{how}[p_2]$), then the price of p_2 must be lower than or equal to that of p_1 . Finally, the objective function is defined as the sum of the prices of the pizzas that are bought.

Table 1 shows the top 10 clauses found in [16] for `freepizza` with data:

```

n = 10; m = 4; price = [70, 10, 60, 65, 30, 100, 75, 40, 45, 20];
buy = [1, 2, 3, 3]; free = [1, 1, 2, 1];

```

Each clause is interpreted as the disjunction of its literals. For example, the clause ranked 4th, $\{\text{how}[5] \neq -3, \text{how}[3] \neq 3\}$, states that pizza 3 cannot be

obtained for free using voucher 3 by paying for pizza 5 with that voucher (this might be easier to see in its equivalent form $\neg(\text{how}[5]=-3 \wedge \text{how}[3]=3)$).

Table 1. Taken from [16]: Most effective learnt clauses in `freepizza`

Rank	Reduction	Clause
1	3425	$\text{how}[1]=-1 \text{ how}[2]=-1 \text{ how}[3]=-1 \text{ how}[4]=-1 \text{ how}[5]=-1$ $\text{how}[1]=-2 \text{ how}[2]=-2 \text{ how}[3]=-2 \text{ how}[4]=-2 \text{ how}[5]=-2$ $\text{how}[6] \leq 0 \text{ how}[6] \geq 3$
2	2068	$\text{how}[7] \leq 2 \text{ how}[7] \geq 4 \text{ how}[1] \neq -3 \text{ how}[1] \geq -2$
3	1712	$\text{how}[4] \neq 3 \text{ how}[1] = -3 \text{ how}[2] = -3 \text{ how}[3] = -3 \text{ how}[4] = -3$
4	1636	$\text{how}[5] \neq -3 \text{ how}[3] \neq 3$
5	1636	$\text{how}[8] \neq -3 \text{ how}[3] \neq 3$
6	1636	$\text{how}[9] \neq -3 \text{ how}[3] \neq 3$
7	1636	$\text{how}[10] \neq -3 \text{ how}[3] \neq 3$
8	1489	$\text{how}[6] \leq 2 \text{ how}[6] \geq 4 \text{ how}[1] \neq -3 \text{ how}[1] \geq -2$
9	1404	$\text{how}[5] \neq -3 \text{ how}[4] \neq 3 \text{ how}[4] \leq 2$
10	1403	$\text{how}[10] \neq -3 \text{ how}[4] \neq 3$

3.1 Renaming Literals

The clauses in Table 1 are already the result of significant manual interpretation and analysis. For example, the 4th clause came from renaming (and simplifying) solver-level clause $\{x_INTRODUCED_4 \neq -3, x_INTRODUCED_2 \neq 3, x_INTRODUCED_2 \leq 2\}$, where the names `x INTRODUCED 4` and `x INTRODUCED 3` were introduced by the MiniZinc compiler for the model variables in array positions `how[5]` and `how[3]`, respectively. Thus, our first step towards automation is to transform clauses to refer to model-level variables and expressions. For simple renamings as in the example above, we could instrument the MiniZinc compiler to keep a map from solver-level to model-level names. New variables may, however, also be introduced when flattening expressions. For example, the compiler may introduce an auxiliary variable for the result of an addition, or the result of flattening a `let` expression. Connecting such variables back to model-level names is more complex. We propose to do this by using variable paths [6]. Consider, for example, the literal `x INTRODUCED 244 = true`, which appears in one of the clauses generated by Chuffed for `freepizza`. As variable `x INTRODUCED 244` does not correspond directly to any model-level variable, the compiler produces the following path:

```
14:12-15:61 forall:p1=1,p2=6 14:36-15:60 -> 14:37-14:74 /\
14:37-15:74 clause 14:58-14:74 = 14:58-14:74 int_lin.eq
```

The final entry in the path shows the location in the model of the expression that corresponds to `x INTRODUCED 244`: line 14, columns 58–74, which has expression "`how[p1] = -how[p2]`". The path also shows that the expression is located within the `forall` that spans lines 14:12–15:61, within the implication (`->`) that spans lines 14:36–15:60, within the conjunction (`/\`) that spans line 14:37–14:75, within

the `clause` that the negated conjunction was encoded as, and within the = that spans line 14:58–14:74. Since the path ends with an `int_lin_eq` constraint (integer linear =), we can deduce that the introduced variable is the Boolean control variable for the reified version of expression "`how[p1]= -how[p2]`" in the model. Since the path also records the values of loop variables `p1` and `p2`, these can be automatically substituted, yielding "`how[1]= -how[6]`"=`true`.

3.2 Simplifying Literals and Clauses

The readability of any clause can be further improved by simplifying its literals based on their semantics. We do this in two different ways. First, we simplify literals whose variables were introduced by the compiler, and which correspond to Boolean expressions in the model, as follows:

positive: if the right-hand side of a literal is true (e.g. `=true`, `=1`, `>=1`), and the left-hand side is a Boolean expression of the form $e_1 \text{ op } e_2$, where op is a binary operator, then we can simply remove the right hand side.

negative: if the right-hand side of a literal is false (e.g. `=false`, `=0`, `<=0`), and the left-hand side is a Boolean of the form $x \text{ op } v$, where x is a variable, op is a binary operator and v is an integer value, then we can negate the left hand side and remove the right hand side.

For example, literal "`how[1] < how[6]`"=`true` becomes `how[1] < how[6]`, and literal "`how[1]=-1`"`<=0` (which comes from the sum of reified equality constraints in line 12 of the model) becomes `how[1]≠-1`. Simplifying literals with more complex left-hand sides (i.e., arbitrary MiniZinc expressions), requires a deeper integration with the MiniZinc compiler and is left for future work.

Second, we eliminate from each clause any literal that entails (i.e., implies) other literals in the clause, since $A \vee B \vee C$ is equivalent to $A \vee B$, if C entails B . This makes the clause easier to understand and, as shown in Sect. 3.4, makes it easier to automatically detect clause patterns. We automatically simplify a clause by applying the following rules to literals that operate on the same variable x :

\neq a literal of the form $x \neq v$ is entailed by any other literal $x = v'$ such that $v \neq v'$ and any literal $x \leq v'$ ($x \geq v'$) such that $v' \leq v$ ($v' \geq v$). Thus, those other literals are eliminated. For example, clause $\{x = 1, x \leq 1, x \geq 4, x \neq 2, \dots\}$ becomes $\{x \neq 2, \dots\}$.

\geq (\leq) a literal of the form $x \geq v$ ($x \leq v$) is entailed by any other literal $x \geq v'$ ($x \leq v'$) s.t. $v < v'$ ($v > v'$). Thus, we only keep the literal with the lowest (highest) bound. For example, clause $\{x \leq 1, x \leq 2, x \geq 3, x \geq 4, \dots\}$ becomes $\{x \leq 2, x \geq 3, \dots\}$.

$\leq \geq$ if a clause contains two literals $x \leq v_1$ and $x \geq v_2$, s.t. $v_2 - v_1 = 2$, then these two literals can be replaced by a single literal $x \neq v_2 - 1$. For example, clause $\{x \leq 1, x \geq 3, \dots\}$ becomes $\{x \neq 2, \dots\}$.

Applying these rules to our clause $\{\text{how}[5] \neq -3, \text{how}[3] \neq 3, \text{how}[3] \leq 2\}$ yields the one ranked 4th in Table 1: $\{\text{how}[5] \neq -3, \text{how}[3] \neq 3\}$, since $\text{how}[3] \leq 2$ entails $\text{how}[3] \neq 3$, and is thus eliminated by the \neq rule.

Note that all simplifications are done after renaming the introduced variables. This is useful, as literals that reference different introduced variables may later become a single literal. This happens often in `freepizza` as, for example, the `sum` function expects integer variables, but in the model it is given Booleans. Each Boolean variable is coerced by the compiler to be integer by introducing a new integer variable and posting a `bool2int` predicate equating the Boolean to the integer one. For expression `sum(p in PIZZA)(how[p]==-v)`, both the original Boolean variables and the introduced integer variables refer to the expression "`how[p]==-v`" in the model. However, before renaming, the literals may appear different (e.g., `X INTRODUCED_45=false` and `X INTRODUCED_53<=0`) even though they mean the same thing (`how[p]≠-v`). By performing the simplification after the renaming, both variables are known to be (or come from) a Boolean expression, and are thus simplified to `how[p]≠-v`. Interestingly, if these simplifications had been applied to the clauses in Table 1, Shishmarev et al. would have realised that clauses 2, 8 and 9, can be further simplified to $\{\text{how}[7] \neq 3, \text{how}[1] \neq -3\}$, $\{\text{how}[6] \neq 3, \text{how}[1] \neq -3\}$, and $\{\text{how}[5] \neq -3, \text{how}[4] \neq 3\}$.

3.3 Connecting Clauses to the Constraints in the Model

One of the main achievements of Shishmarev et al. was to use the learnt clauses to identify the need to strengthen the constraint in line 14 of the model. This need was discovered by realising that some of the top clauses were direct consequences of a single constraint; the one in line 14. To do this, Shishmarev et al. manually linked the learnt clauses (or more accurately, their literals) to the model constraints they were derived from.

To automate this step, we instrumented Chuffed to record the solver-level constraint directly responsible for adding any literal to the clause database. That is, for each explanation $S \rightarrow \ell$ added by a constraint with identifier id_c , we record that ℓ was directly generated from id_c . Then, when a clause Cl is generated, we obtain the constraint identifier of each literal ℓ in Cl , ask that constraint to provide us with the explanation S that was used to generate ℓ , and recursively apply the same method for all literals in S . This allows us to (lazily) trace back all constraints involved in generating the literals of Cl .

For clause $\{\text{how}[5] \neq -3, \text{how}[3] \neq 3\}$, our method identifies a single solver-level constraint as responsible for all literals. Using constraint paths, we can trace it back to the expression `how[p1]==-how[p2]` on line 14 of `freepizza`, with loop variables `p1=5` and `p2=3`. Thus, our automatic method successfully identifies the line in which the constraint responsible for the clause appears. While connecting the clauses to the model constraints via a textual path is already very useful, it would be easier for users to get a visual connection. To achieve this, we have modified the MiniZinc IDE to visually highlight the parts of the user's model responsible for a given clause (see Fig. 1).

3.4 Finding Patterns Among Clauses

At this point, our automatic method can make a clause clearer, simpler and visually connect it to the constraints it came from. However, a single clause may not be worth exploring, even if it led to a considerable search reduction.


```

int: n;      set of int: PIZZA = 1..n;      % number of pizzas
array[PIZZA] of int: price;                % price of each pizza
int: m;      set of int: VOUCHER = 1..m;   % number of vouchers
array[VOUCHER] of int: buy;                % buy this many to use voucher
array[VOUCHER] of int: free;                % get this many free

set of int: ASSIGN = -m .. m; % i -i 0 (pizza is free/paid with voucher i or not)
array[PIZZA] of var ASSIGN: how;
array[VOUCHER] of var bool: used;

constraint forall(v in VOUCHER)(used[v]<->sum(p in PIZZA)(how[p] = -v) >= buy[v]);
constraint forall(v in VOUCHER)(sum(p in PIZZA)(how[p]=-v) <= used[v]*buy[v]);
constraint forall(v in VOUCHER)(sum(p in PIZZA)(how[p]=v) <= used[v]*free[v]);
constraint forall(p1, p2 in PIZZA)((how[p1] < how[p2] /\ how[p1]= -how[p2])
-> price[p2] <= price[p1]);

int: total = sum(price);
var 0..total: objective = sum(p in PIZZA)((how[p] <= 0)*price[p]);

```

Fig. 1. Constraint responsible for learnt clause $\{\text{how}[5] \neq -3, \text{how}[3] \neq 3\}$

The modeller may need stronger evidence to start what can be a lengthy exploration (as clauses are in practice quite complex, even after renaming and simplification). The evidence can be much stronger if several clauses with a similar *pattern* can be found to have significantly reduced the search. For example, Shishmarev et al. [16] focused on clause 4 ($\{\text{how}[5] \neq -3, \text{how}[3] \neq 3\}$) not only because it was one of the top clauses and it was short (and thus easier to understand), but importantly, because it was part of several *similar clauses* in the top 10 (which they identified as clauses 5, 6, 7, and 10).

It is easy to show that these five clauses share the same *pattern*: $\{\text{how}[A] \neq -B, \text{how}[C] \neq B\}$, where A and C are pizzas, and B is a voucher. For example, clause 4 can be obtained by mapping $A/5$, $C/3$, and $B/3$. In fact, with the simplifications provided above, all clauses in Table 1 except 1 and 3 can be shown to share this pattern *and* to come from the same constraint. Grouping clauses with the same pattern can help modellers in two ways: (a) while individual clauses may not seem to contribute much to the overall search reduction, a group of clauses with the same pattern may do so; and (b) a pattern may identify a general constraint, i.e., something that is true for a whole range of parameters, and may therefore suggest a redundant constraint that can be added to the model.

We define a *pattern* to be the most specific (or least) generalisation [12] (MSG) of a set of clauses. To automatically obtain patterns, our method first renames and simplifies each clause. Then, it sorts the literals in each clause by variable name, operator and constant value. Finally, for all clauses that have the same sequence of literal operators, and the same sequence of variable types, it computes the MSG for all its subsets. For example, the following two (renamed, simplified and sorted) clauses for our `freepizza` instance:

```

{how[1]==-1,how[2]==-1,how[3]==-1,how[4]==-1,how[6]≠1} and
{how[1]==-2,how[2]==-2,how[5]==-2,how[7]==-2,how[6]≠2},

```

have the same sequence of literal operators ($=, =, =, =, \neq$) and the same sequence of variable types ($\text{how}[_], \text{how}[_], \text{how}[_], \text{how}[_], \text{how}[_]$). Our method computes the pattern $\{\text{how}[1] = -A, \text{how}[2] = -A, \text{how}[B] = -A, \text{how}[C] = -A, \text{how}[6] \neq A\}$, which can be mapped back to the clauses by applying the name/constant maps $\{A/1, B/3, C/4\}$ and $\{A/2, B/5, C/7\}$, respectively, where $\{B, C\}$ are known to be pizzas and $\{A\}$ a voucher. However, neither of the two clauses would form a pattern with $\{\text{how}[1] = -2, \text{how}[2] = -2, \text{how}[6]$

$\neq 2\}$, as they have a different number of literals, or with $\{\text{how}[1]=-1, \text{how}[2]=-1, \text{how}[3]=-1, \text{used}[1]=\text{true}, \text{how}[6]\neq 1\}$, as they have different variable types.

Formally, each clause Cl is stored as a tuple $(Id, SeqLits, Red, Cons)$ containing the clause identifier Id , its renamed, simplified and sorted sequence of literals $SeqLits$, its associated search reduction Red , and the constraints $Cons$ that generated it. A pattern Pt is stored as a tuple $(PId, PSeqLits, Maps, PRed, PCons)$ containing the pattern identifier PId , its sequence of literals $PSeqLits$, a set $Maps$ of tuples of the form (Map, Id) , where applying Map to $PSeqLits$ yields the sequence of literals in the clause identified by Id , the total search reduction $PRed$ achieved by its clauses (the sum of the Red of each clause identified by $Maps$), and the set of constraints $Cons$ that generated any of its clauses (the union of the $Cons$ of each of the clauses identified by $Maps$).

Note that a clause may appear in many different patterns. In fact, this will often be the case, as our algorithm can be seen as computing a pattern for each subset of the set of (renamed, simplified and sorted) clauses that have the same sequence of (a) literal operators and (b) variables types. The algorithm starts with the set of renamed, simplified and sorted *Clauses* computed for a given instance as described in previous sections, and an empty set of *Patterns*. Then, for every clause $Cl = (Id, SeqLits, Red, Cons)$ in *Clauses*, it does the following:

1. Transform Cl into pattern $Pt = (PId, SeqLits, \{([\], Id)\}, Red, Cons)$, where PId is a new identifier.
2. Set *NewPatterns* to \emptyset
3. For each pattern $Pt' = (PId', PSeqLits, Maps, PReds, PCons)$ in *Patterns* with the same sequence of literal operators and variable types as $SeqLits$:
 - (a) Find a most specific generalisation $PSeqLits'$ for $SeqLit$ and $PSeqLits$, with associated mapping Map .
 - (b) Add $(PId', PSeqLits', Maps \cup \{(Map, Id)\}, PRed + Red, PCons \cup Cons)$ to *NewPatterns*.
4. Merge *Patterns* and *NewPatterns*.
5. Add Pt to *Patterns*.

Note that while the pattern Pt added in step 5 above is known to be different from all others in *Patterns* (since Pt is essentially a clause and no two clauses are identical), it is possible to have a pattern P_1 in *NewPatterns* with a sequence of literals identical (up to variable renaming Ren) to pattern P_2 in *Patterns*. We can detect this when merging *Patterns* and *NewPatterns* in step 4, and simply merge P_1 and P_2 (by first applying Ren to them, and then computing the unions of their mapping and constraint sets, and summing up their reductions).

Once all patterns are computed, our method ranks the results by the amount of reduction associated with each pattern and presents it to the modeller. This allows modellers to notice clauses that may not result in significant search reductions when considered individually, but do when considered together.

3.5 Inferring Facts for Clause Patterns

While patterns are a clearer way to condense the information provided by several clauses, the insights obtained by Shishmarev et al. included information that was not present in the clauses. For example, for clause $\{\text{how}[5]\neq -3, \text{how}[3]\neq 3\}$

they considered the fact that pizza 3 is more expensive than pizza 5, and for $\{\text{how}[1]=-1, \text{how}[2]=-1, \text{how}[3]=-1, \text{how}[4]=-1, \text{how}[5]=-1, \text{how}[1]=-2, \text{how}[2]=-2, \text{how}[3]=-2, \text{how}[4]=-2, \text{how}[5]=-2, \text{how}[6] \leq 0, \text{how}[6] \geq 3\}$, the fact that pizza 6 is more expensive than any other pizza in the clause.

When generalising a set of clauses to a pattern, these additional facts can serve as *conditions* under which the pattern is indeed a valid (implied) constraint. For example, let us again consider the pattern $\{\text{how}[A] \neq -B, \text{how}[C] \neq B\}$ or, in implication form, $\text{how}[A] = -B \rightarrow \text{how}[C] \neq B$. This pattern is clearly not valid for arbitrary values of A , B , and C . However, for all clauses that contributed to the pattern (including clauses 2,4–10), it is true that pizza A is cheaper than pizza C . This yields a valid constraint: if pizza A is cheaper than pizza C , then for any voucher B , if we use A with voucher B , then we cannot get pizza C for free with voucher B . We have thus inferred the following constraint, which can be added to the model (expressed as the contrapositive of the statement above to make it more similar to the original formulation):

```
constraint forall(A,C in PIZZA, B in VOUCH) ((how[A]=-B /\ how[C]=B)
-> price[C]<=price[A]);
```

This constraint is logically equivalent to line 14 of the model, but it provides stronger propagation, since it explicitly states that we cannot get pizza C for free with voucher B , rather than stating this requirement indirectly. The manual analysis in [16] resulted in a different reformulation:

```
constraint forall(A,C in PIZZA) ((0 < how[C] /\ how[A]= -how[C])
-> price[C] <= price[A] );
```

While they are equivalent, the automatically derived constraint is at a lower level of abstraction, as it is stated for each individual voucher B , rather than as a general statement on the `how` variables. An interesting direction for future research is how to enable the automatic approach to perform such generalisations.

Automating the extraction of these facts is challenging, since it requires finding relations in the model's parameters that are true for a set of clauses. We have implemented an approach that extracts *relevant* facts for a certain pattern. A fact is relevant for a pattern if it concerns objects from it, e.g., if it relates the prices of pizzas mentioned in a pattern, or the `buy` and `free` values of a voucher mentioned in a pattern. The algorithm proceeds as follows for each pattern:

1. Extract the objects in the pattern; e.g., $\{\text{how}[A] \neq -B, \text{how}[C] \neq B\}$ refers to pizzas A and C , and voucher B .
2. Extract parameter values relating to all objects referred to by the clauses in the pattern. For the example above, infer prices for all pizzas A and C in any clause of the pattern, and `buy/free` information for all vouchers B .
3. For the extracted parameters, infer facts as simple unary relations on the objects, such as $p[X] = 0$, $p[X] < 0$ or $p[X] = \max(p)$, for each parameter p that is indexed by the type of object X . When there are two objects X and Y of the same type in a pattern, additionally infer facts as binary relations such as $p[X] = p[Y]$, $p[X] \neq p[Y]$, $p[X] \leq p[Y]$, $p[X] = -p[Y]$.

For the example with the most expensive pizza 6 above, we infer the fact that $\text{price}[A] = \max(\text{price})$ if a pattern exists where pizza A is the most expensive one. For the example comparing two pizzas, we will infer that

$\text{price}[C] > \text{price}[A]$ for all clauses in the pattern $\{\text{how}[A] \neq B, \text{how}[C] \neq B\}$. Since patterns may contain many clauses, the extracted facts are unlikely to always be true for all clauses. We therefore compute, for each extracted fact, the *percentage* of clauses in the pattern that it matches, and only report facts that match above a certain threshold. This is clearly a very incomplete and costly method, and it will be interesting to pursue further research into how more general facts can be extracted automatically in a reasonable amount of time.

3.6 Finding Patterns Across Searches and Across Instances

Finding a pattern whose cumulative search reduction is significant for a given instance, provides the modeller with some confidence regarding the importance of the pattern and the possibility of generalising it to a model constraint. Confidence will be stronger if clauses with the same pattern appear in different instances of the same model, as this suggests the pattern and its associated information holds across different input data and is, thus, more likely to be a general property of the model. Confidence would also be stronger if the pattern significantly reduced the search across different searches, as this would indicate the associated model modification may lead to speed-ups for all those searches.

To achieve this we have implemented a simple algorithm that, for each instance, takes the union of all clauses obtained by the given set of searches, and then finds the patterns (and facts) associated to them. For each identified pattern, it then computes the percentage of instances that have produced this pattern (up to renaming). Applying this method to the results obtained with several instances and searches of the `freepizza` model, we found that the pattern $\{\text{how}[A] \neq B, \text{how}[C] = -B, \text{how}[D] = -B, \text{how}[E] = -B, \text{how}[F] = -B, \text{how}[G] = -B\}$ appears in at least two instances and most of their searches, with information $\text{price}[A] > \text{price}[C], \text{price}[A] > \text{price}[D], \text{price}[A] < \text{price}[E], \text{price}[A] < \text{price}[F]$, and $\text{price}[A] > \text{price}[G]$, having percentages between 79% and 55%. The pattern states that if pizza A is free with voucher B, then one of the other pizzas needs to be paid for with the same voucher. It is a weaker version of clause 1 in Table 1 and indicates that pizza A can never be obtained for free by paying the cheaper pizzas. The fact it appears so often suggests exploring a new constraint that, for each pizza p , considers the subset of pizzas cheaper (or more expensive) than p .

Note however that the absence of a clause in a particular execution does not mean the clause (and thus its associated pattern) is not valid. We therefore need to be somewhat cautious when considering the importance of the patterns obtained by the above method. Alternatively, we could modify our method to do something similar to that of [8], by testing for each pattern Pt that does not appear in some of the executions, whether at least one of its instantiations holds.

4 Preliminary Case Studies

While automating the method, we discovered new examples where the approach is applicable. This section explores two such cases.

4.1 Case Study 1: Redundant Constraints

Our semi-automated approach allowed us to discover a redundant constraint that can be added to a model of the Grid Colouring problem. A grid colouring is a colouring of an $n \times m$ grid x , where no sub-rectangle of the grid has all of its corner cells assigned the same colour. A simplified version of the MiniZinc model for this problem is presented below. The loop on lines 4 – 8 enumerates all sub-rectangles of the grid, and states that at least one pair of orthogonally adjacent corners must be assigned distinct colours.

```

1  int: n; int: m; % Width and height of grid
2  array[1..n,1..m] of var 1..min(n,m): x; % Colour in each cell
3
4  constraint forall(i in 1..n, j in i+1..n, k in 1..m, l in k+1..m)(
5      ( x[i,k]!=x[i,l] \/\ x[i,l]!=x[j,l]
6        \/\ x[j,k]!=x[j,l] \/\ x[i,k]!=x[j,k] );
7
8  solve minimize max(x); % Number of colours used

```

Looking at the clauses produced for instances of this problem, we found a frequent, high-ranking pattern with literals $\{x[A,B] \neq x[A,C], x[A,B] \neq x[D,B], x[A,C] \neq E, x[D,B] \neq E, x[D,C] \neq E\}$. One interpretation of the pattern states that if corner $x[A,B]$ is the same colour as $x[A,C]$ and $x[D,B]$, then one of $x[A,C], x[D,B]$ or $x[D,C]$ must be assigned a different colour. Upon examination, it became clear that the constraints in the model only indirectly compared diagonally adjacent corners. To address this weakness, we added the following constraints:

```

constraint forall(i in 1..n, j in i+1..n, k in 1..m, l in k+1..m)
((x[i,l]=x[j,k] /\ x[i,l]=x[j,l]) -> (x[i,k]!=x[i,l]))

```

These constraints did not improve Chuffed’s performance but did improve Gecode’s significantly. Table 2 shows Gecode’s solve time (in seconds) and node count for several instances of the original model and the modified one. The number of extra constraints result in Gecode spending more time at each node but the added propagation leads to faster solve times.

Table 2. Gecode’s solving time for different instances of the Grid Colouring problem.

n m	Original		Modified	
	time (s)	nodes	time (s)	nodes
5 5	0.10	34,987	0.07	5,452
5 6	0.13	35,223	0.09	5,468
6 6	0.55	131,661	0.29	16,773
6 7	1.53	9,484,042	0.77	37,727
7 7	65.47	11,565,900	23.15	904,148

4.2 Case Study 2: Strengthening Model Constraints

Our method also helped us discover a constraint in the time-changing graph colouring problem model that could benefit from domain (rather than the commonly used bounds) consistency [1]. In this problem, the given initial colouring of

a graph must be transitioned to a given final colouring. The transition requires a certain number s of steps, each performing at most k modifications to the colours while maintaining a valid graph colouring. The objective is to first minimise the number of steps required, and then the number of modifications.

The following shows an extract from the MiniZinc model we used, which appeared in the annual MiniZinc Challenge as `tcgc2`¹. The model takes as input, among others, the maximum number k of transformations per step, and sets the maximum number of steps `max_s` to 10. It has an array of decision variables, where `a[i,n]=j` represents the colour j of node n in step i , and a decision variable `s` representing the final number of steps. The constraint displayed states that in every non-final step i , the sum of all transformations must be less or equal to k .

```

int: k;                                % maximum colour changes per step
int: max_s = 10;                        % maximum number of steps
array [STEPS,NODES] of var COLORS: a; % Colours of nodes at each time step
var 2..max_s: s;                        % final number of steps
...

51 constraint forall(i in 1..max_s-1)
52     (i < s -> sum (n in NODES) ( a[i,n] != a[i+1,n] ) <= k);
53 ...

```

Our method identified pattern $\{a[A,B] \neq a[A+1,B], a[A,B]=C, a[A+1,B] \neq C\}$ as interesting, since its clauses are highly ranked across different searches of different instances and, when combined, are responsible for a large search reduction. Further, the pattern is short (and thus easy to understand) and all its clauses come from a single model constraint (which often indicates lack of expected propagation by the constraint). Upon examination, we felt the pattern stated something so simple it should have already been captured by the propagation of the model constraints. The first literal, derived from the expression highlighted on line 52, represents the result of a reified not-equals constraint. The literal is true when the variables take different values. If this is false, the remaining literals state that either the variables both take the value C or they take some other value (this becomes clear when presented in implication form: $\{a[A,B]=a[A+1,B] \wedge a[A,B] \neq C \rightarrow a[A+1,B] \neq C\}$ and $\{a[A,B]=a[A+1,B] \wedge a[A+1,B]=C \rightarrow a[A,B]=C\}$). While this information is obvious, the fact that the associated nogoods were being reported by the solver indicated propagation was not performing as expected. We then realised that the implementation of a reified not-equals constraint in Chuffed is bounds consistent rather than domain consistent. To resolve this, we manually modified the model to add the following information:

```

48 array[1..max_s-1,NODES] of var bool: aa;
49 constraint forall (i in 1..max_s-1, n in NODES)
50     (aa[i,n] <-> forall (c in COLORS) (a[i,n]=c <-> a[i+1,n]=c));
51 constraint forall (i in 1..max_s-1)
52     (i < s -> sum (n in NODES) ( (not aa[i,n] ) ) <= k);
53 ...

```

Table 3 shows solve times for instances of the original and modified model. Three different search strategies with a 5 minute time limit were executed. The strategies were the fixed strategy defined in the model, Chuffed's free

¹ <https://github.com/MiniZinc/minizinc-benchmarks/tree/master/tc-graph-color>.

search strategy, and, one that alternated between the two. The results show the constraint can improve Chuffed’s solving performance on some (but not all) instances of the problem. For a traditional CP solver with a domain consistent reified not-equals constraint, these extra constraints will slow down propagation, and an annotation indicating the need for domain propagation would be preferred. This shows the need for the modeller’s input.

Table 3. Solving times for different instances of `tcgc2` using different search strategies.

Instance	Fixed		Free		Alternate	
	Original	Modified	Original	Modified	Original	Modified
k5.5	48.48	40.54	83.68	86.20	66.65	86.62
k9.39	∞	∞	∞	∞	262.28	295.16
k10.31	∞	∞	∞	∞	78.35	63.81
k10.34	23.66	21.18	80.65	91.60	77.57	69.59
k10.41	1.67	1.99	17.80	19.60	3.15	3.25

5 Status and Limitations

The semi-automatic method presented in the previous sections is a significant improvement over the manual method presented in [16], making it much easier and effective to explore the impact of the clauses learnt by a learning solver, and the possible model modifications these clauses suggest. However, the current implementation still has important limitations that are worth exploring. Here we present some of these limitations and point to possible ways to address them.

Variable Paths: Our approach for reconstructing expressions based on paths is purely textual. Thus, the paths for variables added by the compilation of a function/predicate call currently appear connected to the names of the parameters or the local variables of the function/predicate. For example, a decomposition of the `alldifferent(array[int] of var int: x)` predicate (from the MiniZinc library) will contain expression `x[i]!=x[j]`. If the model contains a call to this predicate, such as `alldifferent(y)`, our purely textual approach would not be able to rename `x` to `y` in any clauses resulting from this constraint. Therefore, modellers cannot distinguish between different invocations of the same predicate.

We plan to address this limitation by using the MiniZinc compiler to hoist local variables and parameters to be expressed in terms of the top level model variables where possible, to apply simplification rules, and to deduce accurate information about the types of expressions we have extracted.

Finding Patterns: Our current implementation has three main limitations. First, to compute the patterns of a given model we must manually provide the names of the decision variables that can appear in the literals, their possible values and their types. This is needed to correctly (a) parse the clauses produced either by Chuffed or by our path-based renaming process, and (b) obtain the most specific generalisation for two clauses. This manual step can be resolved by

integrating our method with the MiniZinc compiler, so that the required information can be extracted directly from it. Second, our current implementation can only handle very simple expressions in the clause literals, even though the paths for introduced variables can result in complex ones.

Finally, and importantly, the current implementation of the most specific generalisation of a pattern is very limited, as it (a) only matches sequences of literals that have the same sequence of operators and variable types, and (b) only matches the i th literal in each pair of clauses. Due to (a) we might miss a relationship between clauses that have different numbers of literals but share a *parametric* pattern. For example, clauses $\{w[1]<3, w[2]<3\}$, $\{w[1]<3, w[2]<3, w[3]<3\}$ and $\{w[1]<3, w[2]<3, w[3]<3, [w4]<3\}$, share the pattern $\{w[x]<3 \mid x \in 1..n\}$. Due to (b) we might miss patterns that were obscured by the sorting or renaming of literals. For example, for clauses $\{w[10]=1, w[20]=2\}$ and $\{w[40]=2, w[50]=1\}$ we currently only find a pattern with sequence of literals $\{w[A]=B, w[C]=D\}$, and maps $\{A/10, B/1, C/20, D/2\}$ and $\{A/40, B/2, C/50, D/1\}$, while matching the first and second literal of each clause would yield a more specific pattern with sequence: $\{w[A]=1, w[B]=2\}$ and maps $\{A/10, C/20\}$ and $\{A/50, C/40\}$. More sophisticated algorithms (such as the anti-unification of [5]) can handle different numbers of literals or matching any literals with the same operator. However, given the computational cost, it might only be worth it for top-ranking clauses.

Finding Facts: This is one of the most difficult and interesting areas for future research. As described above, our current implementation blindly collects all simple relations between the parameter values that can be associated to objects appearing in a clause. While the resulting information is a small subset of the knowledge space, it is still costly to compute. Furthermore, other knowledge may be missed (e.g., arbitrary parametric expressions not explicitly present in the original constraints), which could help modellers understand the origin of the clause and point to better ways to improve the model.

6 Conclusions

This paper presented improvements to and steps towards the automation of the methods presented in [16]. In particular, we have shown how the literals in learnt clauses can be automatically renamed, simplified and both textually and visually tied back to the parts of the model they originated from. This will make it easier for modellers to understand the clauses. We have then shown how the resulting clauses can be grouped by generic patterns, and how to derive information regarding the objects present in these patterns. Further, we have shown how these patterns can be inferred across searches and across instances. This will help modellers determine the likelihood of a pattern being generalisable to the model to either strengthen a constraint or as a new, redundant constraint. We have also shown how the information associated with the patterns will help modellers determine the particular strengthening that should be applied, or the new constraint that should be added. We have provided two new case studies that show the applicability of the approach. Finally, we have identified several significant challenges our method still faces, and potential avenues for future

research. We believe that by further automating this methodology, and potentially taking advantage of other learning solvers (such as cuts in MIP solvers), we will be able to provide tools that significantly simplify the task of finding model refinements for users of the MiniZinc toolchain.

Acknowledgements. This research was partly sponsored by the Australian Research Council grant DP180100151.

References

1. Choi, C.W., Harvey, W., Lee, J.H.M., Stuckey, P.J.: Finite domain bounds consistency revisited. In: Sattar, A., Kang, B. (eds.) AI 2006. LNCS (LNAI), vol. 4304, pp. 49–58. Springer, Heidelberg (2006). https://doi.org/10.1007/11941439_9
2. Chu, G.G.: Improving combinatorial optimization. Ph.D. thesis, The University of Melbourne (2011)
3. Feydy, T., Stuckey, P.J.: Lazy clause generation reengineered. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 352–366. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_29
4. Frisch, A., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: a constraint language for specifying combinatorial problems. *Constraints* **13**(3), 268–306 (2008)
5. Kutsia, T., Levy, J., Villaret, M.: Anti-unification for unranked terms and hedges. *J. Autom. Reasoning* **52**(2), 155–190 (2014)
6. Leo, K., Tack, G.: Multi-pass high-level presolving. In: Yang, Q., Wooldridge, M. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, 25–31 July 2015, pp. 346–352. AAAI Press (2015). <http://ijcai.org/proceedings/2015>
7. Leo, K., Tack, G.: Debugging unsatisfiable constraint models. In: Salvagnin, D., Lombardi, M. (eds.) CPAIOR 2017. LNCS, vol. 10335, pp. 77–93. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59776-8_7
8. Mears, C., Garcia de la Banda, M., Wallace, M., Demoen, B.: A method for detecting symmetries in constraint models and its generalisation. *Constraints* **20**(2), 235–273 (2015)
9. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, pp. 530–535. ACM (2001)
10. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_38
11. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation = Lazy Clause Generation. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 544–558. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_39
12. Plotkin, G.D.: A note on inductive generalization. *Mach. Intell.* **5**(1), 153–163 (1970)
13. Schulte, C., Tack, G., Lagerkvist, M.Z.: Modeling and programming with Gecode (2016). <http://www.gecode.org>
14. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Why cumulative decomposition is not as bad as it sounds. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 746–761. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_58

15. Schutt, A., Stuckey, P.J., Verden, A.R.: Optimal carpet cutting. In: Lee, J. (ed.) Principles and Practice of Constraint Programming - CP 2011, pp. 69–84. Springer, Heidelberg (2011)
16. Shishmarev, M., Mears, C., Tack, G., Garcia de la Banda, M.: Learning from learning solvers. In: Rueher, M. (ed.) CP 2016. LNCS, vol. 9892, pp. 455–472. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44953-1_29
17. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The MiniZinc challenge 2008–2013. *AI Mag.* **35**(2), 55–60 (2014)



Finding Solutions by Finding Inconsistencies

Ghiles Ziat¹(✉), Marie Pelleau², Charlotte Truchet³, and Antoine Miné¹

¹ Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6,
75005 Paris, France
ghiles.ziat@gmail.com

² Université Côte d'Azur, CNRS, I3S, Nice, France

³ TASC LS2N UMR 6004, Univ., Nantes, France

Abstract. In continuous constraint programming, the solving process alternates propagation steps, which reduce the search space according to the constraints, and branching steps. In practice, the solvers spend a lot of computation time in propagation to separate feasible and infeasible parts of the search space. The constraint propagators cut the search space into two subspaces: the inconsistent one, which can be discarded, and the consistent one, which may contain solutions and where the search continues. The status of all this consistent subspace is thus indeterminate. In this article, we introduce a new step called *elimination*. It refines the analysis of the consistent subspace by dividing it into an indeterminate one, where the search must continue, and a *satisfied* one, where the constraints are always satisfied. The latter can be stored and removed from the search process. Elimination relies on the propagation of the negation of the constraints, and a new difference operator to efficiently compute the obtained set as an union of boxes, thus it uses the same representations and algorithms as those already existing in the solvers. Combined with propagation, elimination allows the solver to focus on the frontiers of the constraints, which is the core difficult part of the problem. We have implemented our method in the AbSolute solver, and present experimental results on classic benchmarks with good performances.

1 Introduction

Constraint solvers generally alternate two steps: propagation and exploration. The propagation step reduces the domains of the variables using the constraints. The exploration step adds hypotheses to divide the problem into several smaller sub-problems. In this article, we are interested in continuous constraint solving, where the variables have real values. In this case, the resolution of a problem usually consists in a paving of the solution space, which is not computer representable in general, using elements which are simple enough to manipulate (often floating-point boxes). This paving may correspond to an outer approximation or

The work was supported, in part, by the project ANR-15-CE25-0002 Coverif from the French Agence Nationale de la Recherche, and in part by the European Research Council under Consolidator Grant Agreement 681393 – MOPSA.

over-approximation of solutions, as in Ibex [6], or may correspond to an inner approximation or under-approximation as in [7].

The efficiency of a solver depends on the choices made by the exploration process, these choices being often guided by heuristics. On discrete variables, such heuristics can for example try and provoke early failures (such as *fail-first* [9] or *dom /w deg* [4]).

On continuous variables, classic heuristics include: *largest first* [13], which consists in splitting the largest domain; *round robin*, where the domains are processed successively; or *maximal smear* [8], choosing the domain with the greatest slope based on the derivatives of the constraints. More recently, and closer to our work, *Mind The Gaps* [1] uses the idea from [8, 13] and uses partial consistencies to find interesting splitting points within the domain, according to the “gaps” in the search space: splitting the domains by taking into account such gaps reduces the search space.

In this paper, we focus on covering the entire solution space of continuous problems. We propose to add a new step, complementary to constraint propagation, in the solving process: the *elimination* step. This step divides the search space into two sub-spaces: one containing only solutions, and the other where the constraints are indeterminate—it may contain solutions as well as non-solutions. Our solving method alternates three steps: propagation, elimination, and exploration. It offers another way of reasoning on the constraints, since we are not only exploiting the constraints’ consistencies (as does propagation) but also the constraint inconsistencies. With this improved reasoning, the interesting zones of the search space are better targeted: zones without solutions are discarded by propagation, and zones with only solutions are set aside by elimination into the solution space, which means in practice that they also are excluded from the search. The search effort can then focus on the indeterminate space—the part of the search space effectively requiring deeper exploration by the solver.

Our new step can be seen as a new contraction, in the same framework as the contractors described in [10] and used in Ibex [6] to perform a smarter exploration. We add an automatic propagation on the negation of the constraints, to identify subspaces containing only solutions. We thus reason on the negation of the constraints, hence we compute sets which are not boxes: to overcome this issue, we also add an operator on boxes to efficiently compute the difference of two boxes (or the complementary of one box in another) as a union of boxes. Thus, our method can be integrated into any solver without changing its domain representation nor modifying the propagators.

Our elimination phase relies on a notion of consistency to divide the search space and guide the search, similarly to *Mind The Gaps* [1] where consistency is also used to guide the search. But we go a step further by not only trying to identify the inconsistent parts of the search space (the “gaps”), but by using set complement to identify sub-spaces containing only solutions.

Our method is tailored to output an outer approximation as well as an inner approximation of the solution set: when the size of the indeterminate part is small enough and exploration stops, we can either include the indeterminate

part with the definite solution space found by elimination steps to get an outer approximation, or return only the solution space found by elimination which is an inner approximation.

In fact, our solver can provide within the same process both an inner and an outer approximation, and due to the fact that the computed boxes better fit the constraints' shapes, this comes at no cost according to our experiments. We have tested our method on both the Coconut [16] and the MinLPLib [5] benchmarks. Our results show that first, our method computes both the inner and outer approximation with no time overhead, and second, it produces fewer boxes as an output, which makes the computed solution much more tractable.

This paper is organized as follows. Section 2 presents formally classic continuous constraint solving, on which our work is based. Section 3 introduces our new solving step: elimination. Section 4 presents experiments with our new solving method. Finally, Sect. 5 concludes and discusses future work.

2 Preliminaries

This section recalls basic notions of continuous Constraint Programming (CP). For a more detailed presentation, we refer the reader to [14, Chap. 16].

2.1 Constraint Satisfaction Problems

We consider a Constraint Satisfaction Problem (CSP) defined by: a set of n variables $\mathcal{X} = \{x_1, \dots, x_n\}$; the domain of each variable $\mathcal{D} = \{d_1, \dots, d_n\}$, i.e., $x_k \in d_k, \forall k \in [1, n]$; and a set of m constraints $\mathcal{C} = (C_1, \dots, C_m)$. A possible assignment of the variables is a tuple in $D = d_1 \times \dots \times d_n$. A solution of the CSP is an element of D satisfying all the constraints in \mathcal{C} . We denote as \mathcal{S} the set of all solutions, i.e., $\mathcal{S} = \{(s_1, \dots, s_n) \in D \mid \forall i \in \{1, \dots, m\}, C_i(s_1, \dots, s_n)\}$. We also denote as \mathcal{S}_C the solution set for the constraint C alone: $\mathcal{S}_C = \{(s_1, \dots, s_n) \in D \mid C(s_1, \dots, s_n)\}$.

In the CP framework, variables can either be discrete or continuous. In this article, we focus on continuous, real-valued variables. Domains of variables are intervals of \mathcal{R} . We also assume that the bounds are (finite) floating point numbers, to be computer-representable. They can be either excluded or included. Let \mathcal{F} be the set of finite floating point numbers. For $a, b \in \mathcal{F}$, we define a real-interval as the conjunction of two half-spaces $\{x \in \mathcal{R} \mid a \triangleleft x \triangleleft b\}$ where $\triangleleft \in \{<, \leq\}$, and let \mathcal{I} be the set of all such intervals.

A Cartesian product of intervals is called a box. We note $\mathcal{B} = \mathcal{I}^n$ the set of boxes of dimension n . Note that our definition of interval encompasses intervals with excluded end-points which will be useful later.

For continuous CSPs, with domains in \mathcal{I} , the exact solution set $\mathcal{S} \subseteq \mathcal{R}^n$ is generally not computer-representable. Constraint solvers usually return a collection of boxes with floating-point bounds containing the solutions, the union of these being an over-approximation of \mathcal{S} .

2.2 Consistency

The notion of *local consistency* is central in CP. We recall the definition of Hull-consistency [3], one of the classic local consistencies for continuous constraints.

Definition 1 (Hull-Consistency). *Let x_1, \dots, x_n be variables over continuous domains represented by intervals $d_1, \dots, d_n \in \mathcal{I}$, and C a constraint. The domains are said to be Hull-consistent for C if and only if $D = d_1 \times \dots \times d_n$ is the smallest floating-point box containing the solutions for C in D .*

Intuitively, no bound of a consistent box D can be tightened without losing a solution of C . Given a constraint C over domains d_1, \dots, d_n , an algorithm that computes locally consistent domains d'_1, \dots, d'_n that contain the same solution set as C in $d_1 \times \dots \times d_n$ is called a *propagator* for C . Naturally, $\forall k \in [1, n], d'_k \subseteq d_k$. Given a constraint C and domains d_1, \dots, d_n , we will write $H_C(d_1, \dots, d_n)$ the corresponding Hull-consistent domains and $\rho_C : \mathcal{B} \rightarrow \mathcal{B}$ a propagator for C . While we only refer to the Hull-consistency in this work, our method is based upon the propagator notion and holds for any kind of consistency.

The domains which are locally consistent for all constraints are the largest common fixpoints of all the constraint propagators [2, 15]. In practice, propagators often compute over-approximations of the locally consistent domains. In the following, we will use the standard algorithm HC4 [3], which propagates continuous constraints, relying on the syntax of the constraints and interval arithmetic [11], although our method could be combined with other propagators. HC4 generally does not reach Hull consistency, in particular in case of multiple occurrences of the variables in the constraints.

Local consistency computations can be seen as deductions, performed on domains by analyzing the constraints. If the propagators return the empty set, the domains are inconsistent and the problem has no solution. Otherwise, non-empty local consistent domains are computed. This is often not sufficient to accurately approximate the solution set. In that case, choices are made on the variable values. For continuous constraints, typically a domain d is chosen and split into two (or more) parts, which are in turn narrowed by the propagators. The solver alternates propagation and split phases a given precision is reached, *i.e.* all the boxes which are still considered are smaller than a given parameter. Of course, as soon as a box is proven to contain only solutions, it can be removed from the search space and added to the solution set. Upon termination, the collection of boxes returned covers the solution set \mathcal{S} , under some hypotheses on the propagators and splits [2].

A solving method is said to be *complete* if it returns an over-approximation of the solution set (no solution is missed). It is said to be *sound* if it returns an under-approximation of the solution set (only solutions are returned). For problems with real variables, the solving method cannot be both complete and sound in general as being sound (returning only solution), requires the result to under-approximate the solution space, and being complete (returning all the solutions) requires the result to over-approximate the solution space. In practice, solving methods are often complete and not sound.

Algorithm 1. Solving without / with elimination (in pink)

```

1: function SOLVE( $\mathcal{D}, \mathcal{C}, r, \text{elim}$ )   $\triangleright$   $\mathcal{D}$ : domains,  $\mathcal{C}$ : constraints,  $r$ : real, set elim to
2:                                     false for classic solving, true for elimination
3:   sols  $\leftarrow \emptyset$   $\triangleright$  sound solutions
4:   undet  $\leftarrow \emptyset$   $\triangleright$  indeterminate solutions
5:   explore  $\leftarrow \emptyset$   $\triangleright$  boxes to explore
6:    $e = \text{init}(\mathcal{D})$   $\triangleright$  initialization
7:   push  $e$  in explore
8:   while explore  $\neq \emptyset$  do
9:      $e \leftarrow \text{pop}(\text{explore})$ 
10:     $e \leftarrow \text{filter}(e, \mathcal{C})$ 
11:    if  $e \neq \emptyset$  then
12:      if satisfies( $e, \mathcal{C}$ ) then
13:        sols  $\leftarrow \text{sols} \cup e$ 
14:      else
15:        if  $\tau(e) \leq r$  then
16:          undet  $\leftarrow \text{undet} \cup e$ 
17:        else
18:          if !elim then
19:            push  $\oplus(e)$  in explore  $\triangleright$  Classic solving process
20:          else
21:             $(S, E) = \text{elimination}(e, \mathcal{C})$   $\triangleright$  Solving with elimination
22:            sols  $\leftarrow \text{sols} \cup S$ 
23:            push  $\oplus(E)$  in explore

```

2.3 Solving Method

In this article, we rely on the general abstract solving process described in [12], instantiated with the interval domain. The solver thus operates on boxes, as defined above. Algorithm 1 gives the pseudo-code of the abstract solving method, where $\tau \in \mathcal{B} \rightarrow \mathbb{R}$ is the precision measure and $\oplus \in \mathcal{B} \rightarrow \wp(\mathcal{B})$ is the split operator. In this section we have the **elim** parameter set to false, thus we do not consider the part highlighted in pink. By alternating propagation and exploration, Algorithm 1 builds a disjunction of boxes that covers the solution space. It uses three auxiliary functions: **init** $\in \mathcal{D} \rightarrow \mathcal{B}$, **filter** $\in \mathcal{B} \rightarrow \mathcal{B}$, and **satisfies** $\in \mathcal{B} \times \mathcal{C} \rightarrow \{\text{true}, \text{false}\}$. Firstly, **init** creates a box from the initial domains of the problem. Then, **filter** corresponds to the propagation loop: it applies the propagator for each constraint in turn. Finally, **satisfies** checks whether a box satisfies all the constraints, that is, if it contains only solutions. This function corresponds to a contractor as defined in [6].

This solving method works as follows: at each step, the current box is tightened using the propagators on the constraints (function **filter**). After propagation, if the tightened box is not empty, three cases are possible:

- If the box contains only solutions (function **satisfies**), then it is directly added to the set of solutions **sols**.

- Otherwise, if the box is small enough with respect to a parameter r ($\tau(e) \leq r$), then it is added to the set of indeterminate solutions **undet**—i.e., the box, which may contain both solutions and non-solutions, is considered small enough to be left out of the search.
- Finally, if the size of the box is larger than r and may contain solutions, as **elim** is set to false, then it is divided using a split operator \oplus and the process is repeated on the resulting boxes.

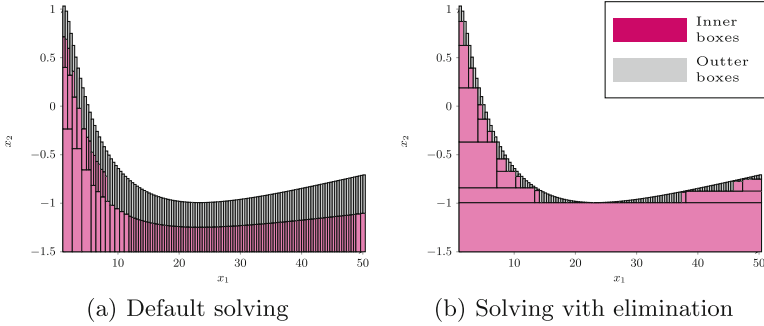


Fig. 1. (a): 127 inner boxes, 128 outer boxes. Inner boxes represent 59% of the coverage area. Computation time 0.015s. (b): 18 inner boxes, 128 outer boxes. Inner boxes represent 92% of the coverage area. Computation time: 0.008s

Figure 1(a) shows the result obtained with Algorithm 1 with **elim** set to false, for a problem with two variables $x_1 \in [1, 50]$ and $x_2 \in [-1.5, 1]$ constrained by $\cos(\ln(x_1)) > x_2$. Note that this solving method can either produce an under-approximation of the solution set by considering only the inner elements, or an over-approximation by considering all the resulting elements. Figure 1(b) shows that by making different splitting choices, we could avoid computations and reach the given precision with less iterations. We achieve that using Elimination, with which we obtain fewer, but larger inner boxes. We introduce this new step in the next section and explain how it pushes the reasoning based on the constraints one step further in order to avoid superfluous splitting steps.

3 Elimination

The propagation step reduces the search space by removing non-consistent subspaces. Elimination aims at reducing the search space by focusing on the frontiers of the problem. This is done in three steps: computing the elimination for each constraint, combining the result with the domains with a new difference operator, and finally integrate this mechanism in the solving process.

3.1 Elimination for One Constraint

We introduce here the concept of elimination for a single constraint. It relies on the constraint propagator to over-approximate the set of instantiations that *can not* be solutions. We will refer to these instantiations as inconsistent instantiations. By elimination, the rest of the search space can only contain solutions.

In the remainder of the subsection, given a constraint C and a box $D = d_1 \times \dots \times d_n$, we will write D_C the set of instantiations of D that satisfy C and $\overline{D_C}$ and, the—complementary—set of inconsistent instantiations w.r.t C . Thus, we have $\overline{D_C} = D \setminus D_C$. As $D_C, \overline{D_C}$ can be uncomputable, so we compute an over-approximation. For a single constraint, this can be achieved simply by reusing the propagation, over the negation of the constraint.

Definition 2. Let x_1, \dots, x_n be variables in domains d_1, \dots, d_n , and C a constraint on x_1, \dots, x_n . We define a function $\theta_C : \mathcal{B} \rightarrow \mathcal{B}$ such that $\theta_C(d_1, \dots, d_n) = \rho_{\neg C}(d_1, \dots, d_n)$.

Combining this function with propagation, we partition D relatively to the satisfiability of C . Let $S_C = \rho_C(d_1, \dots, d_n)$ and $\overline{S_C} = \theta_C(d_1, \dots, d_n)$ be respectively the over-approximation of D_C and $\overline{D_C}$, we differentiate three kinds of instantiations:

- the ones that belong to $\overline{S_C}$ and not to S_C , which are inconsistent,
- the ones that belong to S_C , and not to $\overline{S_C}$, which are consistent,
- the remaining ones that belong to both S_C and $\overline{S_C}$, which are indeterminate.

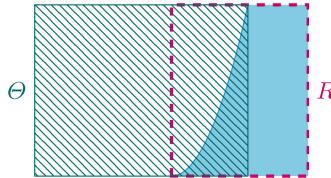


Fig. 2. Given the constraint $y \leq x^3$, in blue, the box R over-approximates the solutions and the hatched box Θ over-approximates the inconsistencies. (Color figure online)

Figure 2(b) shows an example of this partitioning. For the constraint $y \leq x^3$ (filled with blue), the box S_C (dashed), computed through propagation, over-approximates the solutions and the box Θ (hatched in green), computed by applying propagation over the negation of the constraint, over-approximates the inconsistencies. We can see that the complement of Θ under-approximates the set of solutions, while the complement of R under-approximates the set of inconsistencies. The intersection $\Theta \cap R$ can contain both solutions and inconsistencies.

Once this partitioning is done, the inconsistent part can be discarded (as usual) and the consistent one can be directly added to the set `sols` of solutions.

What remains is the indeterminate space in which the solving process continues. This principle is then generalized to the case of several constraints: the consistent part is the intersection of all the consistent parts associated to each constraint. Symmetrically, the inconsistent part is the union of all the inconsistent parts associated to each constraint. What remains is the indeterminate part.

Remark. In practice, in the case of continuous constraints, elimination can rely on the original propagation algorithms of the considered constraint, since we can easily compute the negation of a constraint (based on predicates $<, =, \leq$). It would also be valid for discrete constraints *provided that* the same property holds. Indeed, primitive constraints could be dealt with elimination, but handling global constraints would require to specifically define their negations and introduce dedicated propagators.

The indeterminate space is defined as an intersection of boxes, which results in a box. Hence, the solving process continues within a box, as in a classic propagation-based solver, except that the box is possibly smaller as we intersect the result of propagation with the result of elimination. However, $S_C \setminus \overline{S_C}$ is not necessarily a box. Computing this set difference requires taking the complement of a box relative to another box. In the following section, we define a set difference operator over boxes. It computes the difference as a set of boxes, that can be directly added to `sols`.

3.2 Difference Operator

Given two boxes B_1 and B_2 , their difference $B_1 \setminus B_2$ is not necessarily a box. However, we can express it as a collection of boxes that covers $B_1 \setminus B_2$. To guarantee a non-redundancy property over the result, this cover should be a partition. This would prevent boxes from overlapping and have instantiations covered by several boxes. However, a cover is sufficient to have a sound and complete resolution method, and is easier to build as we will see in the current section. Our difference operator should satisfy the following properties:

Definition 3. (Difference operator). A difference operator $\ominus : \mathcal{B} \times \mathcal{B} \rightarrow \wp(\mathcal{B})$ is a binary operator such that $\forall B_1, B_2 \in \mathcal{B}$:

- (1) $|B_1 \ominus B_2|$ is finite;
- (2) $\forall b \in (B_1 \ominus B_2) \Rightarrow b \cap B_2 = \emptyset$;
- (3) $B_1 = (B_1 \cap B_2) \cup \bigcup \{b \in B_1 \ominus B_2\}$.

The first condition ensures that the solving method produces a finite set of boxes. The second one ensures that the operator eliminates from the box B_1 the values inside the box B_2 . Finally, the third condition guarantees that the difference of B_1 and B_2 , union B_2 , covers the initial box B_1 . The second condition is related to soundness and the third one to completeness.

Our difference operator on boxes works with constraints. A box can be defined as a conjunction of constraints $B = \bigwedge_{i=1,\dots,p} c_i$, where each constraint $c_i = \pm x_i \triangleleft a_i$, with $\triangleleft \in \{<, \leq\}$, gives a lower or an upper bound—not necessarily included—on x_i .

Note that it is mandatory to be able to express both strict and large inequalities. Otherwise, a problem would arise as the negation of $\pm x_i > a_i$ would not be exactly representable, and we would have no way to ensure property Definition 3.(2). As the difference operator is used to compute \mathcal{S} , an under-approximation of the set of solutions, adding to \mathcal{S} the closure of boxes which should actually be open, could add to it points that are not solutions to the problem, and thus break the soundness criterion.

Each c_i defines a half-space, and the difference between a box and a half-space is still a box. A first step is thus to compute the difference between two boxes, by considering each half space independently, as shown on Fig. 3(b).

Definition 4 (Difference for boxes). *Let B_1 and B_2 be two boxes, with B_2 represented as the set of constraints C_2 . The difference of B_1 and B_2 is:*

$$B_1 \ominus B_2 \triangleq \{B_1 \cap (\neg c) \mid c \in C_2\} \tag{1}$$

This naive method can result in widely overlapping boxes in the output. Nevertheless, it is an acceptable difference operator as it satisfies Definition 3:

- (1) $B_1 \ominus B_2$ returns a set that, associates a box to each constraint in B_2 . The number of constraints in B_2 is finite, hence this set is finite (Definition 3.(1)).
- (2) By definition of the intersection, the condition Definition 3.(2) is satisfied as each box in the result is included in B_1 .
- (3) Finally, Definition 3.(3) is also satisfied: $B_1 \ominus B_2$ can be rewritten as $B_1 \cap \overline{B_2}$. No solution is lost as B_1 is entirely covered by B_2 and $B_1 \ominus B_2$.

Figure 3 shows an example of the application of the difference operator on two boxes. Figure 3(a) gives the initial boxes B_1 and B_2 , with B_2 represented by the constraints $\{c_1, \dots, c_4\}$. Figure 3(a) shows the result of the naive difference operator. Here, $B_1 \setminus B_2$ is covered by three elements, one per constraint of C_2 , after removing the constraints that, intersected with B_1 , yield the empty set (c_4 in this case). Overlapping boxes in the output appear in a darker shades. This overlapping implies that some instantiations may be covered by more than one box: the result is redundant.

We now propose an improved difference operator in order to obtain non-overlapping boxes when building a partition of $B_1 \setminus B_2$.

Definition 5 (Non-redundant difference for boxes). *Let B_1 and B_2 be two boxes and B_2 is represented by the set of constraints $C_2 = \{c_1, \dots, c_p\}$. The difference of B_1 and B_2 is defined as:*

$$B_1 \ominus B_2 \triangleq \left\{ B_1 \cap (\neg c_i) \cap \bigcap_{j < i} c_j \mid i \in \{1, \dots, p\} \right\} \tag{2}$$

For similar reasons to the naive difference operator, Definitions 3.(1)–(3) is also satisfied for the non-redundant difference operator. Additionally, we strengthen the property that $B_1 \ominus B_2$ is a cover for $B_1 \setminus B_2$ by making this cover a partition, i.e., the elements of $B_1 \ominus B_2$ are pairwise disjoint: we ensure that, for any pair of boxes $b_i, b_j \in B_1 \ominus B_2$ such that $i \neq j$, we have $b_i \cap b_j = \emptyset$.

Proposition 1. $B_1 \ominus B_2$ is a partition of $B_1 \setminus B_2$.

Proof. If $|B_1 \ominus B_2| = 1$ then, trivially, $B_1 \ominus B_2$ is a partition of $B_1 \setminus B_2$. If $|B_1 \ominus B_2| > 1$, we have to prove that the elements of $B_1 \ominus B_2$ are pairwise disjoint. Let $C_2 = \{c_1, \dots, c_p\}$ be the constraints of B_2 , and b_i, b_j be respectively the i -th and the j -th value of $B_1 \ominus B_2$ according to (2), with $i, j \in 1..p$ and $i \neq j$. Then, b_i is constrained by $\neg c_i$. Assuming w.l.o.g. that $i < j$, then b_j is constrained by c_i , and $b_i \cap b_j = \emptyset$. We also have to prove that $B_1 = B_1 \setminus B_2 \cup B_2$, or equivalently, $\cup_i b_i = B_1 \setminus B_2$: let $x \in \cup_i b_i$ be an instantiation of B_1 . By definition of B_2 , there is at least a constraint $c_i \in C_2$ such that x does not satisfy. Let i_0 be the smallest such i , then $x \in b_{i_0}$. Thus, the whole of $B_1 \setminus B_2$ is covered by the boxes b_i .

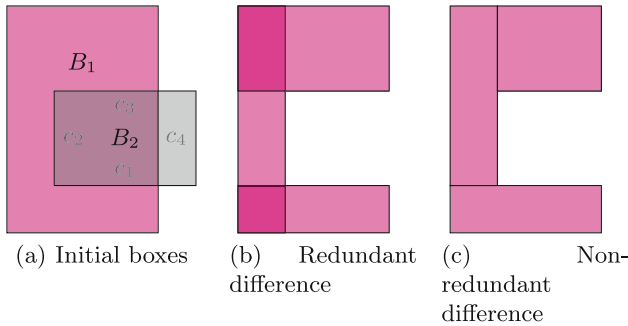


Fig. 3. Comparing naive and non-redundant difference operator: $B_1 \ominus B_2$. (Color figure online)

Figure 3(c) shows the result of the non-redundant difference operator and shows that there are no overlapping darker zones (shown in a darker shade). Here, $B_1 \setminus B_2$ is now partitioned into three elements, one per constraint of C_2 (once again ignoring c_4 which leads to an empty box).

3.3 New Solving Step

Computing $\tilde{S} = \theta_C(d_1, \dots, d_n) \cap \rho_C(d_1, \dots, d_n)$ by employing both propagation and elimination reduces the search space, because it allows the solver to quickly identify parts of the solutions. In fact, when the propagation of ρ_C is done, we propose an elimination step θ_C before splitting. Rather than performing arbitrary splits anywhere on a box, the elimination identifies parts of the box containing only solutions, and allows the solver to perform splits on the part of

Algorithm 2. Elimination function

```

1: function ELIMINATION( $e, \mathcal{C}$ ) ▷  $e$ : box,  $\mathcal{C}$ : constraints
2:    $e_{non-cons} \leftarrow \mathbf{complement}(e, \mathcal{C})$ 
3:    $e_{cons} \leftarrow e \ominus e_{non-cons}$ 
4:    $S \leftarrow \emptyset$ 
5:   for  $e_i \in e_{cons}$  do
6:      $S \leftarrow S \cup e_i$ 
   return  $(S, \oplus(e \cap e_{non-cons}))$ 

```

the search space that can not be discriminated as containing only solutions, nor as containing no solution. More precisely, elimination makes the split happen exactly at the frontier of the constraint.

Algorithm 2 gives the pseudo-code associated with the new elimination step. This algorithm processes elements that do not satisfy at least one constraint. The function `complement` computes $e_{non-cons}$, an over-approximation of the inconsistencies. Then, the difference operator is used to find the boxes containing only solutions. Finally, solving continues in the indeterminate search space $e \cap e_{non-cons}$ (instead of e).

Figure 1(b) shows the results obtained with our propagation/elimination/split loop on the CSP given previously, and gives for the same precision, much more satisfactory results: we require less elements to cover more space and in a comparable amount of time, showing that this technique deduces more relevant frontier than using a simple propagation/split loop.

In the following section, we analyze the performance of our solving method.

4 Experiments

We have implemented our technique for boxes in the open-source solver AbSolute¹. This solver is based on the method presented in [12], where we integrated our elimination step. We rely on the abstract domain representation in AbSolute, which is based on constraints, to efficiently implement the constraint negation necessary for the elimination step. The unified constraint representation makes it possible to have a lightweight and generic difference operator.

4.1 Protocol

We tested our method on problems with continuous variables from the MinLPlib and the Coconut² benchmarks. For minimization problems, we first transform them into satisfaction problems, which can be handled by the solver. This transformation consists in adding an objective variable to the problem that will act

¹ <https://github.com/mpelleau/AbSolute>.

² All informations about the problems can be found at <http://www.gamsworld.org/minlp/minplib/minlpstat.htm> and <http://www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/Benchmark.html>.

as the value to minimize. Default bounds for unconstrained variables are set to -10^7 for the lower bound and 10^7 for the upper bound as our method requires the domains of the variables to be bounded. All of the runs are made with a time limit set to 300s and no memory limit. Precision was fixed to 10^{-3} (i.e., the size limit where exploration stops), and branching depth was limited by 50. Note that limiting depth does not break soundness nor completeness as our algorithm can be tailored to be produce either a complete or a sound output after the same run: its output can be splitted into two sets: the boxes that contain only solutions and the undetermined ones. At any point of the resolution, the union of these two sets yield a complete solution, while taking into account only the first gives a sound solution. Thus, stopping the search at a given depth, makes the resolution faster, yet less precise, but does not break neither soundness nor completeness. The solver was run on a Dell server with two 12-core Intel Xeon E5-2650 CPU at 2.20 GHz, although only one core was used, and 128 GB RAM.

We have tested the solving with the elimination step against the default solving method of the AbSolute solver over all of the problems that the solver’s functionalities (types, constraint, arithmetic functions) are able to cover, that is 197 problems.

4.2 Description

Figure 4 summarizes the results obtained with our method compared to the classic solving. Figure 4(a) compares the ratio δ of inner volume of the cover. It corresponds to $V_i/(V_i + V_e)$ where V_i and V_e are respectively the inner and outer volume. This ratio is a quality measure of the solving method: the closer

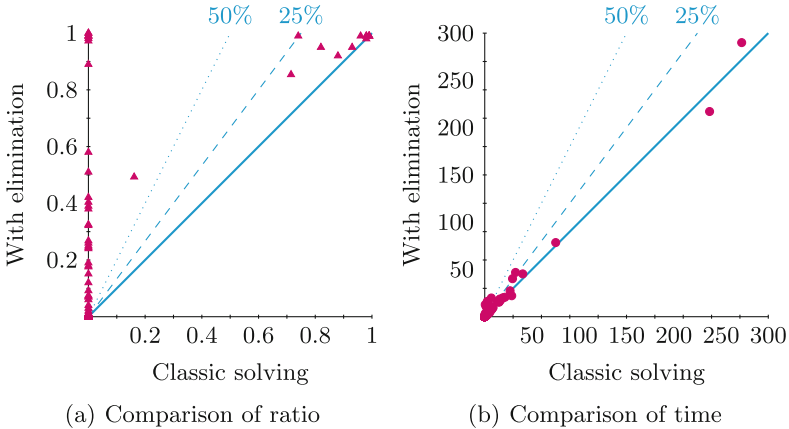


Fig. 4. Comparison between the classic solving method and our method. On the left, comparison of the ratio, a mark above the bisector (in plain blue) means that our method is better than the classic solving. On the right, comparison of the computation time, a mark above the bisector (in plain blue) means that our method is slower than the classic solving. (Color figure online)

Table 1. Comparing solving with and without elimination step

problem	$ \mathcal{X} , \mathcal{C} $	with elimination				without elimination			
		#I	#E	δ	t	#I	#E	δ	t
Coconut problems									
abs1	1,2	2047	3072	0.99	0.04	4092	4096	0.99	0.06
aljazzaf	2,3	2309	19405	0.58	0.89	0	14319	0	0.54
allinitu	1,5	318	5066	0.07	3.26	0	5066	0	2.50
b	4,4	2	88	0.39	0.07	0	88	0.00	0.07
booth	1,2	90	45	0.12	0.11	0	45	0	0.13
bqp	1,1	4	1	0.99	0.01	8	1	0.99	0.01
chi	1,2	1.88e6	3.48e6	0.99	45.30	3.08e6	3.63e6	0.99	40.20
ex1411	2,5	1.78e6	2.59e6	0.98	217.08	1.95e6	3.74e6	0.98	237.89
ex1413	4,3	4884	34893	0.25	0.52	0	32698	0	0.78
ex_newton	2,5	638	950	0.95	0.45	729	892	0.93	0.57
griewank	1,2	19972	31868	0.99	1.44	29645	35105	0.98	2.30
h76	3,4	24	174	0.04	0.05	0	82	0	0.05
hs23	6,2	825	2132	0.99	0.43	1315	1801	0.98	0.58
kear11	8,8	0	844	0	0.05	0	844	0	0.05
ladders	13,7	4	93	0.01	0.90	0	215	0.00	1.07
mickey	2,5	4315	12709	0.99	2.40	8372	9858	0.99	2.73
nonlin1	2,3	1550	1978	0.95	0.49	2059	1772	0.82	0.69
nonlin2	3,2	4238	10560	0.92	0.39	8643	10692	0.88	0.42
zy2	3,3	6260	28147	0.99	1.00	13179	22499	0.74	0.85
MinLP Lib problems									
csched1a	23,29	0	8192	0	6.44	0	8192	0	4.85
deb10	130,183	0	0	0	0.01	0	0	0	0.01
dosemin2d	119,166	0	0	0	0.181	0	0	0	0.177
ex1222	4,4	8	60927	0.01	1.39	0	61787	0	0.97
ex1223a	10,8	746	27097	0.01	20.60	0	48283	0	21.40
ex1223b	10,8	820	44084	0.01	40.22	0	500510	0	29.41
gbd	5,5	576	31829	0.19	1.27	0	22927	0	0.93
prob03	2,3	0	5.81e6	0	10.81	0	5.81e6	0	6.77
qapw	256,451	0	0	0	1.20	0	0	0	1.23
st_e13	4,3	378	3102	0.02	0.05	0	18	0	0.50
st_miqp2	4,5	1352	38104	0.38	2.29	0	4564	0	0.31
st_miqp3	2,3	27	1117	0.24	0.03	0	1051	0	0.02
st_miqp5	14,8	187	2080	0.01	4.71	0	6324	0	2.38
st_test1	2,6	1559	2.44e6	0.03	18.70	0	2.31e6	0	15.97
st_test5	12,11	22	29520	0.01	7.82	0	11167	0.00	7.55
synthes1	7,7	97	33747	0.01	4.18	0	1285	0	5.16
tls2	25,38	0	18030	0	27.46	0	18030	0	26.81
windfac	14,15	0	19561	0	6.66	0	19561	0	6.51

this ratio is to one, the bigger is the part of the coverage that will only contain solutions. In this figure, a mark above the bisector means that our method is better than the classic one. We can see that on most of the instances, our method finds a coverage with a much smaller indeterminate space.

Figure 4(b) compares the computation time of our method to the classic one. In this figure, each mark above the bisector means that our method is slower than the classic one. As can be seen on this figure, our method is slightly slower, the elimination performing additional computation during one iteration. However, solving all the 197 problems took 1157 s with our method against 1032 s with the classic solving method.

For reasons of space, Table 1 highlights only some of the results representative of the behavior of our method. Those are described with respect to solving times (in seconds), cardinality of the partition and volume covered. We performed experiments on the whole benchmarks (Coconut and MinLPLib), but we do not show here the problems which time out for both methods.

The first two columns provide information about the problem: name, number of variables $|\mathcal{X}|$, and number of constraints $|\mathcal{C}|$. The rest of the table provides information on each solving methods: the number of inner (columns $\#I$) and outer (columns $\#E$) boxes.

4.3 Analysis

These runs highlight one very crucial feature of our method: it is able to quickly find boxes that contain only solutions of problems where the default solving method fails to do so (problems *aljazzaf*, *allintu*, *ex1222*, *gbd*, ...): on the whole benchmark, for almost 30% of the problems (58 out of 197), solving with the elimination step exhibited at least one solution while the default solving method did not succeed to do so. This comes with no time less in average: on the whole benchmark, solving with elimination was slightly slower than without (1157 s against 1032 s). In fact, 39% of the problems (39 out of 197) were solved faster with the elimination than without (problems *ex1411*, *mickey*, *ex1223a*, *synthes1...*). This illustrates the fact that results of the solver are more precise: elimination avoids unnecessary splits, better identifies the constraints' frontiers, and compute within the same process inner and outer approximations for no (or little) overhead. A deeper analysis of the results shows that the default solving method spends time splitting variables with large ranges, while elimination focuses on the shape of the constraints to locate areas than can be directly removed from the search space and added to the solution set.

Another conclusion of the analysis of this benchmark is about the solution coverage. The experiments show that the coverage of the solution space is significantly more accurate with the elimination step. On all of the runs, our method always finds a greater or equal inner volume than the one found by the default method. Moreover, it also reduces the number of elements involved in the partition in the same time, which means that the inner approximation is achieved with fewer, bigger elements. This is shown by examples *chi* and *mickey* where both methods achieve a 0.99 ratio of inner volume, only with elimination, we need half the elements required by the default solving method to do so. On the whole benchmark, on average, we need 40 times fewer elements to cover the same inner volume with elimination. This property may become very handy as it allows a better re-usability of the results since we need to treat fewer elements to cover

the solution space. The δ columns indicates the part of the returned elements that corresponds to an inner approximation, i.e. contains only solutions. This ratio is always greater with the elimination step. On the whole benchmark, the average ratio is 0.49 of inner volume for the elimination while it is 0.27 without. This confirms that the elimination step allows the solving process to target more efficiently the parts of the search space that contain only solutions.

These good results confirm the intuition that cutting an element according to the constraints it does not satisfy can be more interesting than cutting it arbitrarily regardless of the constraints. Since solvers are often used as a pre-computation for other programs, reducing the size of their output (i.e., reducing the number of boxes required to represent a solution at a given precision) can be an important feature. Also, note that, by quickly identifying solutions and removing them from the search space, the elimination step makes it possible to carry out fewer propagation and exploration steps.

5 Conclusion

In classic continuous constraint solvers, propagation is used to remove from the search space values that can not be solutions. We presented in this paper a new method to, symmetrically, eliminate from the search space values that can only be solutions. We have incorporated the elimination mechanism to improve the results in terms of a qualitative and quantitative criterion, also without a too large time overhead. This technique, which delays a splitting heuristic that can be inaccurate, makes it possible to take better advantage of the constraints of a problem, by reusing and adapting the same tools as propagation, combined with a difference operator we have introduced. Finally, it should be emphasized that, although it is implemented in a specific solver using abstract domains, this technique can perfectly be integrated into a more classic solver and combined with any type of propagator.

We believe that this resolution technique can be useful in many cases. For problems or zones of non-consistent instantiations forming “holes” in the solution space, or more generally, when it is non-convex, it can avoid several cutting steps by directly targeting the most relevant boundaries. This property may be particularly interesting in the context of inner-approximation applications, as shown by the experiments, or counter-example exhibition (feasibility proving) when it comes to find at least one solution as our method outperforms the default solving method in that competence.

Further research includes the development of elimination beyond boxes, for instance on polyhedra which can also be defined as a conjunction of constraints, making it possible to add a difference operator. It would also be interesting to measure the performance of this technique with other consistency and splitting heuristics.

References

1. Batnini, H., Michel, C., Rueher, M.: Mind the gaps: a new splitting strategy for consistency techniques. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 77–91. Springer, Heidelberg (2005). https://doi.org/10.1007/11564751_9
2. Benhamou, F.: Heterogeneous constraint solvings. In: Proceedings of the 5th International Conference on Algebraic and Logic Programming, pp. 62–76 (1996)
3. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.-F.: Revisiting hull and box consistency. In: Proceedings of the 16th International Conference on Logic Programming, pp. 230–244 (1999)
4. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: Proceedings of the 16th European Conference on Artificial Intelligence, (ECAI 2004), pp. 146–150. IOS Press (2004)
5. Bussieck, M.R., Drud, A.S., Meeraus, A.: Minplib - a collection of test models for mixed-integer nonlinear programming. *INFORMS J. Comput.* **15**(1), 114–119 (2003)
6. Chabert, G., Jaulin, L.: Contractor programming. *Artif. Intell.* **173**, 1079–1100 (2009)
7. Collavizza, H., Delobel, F., Rueher, M.: Extending consistent domains of numeric CSP. In: Proceedings of the 16th International Joint Conference on Artificial Intelligence, pp. 406–413 (1999)
8. Hansen, E.: Global optimization using interval analysis. Marcel Dekker, New York (1992)
9. Haralick, R.M., G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In: Proceedings of the 6th International Joint Conference on Artificial intelligence (IJCAI 1979), pp. 356–364. Morgan Kaufmann Publishers Inc. (1979)
10. Jaulin, L., Walter, E.: Set inversion via interval analysis for nonlinear bounded-error estimation. *Automatica* **29**(4), 1053–1064 (1993)
11. Moore, R.E.: Interval Analysis. Prentice-Hall, Englewood Cliffs (1966)
12. Pelleau, M., Miné, A., Truchet, C., Benhamou, F.: A Constraint Solver Based on Abstract Domains. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 434–454. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_26
13. Ratz, D.: Box-splitting strategies for the interval Gauss-Seidel step in a global optimization method. *Computing* **53**, 337–354 (1994)
14. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Elsevier Science Inc., New York (2006)
15. Schulte, C., Stuckey, P.J.: Efficient constraint propagation engines. *Trans. Program. Lang. Syst.* **31**(1), 1–43 (2008)
16. Shcherbina, O., Neumaier, A., Sam-Haroud, D., Vu, X.-H., Nguyen, T.-V.: Benchmarking global optimization and constraint satisfaction codes. In: Bliet, C., Jeremann, C., Neumaier, A. (eds.) COCOS 2002. LNCS, vol. 2861, pp. 211–222. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39901-8_16



The Effect of Structural Measures and Merges on SAT Solver Performance

Edward Zulkoski¹(✉), Ruben Martins², Christoph M. Wintersteiger³,
Jia Hui Liang¹, Krzysztof Czarnecki¹, and Vijay Ganesh¹

¹ University of Waterloo, Waterloo, ON, Canada
ezulkosk@uwaterloo.ca

² Carnegie Mellon University, Pittsburgh, PA, USA

³ Microsoft Research, Cambridge, UK

Abstract. Over the years complexity theorists have proposed many structural parameters to explain the surprising efficiency of conflict-driven clause-learning (CDCL) SAT solvers on a wide variety of large industrial Boolean instances. While some of these parameters have been studied empirically, until now there has not been a unified comparative study of their explanatory power on a comprehensive benchmark. We correct this state of affairs by conducting a large-scale empirical evaluation of CDCL SAT solver performance on nearly 7000 industrial and crafted formulas against several structural parameters such as treewidth, community structure parameters, and a measure of the number of “mergeable” pairs of input clauses. We first show that while most of these parameters correlate well with certain sub-categories of formulas, only the merge-based parameters seem to correlate well across most classes of industrial instances. To further methodically test this connection, we perform a scaling study of mergeability of randomly-generated formulas and CDCL solver running time. We show that as the number of resolvable merge pairs are scaled up for randomly-generated instances while keeping most properties invariant, unsatisfiable instances show a very strong negative correlation with solver runtime. We further show that there is strong negative correlation between the size of learnt clauses and the number of merges as the number of merge pairs are scaled up. A take-away of our work is that mergeability might be a powerful complexity theoretic parameter with which to explain the unusual efficiency of CDCL SAT solvers.

Keywords: SAT solving · Structural measures
Merge resolution · CDCL

1 Introduction

Modern conflict-driven clause-learning (CDCL) satisfiability (SAT) solvers routinely solve real-world Boolean instances with millions of variables and clauses, despite the Boolean satisfiability problem being NP-complete and widely

regarded as intractable in general. This has perplexed both theoreticians and solver developers alike over the last two decades. A commonly proposed explanation is that these solvers somehow exploit the underlying structure inherent in industrial instances. Previous work has attempted to identify a variety of structural parameters, such as backdoors [7, 19], community structure modularity [5, 15], and treewidth [13]. Additionally, researchers have undertaken limited studies to correlate the size/quality measures of these parameters with CDCL SAT solver runtime. For example, Newsham et al. [15] showed that there is moderate correlation between the runtime of CDCL SAT solvers and modularity of community structure of industrial instances.

Although many studies of these parameters have been performed in isolation, a comprehensive comparison has not been performed between them. A primary reason for this is that most of these parameters are difficult to compute – often NP-hard – and many take longer to compute than solving the original formula. Hence, such parameters have often been evaluated on incomparable benchmark sets, making a proper comparison between them difficult.

This state of affairs leads to the following research question: *do previously considered structural measures correlate with solving time “in the large,” i.e., do results generalize to diverse sets of instances?* We address this issue by designing a single comprehensive study of all proposed structural parameters (including one of our own, namely, mergeable resolvable pairs) over a large set of instances obtained from previous SAT competitions, specifically from the application, crafted, and agile tracks [1]. Application instances are derived from a wide variety of sources and can be considered as a sampling of the types of SAT instances from applications as varied as verification, testing, program analysis, and security. Crafted instances mostly contain encodings of combinatorial/mathematical properties, such as the pigeon-hole principle or pebbling formulas. While many of these instances are much smaller than industrial instances, they are often very hard for CDCL solvers. The agile track evaluates solvers on bit-blasted quantifier-free bit-vector instances generated from the whitebox fuzz tester SAGE [9]. In total, we consider approximately 1200 application instances, 800 crafted instances, and 5000 agile instances. While correlation results appears to be weak in general, combinations of these features can lead to slightly stronger regression results.

Since SAT competitions are comprised of a diverse set of instances, we then performed a more fine-grained analysis by considering each individual sub-category of the application track. Our results indicate that many of these measures correlate strongly with certain sub-categories, but not others.

Within this study, we introduce a measure called “mergeability,” and show that it correlates moderately or strongly with many sub-categories. Mergeability quantifies how many pairs of input clauses are mergeable. Two clauses are mergeable if they resolve and share a common literal. Merge resolutions are particularly important, as they allow the resolvent clause to be smaller than the two clauses being resolved. In order to further isolate the effects of mergeability on solving time, we conduct controlled scaling studies over sets of random instances

while scaling the mergeability of the formula. We describe an algorithm which takes a formula, and produces a series of increasingly more mergeable formulas, while retaining many properties of the original instance. We experiment over a set of randomly-generated industrial-like instances, as well as uniform 3SAT instances, and show that as the number of merges increase, the solving time tends to decrease for unsatisfiable instances.

1.1 Contributions

We make the following contributions in this paper:

1. **Comprehensive Study of Complexity-theoretic Structural Parameters and SAT Solver Performance.** We performed a comprehensive study complexity-theoretic structural parameters over 7000 SAT competition instances, and show that while many of the considered measures correlate well in the small, i.e., over sub-categories of application instances, the correlations are not strong when considering large sets of diverse instances. (See Sect. 4.)
2. **The Mergeability Parameter and Method to Generate Scaling Instances.** We introduce the mergeability parameter (motivated by idea that merges can result in smaller resolvents [2]), and show that it correlates well with many categories of instances (See Sect. 4.1). Further, in order to isolate the effects of mergeability and perform scaling studies, we describe a generator to produce instances scaled with the mergeability parameter (See Sect. 5).
3. **Scaling Study of Mergeability and SAT Solver Performance.** In Sect. 6, we describe scaling studies over randomly-generated instances, and show that as the number of merges increase, the SAT solver’s run time tends to decrease for unsatisfiable instances. We further show that the solver tends to on average produce shorter learned clauses as we scale up the number of mergeable clauses in randomly-generated instances.

2 Background

We assume basic familiarity with the satisfiability problem, CDCL solvers and the standard notation used by solver developers and complexity theorists. For an overview we refer to [6]. We assume that Boolean formulas are given in conjunctive normal form (CNF). We refer to treewidth [18] and community structure [5] as graph parameters. Unless otherwise stated, all graphs parameters are computed over the *variable incidence graph (VIG)* of a CNF formula F [5]. There exists a vertex for every variable in F , and edges between vertices if their corresponding variables appear in clauses together (weighted according to clause size). The *clause-variable incidence graph (CVIG)* is a bipartite graph between variables and clauses, such that variable node connects to a clause node exactly when the variable occurs in the clause. Intuitively, the community structure of a graph is a partition of the vertices into communities such that there are more

intra-community edges than inter-community edges. The *modularity* or Q value denotes how easily separable the communities of the graph are. The Q value ranges from $[0, 1]$, where values near 1 means the communities are highly separable and the graph intuitively has a *better* community structure. *Treewidth* intuitively measures how much a graph resembles a tree. Actual trees have treewidth 1. Further details can be found in [18]. The *backbone* of a SAT instance is the set of variables such that all models of the instance contain the same polarity of the variable [14]. The *popularity* of a variable measures how often it occurs in the formula, normalized by the total number of variables occurrences. In [4], it was shown that industrial formula variable occurrences tend to follow a power-law distribution. We denote the exponent in the computed distribution as α_v . Intuitively, the *fractal dimension* of a graph measures self-similarity, in the sense that if nearby nodes are collapsed into a single node according to some distance metric, then the residual graph would have the same structure as the original graph [3]. This property is measured by considering tilings of the graph that cover all vertices, and then showing that the number of necessary tiles decreases polynomially as the tile-size increases. More details can be found in [3]. For a given graph G , we denote the dimension of the graph as D_G .

Two clauses c_1 and c_2 are *resolvable* if there exists a single variable v such that $v \in c_1$ and $\neg v \in c_2$. We say that the clauses are *mergeable* if they are resolvable, and there also exists some literal l such that $l \in c_1$ and $l \in c_2$. We consider two measures of formulas, namely, *mergeability* and *resolvability*, that quantify basic semantic properties of the input. Let C be the number of clauses in a formula F . Let r be the number of resolvable pairs of clauses in F , and m be the number of mergeable clause pairs, such that if a pair merges n times (i.e., it has n overlapping literals), then m is incremented n times. Then the *resolvability* of the input formula is $R = r/C^2$ and the *mergeability* $M = m/C^2$. Both measures can be computed in $\mathcal{O}(C^2 \cdot L^2)$, where L is the length of the longest clause.

3 Related Work

Mateescu computed lower and upper bounds on the treewidth of large application formulas [13], and showed that treewidth did not correlate well over large sets of application instances, however Pohl et al. showed strong correlations between treewidth and solving time for feature models [17]. Ansótegui et al. introduced community structure abstractions of SAT formulas, and demonstrated that industrial instances tend to have much better structure than other classes, such as random [5]. It has also been shown that community-based features are useful for classifying industrial instances into subclasses [10]. Community-based parameters have also recently been shown to be one of the best predictors for SAT solving performance [15]. A difference between [15] and our work is that they group all application, crafted, and random instances into a single study, while we evaluate them separately. We also consider several more parameters and more instances. Kilby et al. demonstrated a moderate positive

Table 1. Adjusted R^2 values for the given features, compared to log of MapleCOMSPS’ solving time. The number in parentheses indicates the number of instances that were considered in each case.

Feature Set	Application	Crafted	Random	Agile
$V \oplus C \oplus C/V$	0.03 (1143)	0.04 (753)	0.08 (126)	0.85 (4968)
$V \oplus C \oplus CM \oplus Q \oplus Q/CM$	0.07 (889)	0.26 (613)	0.28 (126)	0.89 (4968)
$V \oplus C \oplus Bones \oplus Bones/V$	0.17 (193)	0.40 (154)	0.04 (59)	0.43 (208)
$V \oplus C \oplus TW \oplus TW/V$	0.05 (1087)	0.07 (753)	0.28 (126)	0.91 (4968)
$V \oplus C \oplus D_V \oplus D_C \oplus \alpha_V$	0.06 (1143)	0.19 (753)	0.25 (126)	0.88 (4901)
$V \oplus C \oplus M \oplus R \oplus M/R$	0.20 (823)	0.25 (604)	0.19 (126)	0.91 (4870)
$C/V \oplus Q \oplus TW/V \oplus M/R \oplus R \oplus D_V$	0.31 (823)	0.43 (604)	0.15 (126)	0.95 (4870)
$Q \oplus Q/CM \oplus TW/V \oplus M/R \oplus D_V \oplus D_C$	0.30 (823)	0.56 (604)	0.39 (126)	0.95 (4870)
$V \oplus CM \oplus Q \oplus R \oplus D_V \oplus \alpha_V$	0.22 (823)	0.42 (604)	0.66 (126)	0.93 (4870)
$V \oplus C/V \oplus Q \oplus TW/V \oplus M \oplus D_C$	0.23 (823)	0.47 (604)	0.20 (126)	0.96 (4870)

correlations between backbone size and solving time on uniform random 3SAT instances [11]. Andrews introduced resolution with merging, and demonstrated a sound and complete proof system that exploits merges [2]. A clause-learning scheme based on merges during conflict was considered in [16].

4 Relating SAT Measures to CDCL Performance

Our first set of experiments investigate the relationship between structural parameters and CDCL solving time. Specifically, we pose the following question: *Do parameter values correlate with solving time? In particular, can we build significantly stronger regression models by incorporating combinations of these features?* We focus on community structure, popularity, fractal dimension, backbones, treewidth, and merge-resolvability of input.

We use off the shelf tools to compute community structure and modularity [5], and treewidth [13]. Due to the difficulty of exactly computing these parameters the algorithms used in previous work (and our experiments) do not find optimal solutions, e.g., the output may be an upper-bound on treewidth, however backbones were computed exactly. We use *MapleCOMSPS*, the 2016 SAT competition main track winner as our reference solver for runtimes.

We include all instances from the application and crafted tracks of the SAT competitions from 2009 to 2014, as well as the 2016 agile track. We also included a small set of random instances from the 2007 and 2009 SAT competitions as a baseline. As the random instances from recent SAT competitions are too difficult for CDCL solvers, we include a set of instances from older competitions. All instances were simplified using MiniSat’s preprocessor before computing the parameters. The preprocessing time was not included in solving time.

Experiments were run on an Azure cluster, where each node contained two 3.1 GHz processors and 14 GB of RAM. Each experiment was limited to 6 GB. For the application, crafted, and random instances, we allotted 5000 s for MapleCOMSPS solving (the same as used in the SAT competition) and 2 h for metric

computation. For the agile instances, we allowed 60s for MapleCOMSPS solving and 300s for metric computation. Due to the difficulty of computing these parameters, we do not obtain values for all instances due to time or memory limits being exceeded.

We construct linear regression models from subsets of features related to these parameters, akin to [15]. We consider the following “base” features: number of variables (V), number of clauses (C), number of communities (CM), modularity (Q), treewidth upper-bound (TW), fractal-dimensions of the VIG and CVIG (D_V and D_C), variable popularity (α_v), resolvability (R), and mergeability (M). We also include the ratio feature Q/CM , as used in [15]. All features are normalized to have mean 0 and standard deviation 1. For a given subset of these features under consideration, we use the “ \oplus ” symbol to indicate that our regression model contains these base features, as well as all higher-order combinations of them (combined by multiplication), plus an “intercept” feature. Our dependent variable is the log of runtime of the MapleCOMSPS solver [12].

In Table 1, we first consider sets of features defined with respect to a single parameter type, e.g., only community structure based features, along with V and C as baseline features. Each cell reports the adjusted R^2 value of the regression, as well as the number of instances considered in each case (which corresponds to the number of instances for which we have data for each feature in the regression). It is important to note that since different subsets of SAT formulas are used for each regression (since our dataset is incomplete), we should not directly compare the cells in the top section of the table. Nonetheless, the results do give some indication if each parameter relates to solving time.

In order to show that combinations of these features can produce somewhat stronger regression models, in the bottom half of Table 1, we consider all instances for which we have data for every feature, not including backbones which are only defined over satisfiable instances. We considered all subsets of base features of size 6, and report the best model for each benchmark, according to adjusted R^2 . This results in notably stronger correlations than with any of the feature sets defined over a single parameter (i.e. the top half of the table). We also note that R^2 values results tend to be higher for the agile instances, as compared to application and crafted instances. This is somewhat expected, as the set of instances are all derived from the SAGE whitebox fuzzer [9], as compared to our other benchmarks which come from a heterogeneous set of sources.

Not too surprisingly, no single parameter is significantly predictive of SAT solver performance across the wide variety of instances considered in this work. While combinations of these parameters do produce notable improvements with respect to R^2 values, there is still much room for improvement, especially when considering the application and crafted instances.

4.1 Relating Measures to Performance by Sub-category

From the above correlation results, it is clear that these features do not fully explain CDCL performance “in the large.” Nonetheless, it is worth considering how each measure relates to solving performance on a more fine-grained scale,

Table 2. Spearman correlation results between solving time and measures, for each application sub-category. The number of satisfiable and total instances is given with the category name. Blue cells with a single ‘+’ indicate moderate positive correlations ($0.4 \leq r < 0.6$); two ‘+’ symbols indicate $0.6 \leq r < 0.8$; three ‘+’ symbols indicate $r > 0.8$. Red cells indicate negative correlations using the same system (e.g., two ‘-’ symbols indicate $-0.6 \geq r > -0.8$). Blank cells have lower correlations of $|v| < 0.4$.

Category (#SAT/Total)	C/V	Q	Bones/V	TW/V	M	M/R
2d-strip-packing (5/11)	++	++			----	--
argumentation (15/15)	+++	----		+++	+++	+++
bio (19/30)			+++			-
crypto-aes (17/22)						
crypto-des (15/15)				--	++	++
crypto-gos (1/34)		+			+	++
crypto-md5 (10/17)	+					++
crypto-sha (47/47)						
crypto-vmc (20/20)	++		-	+	--	--
diagnosis (38/79)					-	
hardware-bmc (0/22)					-	-
hardware-bmc-ibm (8/12)		+			+	
hardware-cec (0/44)						
hardware-manolios (0/22)	+			--	-	--
hardware-velev (0/5)		+		-		-
planning (3/11)					-	-
scheduling (29/50)						
scheduling-pesp (2/24)					-	-
software-bit-verif (8/43)					-	
termination (26/38)						

by considering the sub-categories of the SAT competition application instances. Most application track instances can be classified into one of 20 sub-categories, as listed in the first column of Table 2. These categories are loosely encompassed by various types of cryptographic instances, software verification, and hardware verification problems. Again, we consider all instances from the 2009–2014 competitions for which we could accurately classify the instance into a category, totalling 780 instances.

Table 2 summarizes Spearman correlation results between various features and solving time. For ease of interpretation, instead of raw correlation values, we label strong correlations as blue with ‘+’ symbols, with the strongest correlations denoted by ‘+++’, as described in the table’s description.

For certain categories, many parameters seem to strongly relate to solving time (e.g., argumentation), whereas for other categories solving time does not relate well to any (e.g., scheduling). Interestingly, modularity often positively correlates with solving time, indicating instances with intuitively better community structure actually require longer solving time. Among our considered measures, mergeability and the ratio mergeable clauses to resolvable clauses (M/R)

seem to correlate strongly with many sub-categories, with mergeability having at least a moderate correlation in 12 sub-categories. However, depending on the sub-category, the correlation may be strongly positive or strongly negative. This suggests that models beyond the linear ones considered previously may offer better correlation results. Further, although not depicted, similar results were found for Pearson correlations.

5 Generating Mergeable Formulas

From Table 2, mergeability tended to correlate with solving time for many sub-categories of application instances. In the remaining sections, we test the effect of mergeability by constructing scaling instances with varying numbers of mergeable clause pairs.

We propose a greedy approach to increase the number of merges, as described in Algorithm 1. Our approach works in the following main steps. We take as input an arbitrary formula in CNF. Then, up until fixed-point, we seek pairs

Algorithm 1. Greedy Approach to Increase Mergeable Input Clauses

```

1: input: CNF Formula  $F$ 
2: output: Modified CNF Formula  $F'$ 
3: set  $lockedLits$ ,  $lockedClauses$ ,  $flipPairs$ 
4: bool  $formulaHasChanged \leftarrow true$ 
5: while  $formulaHasChanged$  do
6:    $lockedLits$ ,  $lockedClauses$ ,  $flipPairs \leftarrow \{\}, \{\}, \{\}$ 
7:    $formulaHasChanged \leftarrow false$ 
8:   for every pair of clauses  $c_i, c_j$  do
9:     if  $(|c_i \cap \{\neg l \mid l \in c_j\}| \geq 1)$  then
10:      for every merge literal  $l \in c_i \cap c_j$  do ▷ lock literals that merge
11:         $lockedLits \leftarrow lockedLits \cup \{(c_i, l), (c_j, l)\}$ 
12:      if  $(|c_i \cap \{\neg l_1 \mid l_1 \in c_j\}| == 1) \wedge (\exists l_2 : l_2 \in c_i \cap c_j)$  then
13:         $lockedLits \leftarrow lockedLits \cup \{(c_i, l_1), (c_j, \neg l_1)\}$  ▷ lock literals that
        resolve mergeable clauses
14:      else if  $(|c_i \cap \{\neg l \mid l \in c_j\}| > 1)$  then
15:        for every literal pair  $l \in c_i$  and  $\neg l \in c_j$  do
16:           $flipPairs \leftarrow flipPairs \cup \{((c_i, l), (c_j, \neg l))\}$ 
17:      for  $((c_i, l), (c_j, \neg l)) \in flipPairs$  do
18:        if  $c_i, c_j \notin lockedClauses \wedge (c_i, l), (c_j, l) \notin lockedLits$  then
19:          if  $\exists c_k \notin lockedClauses \wedge \neg l \in c_k \wedge (c_k, \neg l) \notin lockedLits \wedge c_k \neq c_j$  then
20:             $flip(c_i, l)$  ▷ Changes the polarity of  $l$  in  $c_i$ 
21:             $flip(c_k, \neg l)$ 
22:             $lockedClauses \leftarrow lockedClauses \cup \{c_i, c_j, c_k\}$ 
23:             $formulaHasChanged \leftarrow true$ 
24:          else if  $\exists c_k \notin lockedClauses \wedge l \in c_k \wedge (c_k, l) \notin lockedLits \wedge c_k \neq c_i$  then
25:            “Same as Lines 20-23, substituting  $c_j$  for  $c_i$ .”

```

of “tautologically resolvent” clauses, i.e., clauses that resolve on two or more variables. Consider the following example clauses:

$$(x \vee y) \wedge (\neg x \vee \neg y). \quad (1)$$

This pair of clauses resolve on both x and y , and is therefore not mergeable nor resolvable.¹ However, if we flip the polarity of any one of the four literals:

$$(x \vee y) \wedge (\neg x \vee y), \quad (2)$$

then the clauses both resolve (on x) and merge (on y). This process repeats until no more changes to the formula increase the number of overall merges. In order to ensure progress, an additional invariant ensures that if two clauses merge on the original instance, then they will also merge in the generated instance. While either of the clauses may be modified (by flipping other literals in the clauses to allow additional merges), a merge will still exist between them.

We represent literals as (clause_id, literal) pairs in order to distinguish different instances of the same literal. There are three main sets that ensure our invariants. The set *lockedLits* maintains all literals that are either merged by two resolvable clauses, or are the literals that are actually resolved in a merge. Suppose we have clauses $c_i = (a \vee b \vee c)$, $c_j = (\neg a \vee b \vee d)$. Since the clauses resolve on a single variable a and merge on b , Line 11 will add (c_i, b) and (c_j, b) to *lockedLits*, and Line 13 will add (c_i, a) and $(c_j, \neg a)$. This ensures that the algorithm will never flip the polarity of any of these literals, ensuring that the output formula retains these same merges. Suppose that two clauses, e.g., $c_i = (a \vee b \vee c)$, $c_j = (\neg a \vee b \vee \neg c)$ contain multiple literals upon which to resolve (a and c). Then on Lines 15–16, we add all resolving pairs of literals to *flipPairs* (e.g., $((c_i, a), (c_j, \neg a))$ and $((c_i, c), (c_j, \neg c))$). The intuition is that if any one of these literal’s polarities were flipped, then the two clauses would merge on two literals instead of one, while still being resolvable (e.g. $c_i = (a \vee b \vee \neg c)$, $c_j = (\neg a \vee b \vee \neg c)$).

In Lines 17–25, we iterate through all pairs of literals in *flipPairs*; in practice, we randomize the ordering to ensure we do not prefer flipping literals in earlier clauses. If the associated clauses and literals are not locked (Line 18), then we may try to flip one of the literals to increase the overall number of merges. In order to flip some literal, e.g., (c_i, l) to $(c_i, \neg l)$, there must exist some other unlocked literal $(c_k, \neg l)$ that we can flip, in order to ensure that the overall literal counts remain constant. If so, we flip the corresponding literals and lock the three involved clauses (Lines 20–23, and similarly on Line 25). This ensures that we do not end up flipping too many literals in the same clause, to the point where the clauses no longer resolve. This process repeats until fixpoint, clearing the three involved sets at the start of each iteration (Line 6).

5.1 Properties of the Generator

We designed our algorithm to retain as many properties of the original instance as possible while increasing mergeability. The following are known properties

¹ Note that in the resolution proof system, any resolvent of these clauses (e.g., $(x \vee y \vee \neg y)$) is effectively useless since it is tautologically true.

retained by our approach: the number of variables and clauses, all community structure properties based on the variable incidence graph (i.e. modularity), and popularity of variables and literals. These properties follow simply since we only ever flip the polarities of literals, and if we flip the polarity of a literal l in some clause, we must flip $\neg l$ in another clause. The following are known properties that are *not* retained: satisfiability, resolvability, and community structure properties if computed over the literal incidence graph. We discuss several other properties in the following results. Finally, we demonstrate that a small change to the algorithm can be used to *decrease* the mergeability of the formula instead.

Observation 1. *Algorithm 1 terminates.*

Proof. First, note that the only time that the formula is changed is in calls to $flip(\dots)$ on Lines 20–21, 25. We show that any time these sets of Lines are invoked, the size of $lockedLits$ must increase in the next iteration of the while-loop, and since the formula is finite, the algorithm will eventually terminate. Note that if $flip$ is never invoked, then the while-loop must terminate as $formulaHasChanged$ must be *false*.

We first show every element of $lockedLits$ from the previous while-loop iteration will still be in $lockedLits$ in the next iteration. There are two cases to consider. First, if two clauses resolve on a single literal and merge, then the resolving literal is locked (Line 13) and all merging literals are locked (Lines 10–11). Since these literals can then never be flipped, the same literals will be locked due to this clause pair in the next while-loop iteration. Further, flipping any of the unlocked literals in the clause pair will not reduce the number of merges (e.g., by creating a second pair of conflicting literals between the clauses, thus making the clause pair no longer resolvable). Second, if the clause pair has multiple conflicting literals (thus adding pairs to $flipPairs$), then only the literals that are merges are added to $lockedLits$ (unless added by a different pair of clauses). Since a clause can only be changed on one literal at each iteration of the while-loop (due to $lockedClauses$ being updated on Lines 22, 25), there will still exist at least one pair of conflicting literals to satisfy the if-statement on Line 9 in the next iteration.

Assume *w.l.o.g.* that we flip (c_i, l) and $(c_k, \neg l)$ on Lines 20–21. Then in the next iteration both c_i and c_j will contain the literal $\neg l$, and since they must still have a conflicting literal, $(c_i, \neg l)$ and $(c_j, \neg l)$ will be added to $lockedLits$. \square

Observation 2. *Each iteration of the while-loop in Algorithm 1 (Lines 6–25) cannot decrease the mergeability of the formula.*

Proof. If a clause pair already merges on (potentially several) literals and resolves, then all relevant literals will be locked and cannot be changed (through similar arguments as in Observation 1). Thus, the mergeability will not decrease. As literals get flipped, it will decrease the number of conflicting literals between the clause pair, and increase the number of overlapping literals. If the number of conflicting literals is ever reduced to one, the number of overall merges will increase by the number of merge literals. \square

Observation 3. *Algorithm 1 can be modified to decrease mergeability by changing how lockedLits and flipPairs are computed on Lines 9–16.*

If a pair of clauses has exactly 2 pairs of opposing literals (as in Eq. 1), then we lock all 4 literals. Essentially, if one of these literals were to be flipped, the clause would resolve and merge. We do not lock any other literals. As for *flipPairs* (which now constitute pairs of literals which we hope to flip to *reduce* mergeability), if a clause pair merges, we add all pairs of merging literals, as well as the pair of resolving literals to the data structure. As we will show in the following subsection, for many uniform 3SAT formulas, we can reduce the number of mergeable pairs to zero.

5.2 Mergeability of Random-kSat Instances

Before discussing our empirical results, we demonstrate properties of randomly generated kSAT instances. Specifically, we find the expected number of merges a random-kSAT instance will have, which depends on the clause size k , the number of variables n , and the number of clauses m . We assume that all clauses have exactly k variables, and each possible clause occurs with uniform probability. We use these expected values, in conjunctions with the empirically computed variance of mergeability of the distribution, to contextualize our empirical results.

Theorem 1. *Let F be a random-kSAT formula with n variables and m clauses. Then the expected number of merges over input clause pairs is:*

$$E(\text{merges}(F)) = \binom{m}{2} \sum_{i=1}^k \frac{i(i-1)}{2^i} \cdot \frac{\binom{k}{i} \binom{n-k}{k-i}}{\binom{n}{k}}. \quad (3)$$

6 The Effect of Merges on CDCL Performance

Here we consider the following question: *as one scales Boolean formulas to have more mergeable clause pairs (while keeping all other formula characteristics invariant), are the formulas solved faster by a CDCL solver.* Our experiments suggest that mergeability strongly negatively correlates with solving time when considering unsatisfiable instances. We test our hypothesis by taking pre-generated formulas (i.e. from an off-the-shelf random instance generator), and then modifying them to increase/decrease mergeability, while retaining as many properties of the original instance as possible, using our approach in Sect. 5.

6.1 Experiment

We first use the popularity-similarity random instance generator described in [8] to create our base formulas. The generator has several parameters related to typical properties of industrial/application SAT instances. The *temperature* T allows one to tune the *similarity* in the instance: intuitively, during generation,

Table 3. Correlations between mergeability and solving time for varying temperatures. Min (resp. Max) refer to the number of mergeable clause pairs in the formula with the least (resp. most) mergeable clause pairs in each formula series. Base is the number of mergeable pairs in the original instance not modified by our generator. Min Time and Max Time correspond to the times for the instance with the minimal (resp. maximal) number of mergeable pairs.

T	Spearman	Pearson	Min	Base	Max	Min time (s)	Max time (s)
T1.8	-0.9	-0.89	851.57	851.57	1155.86	1.23	0.6
T1.9	-0.92	-0.92	609	608.29	849.25	95.39	41.7
T2.0	-0.84	-0.89	332.6	363.89	508.9	8.53	5.89
T2.1	-0.92	-0.93	263	286.88	415.78	153.68	50.65
T2.2	-0.94	-0.93	95.4	224.9	326.7	6.88	2.63
T2.3	-0.97	-0.95	68.2	193.62	294.9	35.27	9.49
T2.4	-0.97	-0.94	70.3	183.22	261.7	126.07	18.31
T2.5	-0.98	-0.95	64.4	177.38	251.6	817.03	105.33
T5.0	-0.97	-0.98	42	99	137.4	12.98	4.94
T10.0	-0.97	-0.98	75.33	82	125	33.61	16.2
T100.0	-0.97	-0.95	69.43	82	119.14	94.43	35.24

Table 4. Results for traditional random kSAT instances, split by satisfiability.

Num Vars	Spearman	Pearson	Min	Base	Max	Min time (s)	Max time (s)
200 (unsat)	-0.92	-0.92	52.84	81.6	123.04	2.48	0.74
225 (unsat)	-0.91	-0.91	65.46	83.46	123.11	5.84	2.61
250 (unsat)	-0.91	-0.91	56.6	84.73	123.97	13.53	6.38
200 (sat)	0.26	0.27	2.33	80.42	82.85	0.29	0.46
225 (sat)	0.27	0.30	18.99	81.76	92.76	0.97	1.74
250 (sat)	0.37	0.33	0.68	81.33	89.76	2.52	4.80

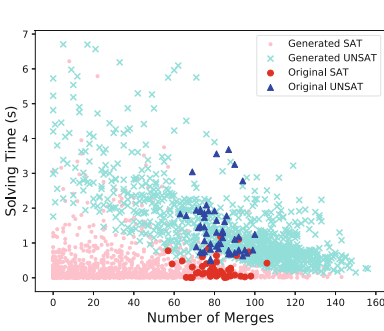
each literal is assigned a random number, and if the temperature is low, literals with small differences in their random numbers are more likely to appear in the same clause. At higher temperatures, the generated formulas appear more close to traditional random SAT instances. We consider different temperature values between 1.8 – 100, as listed in Table 3. We use the default *popularity* parameter $\beta = 1$, such that the variables are expected to occur according to a power-law distribution, as witnessed in industrial instances [4]. We restrict the formulas to be 3SAT instances. The clause popularity parameter β' was set to 0, and the clause/variable ratio is always 4.25, as in [8]. Since formulas with higher temperature are harder for CDCL solvers, formulas for $T < 2$ have 5000 variables, those with $2 \leq T < 2.2$ have 2000, those with $2.2 \leq T < 5$ have 1000, and those with $T \geq 5$ have 300. All other parameters are as default.

For each temperature value, we generate 10 base formulas for a total of 110 base formula. We then use these base formulas to generate a total of 1200 scaled formulas. Importantly, all considered formulas (both base formulas and those generated with our algorithm) are unsatisfiable. We then use our algorithm to increase or decrease the number of merges in each instance, creating a series of formulas associated with each base formula. We allot 1 h for each run of our tool, and record the modified formula every 10 flips of literals, up to a limit of 500 flips (both for decreasing and increasing merges). Although each flip may cause many new merges, in our experiments, each flip tends to introduce one or two merges. In total, we considered 110 base formulas, and 1200 total formulas. We repeated this experiment for uniform random 3SAT instances at the phase transition (clause/variable ratio of 4.26), generating 100 base instances for each number of variables $n \in \{200, 225, 250\}$ (allowing both satisfiable and unsatisfiable instances in this case), and generating a new formula for every 5 flips of literals, in total producing 7043 formulas.

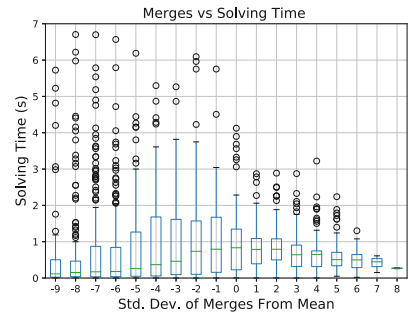
For each formula series (i.e., a base formula with its series of varied mergeability versions), we run MapleSAT as default with a 1 h timeout and record solving time. We then compute the Spearman and Pearson correlations of mergeability versus solving time. Note that although pre-processing was used here, only approximately 5–10% of clauses were changed. Each row in Table 3 is aggregated over 10 formula series (correlations are aggregated using the Fisher transformation). Results are repeated for uniform 3SAT instances in Table 4, split by satisfiability. More specifically, if a given series of instances has both satisfiable and unsatisfiable formulas, we distinguish each sub-series of instances with the same satisfiability.

For the majority of formulas that are unsatisfiable, increasing the number of merges decreases solving time. The correlations, which are frequently greater in magnitude than 0.9, indicate a strong relationship over the benchmark. A possible explanation for this is that the additional merges allow the solver to learn smaller clauses faster, restricting the search space more quickly. Interestingly, there is a slight positive correlation when considering satisfiable instances.

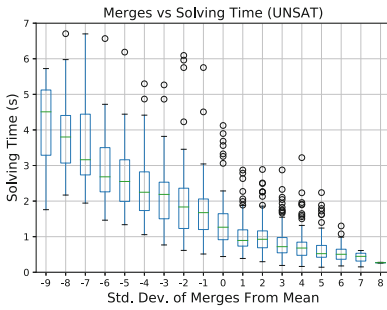
We further examined the set of uniform 3SAT instances with $n = 200$. In Fig. 1a, we display the scatter plot of the number of mergeable clause pairs versus solving time, distinguishing instances based on satisfiability and how they were generated. From Eq. 3, the expected number of mergeable clause pairs is 82. We then computed the sample variance of the number of mergeable clause pairs using the original random 3SAT instances in order to compute the standard deviation over the sample, which was 9.32. In Figs. 1b–d, we depict box plot distribution where the y-axis is again solving time, and the instances are grouped according to the number of standard deviations its merges are from the expected value of 82 merges, rounded toward zero. For example, an instance with 92 mergeable pairs would be 1.07 standard deviations from the expected value, and would fall into bucket 1. Note the clear downward slope over unsatisfiable instances in (c). For satisfiable instances, there does not appear to be much correlation. Also interestingly, the spread of solving times is much more significant if the number



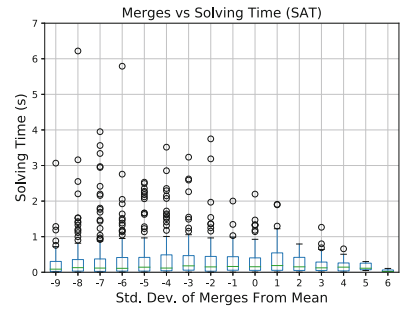
(a) Merges vs solving time.



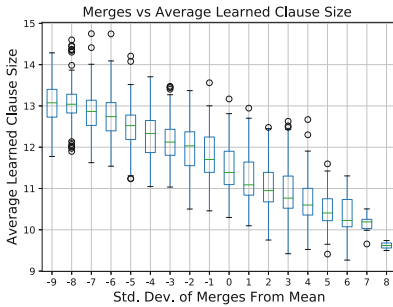
(b) All instances.



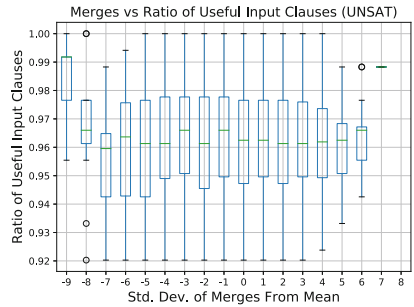
(c) Unsatisfiable instances.



(d) Satisfiable instances.



(e) Average learned clause size.



(f) Percentage of useful input clauses.

Fig. 1. (a) Scatter plot depicting the distribution of 3SAT instances, comparing the number of merges of the input clauses on the x-axis, and solving time on the y-axis. Includes traditional random 3SAT instances at the phase transition with 200 variables, as well as scaled instances using our generator. (b)–(d) Box plot distribution where the y-axis is again solving time, and the instances are grouped according to the number of standard deviations (rounded toward zero) its merges are from the expected value of 82 merges. Note the clear downward slope over unsatisfiable instances in (c). (e) Average learned clause time tends to decrease as the number of mergeable pairs increases. (f) The percentage of useful input clauses is not affected by mergeability in our experiments.

of mergeable pairs is small, as indicated by the many outliers on the left hand side of Fig. 1b, whereas most instances are easily solvable when the number of mergeable pairs is high.

Recall that an underlying motivation for studying mergeability was that the solver could learn shorter clauses during merge resolutions. A natural question is whether this occurs in practice. Figure 1e depicts a box plot comparing mergeability to the average learned clause size during search. As is apparent by the downward trend, instances that have more mergeable input tend to on average produce smaller learned clauses, supporting our intuition.

Last, although we intended to control for as many properties as possible when generating more mergeable formulas, we clearly cannot ensure that only the property of mergeability changes. A possible alternative explanation is that after increasing mergeability we introduce trivial unsatisfiable cores (i.e., small subset of input clauses that are sufficient to derive UNSAT), offering an alternative explanation of the correlation. In Fig. 1f, we empirically verify that this is not the case by measuring the number of *useful* input clauses for each SAT solver run of an unsatisfiable instance. We define a useful clause to an unsatisfiable proof as follows. Let P be the proof of unsatisfiability constructed by the SAT solver represented as a graph G , such that nodes represent clauses, input clauses have no incoming edges, and an edge exists from C_1 to C_2 iff the clause C_1 was in the implication graph used to derive C_2 . (Additional edges are needed to account for extra components of real-world solvers, such as clause minimization.) The final node added to the graph is the empty clause E . Then, if we reverse all edges in the graph, the *useful clauses* correspond to the set of nodes reachable from E .

As can be seen by the graph, approximately 96% of the input clauses were useful to the proof, regardless of mergeability. While this does not necessarily calculate the minimal unsatisfiable core, it more closely reflects the actual run of the SAT solver. Thus, we do not believe that unsatisfiable core size is a confounding factor in determining the runtime of these instances.

7 Future Work and Conclusions

We conducted a large-scale comprehensive study of several well-known structural parameters of SAT instances and their correlations with solver runtime over a diverse and representative set of 7000+ SAT competition instances. We found that while most of these features correlate with solving time for certain classes of formulas, these correlations were not strong for the entire benchmark suite we studied. We further introduced the mergeability parameter, and showed that it correlates well with many categories of instances, although satisfiability seems to affect this correlation. Finally, in order to isolate the effects of mergeability, we performed scaling studies. Specifically, we describe a formula generator capable of scaling the mergeability parameter, and showed that mergeability of unsatisfiable instances tend to strongly correlate negatively with solver runtime (or strongly correlate positively with solver performance). Also, we showed that the solver tends to produce shorter learned clauses on average for instances with higher

mergeability. Given the strong correlations, we believe that mergeability may be a useful measure in portfolio solvers.

References

1. The international SAT Competitions web page (2017). <http://www.satcompetition.org/>
2. Andrews, P.B.: Resolution with merging. In: Siekmann, J.H., Wrightson, G. (eds.) *Automation of Reasoning*, pp. 85–101. Springer, Heidelberg (1968). https://doi.org/10.1007/978-3-642-81955-1_6
3. Ansótegui, C., Bonet, M.L., Giráldez-Cru, J., Levy, J.: The fractal dimension of SAT formulas. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) *IJCAR 2014. LNCS (LNAI)*, vol. 8562, pp. 107–121. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_8
4. Ansótegui, C., Bonet, M.L., Levy, J.: On the structure of industrial SAT instances. In: Gent, I.P. (ed.) *CP 2009. LNCS*, vol. 5732, pp. 127–141. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_13
5. Ansótegui, C., Giráldez-Cru, J., Levy, J.: The community structure of SAT formulas. In: Cimatti, A., Sebastiani, R. (eds.) *SAT 2012. LNCS*, vol. 7317, pp. 410–423. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_31
6. Biere, A., Heule, M., van Maaren, H.: *Handbook of Satisfiability*, vol. 185. IOS press (2009)
7. Dilkina, B., Gomes, C.P., Sabharwal, A.: Tradeoffs in the complexity of backdoors to satisfiability: dynamic sub-solvers and learning during search. *Ann. Math. Artif. Intell.* **70**(4), 399–431 (2014). <https://doi.org/10.1007/s10472-014-9407-9>
8. Giráldez-Cru, J., Levy, J.: Locality in random sat instances. In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)* (2017)
9. Godefroid, P., Levin, M.Y., Molnar, D.A., et al.: Automated whitebox fuzz testing. In: *Network and Distributed System Security Symposium*, pp. 151–166. Internet Society (2008)
10. Jordi, L.: On the classification of industrial SAT families. In: *International Conference of the Catalan Association for Artificial Intelligence*, p. 163. IOS Press (2015)
11. Kilby, P., Slaney, J., Thiébaux, S., Walsh, T.: Backbones and backdoors in satisfiability. In: *AAAI Conference on Artificial Intelligence*, pp. 1368–1373. AAAI Press (2005)
12. Liang, J.H., Oh, C., Ganesh, V., Czarnecki, K., Poupart, P.: Maple-COMSPS, mapleCOMSPS LRB, maplecomsps CHB. *SAT Competition*, p. 52 (2016)
13. Mateescu, R.: Treewidth in Industrial SAT Benchmarks (2011). <http://research.microsoft.com/pubs/145390/MSR-TR-2011-22.pdf>
14. Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., Troyansky, L.: Determining computational complexity from characteristic ‘phase transitions’. *Nature* **400**(6740), 133–137 (1999)
15. Newsham, Z., Ganesh, V., Fischmeister, S., Audemard, G., Simon, L.: Impact of community structure on SAT solver performance. In: Sinz, C., Egly, U. (eds.) *SAT 2014. LNCS*, vol. 8561, pp. 252–268. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_20
16. Pipatsrisawat, K., Darwiche, A.: A new clause learning scheme for efficient unsatisfiability proofs. In: *AAAI Conference on Artificial Intelligence*, pp. 1481–1484 (2008)

17. Pohl, R., Stricker, V., Pohl, K.: Measuring the structural complexity of feature models. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, pp. 454–464. IEEE Press (2013)
18. Robertson, N., Seymour, P.D.: Graph minors. III. Planar tree-width. *J Comb Theor Series B* **36**(1), 49–64 (1984)
19. Williams, R., Gomes, C., Selman, B.: On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. In: International Conference on Theory and Applications of Satisfiability Testing, pp. 222–230. Springer, Heidelberg (2003). <https://doi.org/10.1.1.128.5725>



Learning-Sensitive Backdoors with Restarts

Edward Zulkoski¹(✉), Ruben Martins², Christoph M. Wintersteiger³,
Robert Robere⁴, Jia Hui Liang¹, Krzysztof Czarnecki¹, and Vijay Ganesh¹

¹ University of Waterloo, Waterloo, ON, Canada
ezulkosk@uwaterloo.ca

² Carnegie Mellon University, Pittsburgh, PA, USA

³ Microsoft Research, Cambridge, UK

⁴ University of Toronto, Toronto, ON, Canada

Abstract. Restarts are a pivotal aspect of conflict-driven clause-learning (CDCL) SAT solvers, yet it remains unclear when they are favorable in practice, and whether they offer additional power in theory. In this paper, we consider the power of restarts through the lens of backdoors. Extending the notion of learning-sensitive (LS) backdoors, we define a new parameter called learning-sensitive with restarts (LSR) backdoors. Broadly speaking, we show that LSR backdoors are a powerful parametric lens through which to understand the impact of restarts on SAT solver performance, and specifically on the kinds of proofs constructed by SAT solvers. First, we prove that when backjumping is disallowed, LSR backdoors can be exponentially smaller than LS backdoors. Second, we demonstrate that the size of LSR backdoors are dependent on the learning scheme used during search. Finally, we present new algorithms to compute upper-bounds on LSR backdoors that intrinsically rely upon restarts, and can be computed with a single run of a CDCL SAT solver. We empirically demonstrate that this can often produce much smaller backdoors than previous approaches to computing LS backdoors. We conclude with empirical results on industrial benchmarks which demonstrate that rapid restart policies tend to produce more “local” proofs than other heuristics, in terms of the number of unique variables found in learned clauses of the proof.

Keywords: SAT solving · Backdoors · Restarts · CDCL

1 Introduction

Restarts are a pivotal aspect of conflict-driven clause-learning (CDCL) SAT solvers. Not only are they crucially important to the performance of CDCL solver implementations, but foundational theoretical work on the power of CDCL intrinsically relies on exploiting restarts [1, 24]. For example, the powerful theorems by Pipatsrisawat and Darwiche that show polynomial equivalence between

CDCL SAT solvers (with perfect branching and restarts) and the general resolution proof system seem to crucially rely on the use of restarts. Nevertheless, it remains unclear when or why it is favorable to restart in practice, and whether restarts truly make CDCL SAT solvers a more powerful proof system in theory. Further, it is unclear whether restarts provide any additional power over backjumping, which is standard in all modern CDCL SAT solvers. Understanding restarts can go a long way in explaining one of the most important problem in SAT and SMT solver research, namely, “why are solvers efficient for a large class of industrial applications?”

Complexity theorists have long proposed a variety of parameters in explaining the surprising power of certain algorithms at solving NP-complete problems. One such class of parameters, originally introduced by Williams, Gomes, and Selman, are *backdoors* [29]. Since their seminal paper, a variety of backdoors have been proposed in an attempt to understand why and for what kind of instances SAT solving algorithms work efficiently. Intuitively, backdoors measure how many variables need to be assigned such that a polynomial-time subsolver can solve the residual formula. For “traditional” types of backdoors (i.e. strong and weak backdoors [29]), if the backdoor is small, then efficient algorithms can determine satisfiability by trying all possible assignments to the backdoor. Unfortunately, traditional backdoors do not account for some pivotal aspects of CDCL SAT solvers, such as clause-learning or restarts. Dilkina et al. extended traditional backdoors to learning-sensitive (LS) backdoors in order to account for the power of clause-learning during the search performed by a SAT solver [9]. They showed that these learning-sensitive (LS) backdoors are exponentially smaller than traditional strong backdoors on certain class of formulas. However, they only considered a single configuration of a CDCL SAT solver, namely one that uses the first unique implication point (1UIP) learning scheme and disallows restarts.

In this work, we extend the notion of LS backdoors to allow restarts by introducing the concept of learning-sensitive with restarts (LSR) backdoors. Our main contribution is an exponential separation between LS and LSR backdoors (using the 1UIP clause-learning scheme), under the condition that backjumping during search is not allowed (only backtracking is allowed). Determining whether or not restarts add significant power to CDCL SAT solvers in full generality remains a major and important open problem. We hope that our work will be a useful step toward tackling this problem.

We further consider the effect of different clause-learning schemes on the size of LSR backdoors. We show that if a formula has a backdoor set B under the decision learning (DL) scheme (an “LSR-DL backdoor”), then B is also an LSR-1UIP backdoor. The converse however does not hold, and we show a family of formulas where the smallest LSR-1UIP backdoor is exponentially smaller than the smallest LSR-DL backdoor. This may indicate that 1UIP can allow the solver to remain more “local” during search when compared to other heuristics.

In the context of strong and weak backdoors, if we are given *a priori* knowledge of a backdoor B of size n , then we can simply invoke the subsolver on all 2^n

assignments to the backdoor to determine satisfiability. The situation is not so simple in the context of LSR (and LS) backdoors, which rely upon the order in which the search space is explored to learn clauses. We demonstrate pitfalls that may arise for a solver trying to exploit *a priori* knowledge of LSR backdoors. We describe formulas such that if the solver is given additional unit clauses for free (thus shrinking the search space), then it is impossible to determine unsatisfiability by only branching on the LSR backdoor (of the original formula without the unit clauses). We also describe issues that can arise if multiple conflicts can be learned after the same sequence of decisions. Under certain probabilistic assumptions, we show that *even if the solver is given a perfect branching sequence witnessing the backdoor*, it still may have to branch on additional variables. This result exploits the fact that a typical CDCL solver only learns one clause per conflict.

Finally, through an extension of results from [24], we show that the set of variables found in the learned clauses used in the proof of unsatisfiability constitute an LSR backdoor. We use this result to empirically compare the proofs produced by various restart policies. Results suggest that rapid restart policies tend to have more “local proofs” (in terms of the aforementioned measure) on several classes of industrial SAT formulas.

2 Related Work

Williams et al. introduced traditional weak and strong backdoors for both SAT and CSP [29]. The size of backdoors with respect to subsolvers different from unit propagation (UP) was considered in [10, 17]. Several extensions of traditional strong and weak backdoors have been proposed. Backdoor keys measure certain interdependencies of variables in the backdoor set [26]. Backdoor trees refine strong backdoors by measuring what fraction of the total search of a strong backdoor must be explored before determining satisfiability [27]. The backdoor treewidth is a measure that is bounded above by both the strong backdoor size and also the treewidth of certain graphical abstractions of the formula [11]. It was shown in [16] that there is little relationship between weak backdoors and the backbone of a formula. Dilkina et al. were the first to consider clause learning in the context of backdoors and introduced learning-sensitive (LS) backdoors. LS backdoors allow clause-learning to occur while traversing the assignment tree of backdoor variables, which yield exponentially smaller backdoors than strong backdoors for certain class of instances [8, 9].

Our work is inspired by several lines of work aimed at relating the power of CDCL to general resolution. Pool resolution was first introduced to model CDCL without restarts [28], and it was shown that pool resolution is exponentially stronger than regular resolution. Resolution trees with lemmas were similarly introduced in [7], and more closely match clause-learning algorithms in practice. In their seminal paper, Beame et al. formally defined CDCL as a proof system and showed that CDCL can polynomially simulate natural refinements of general resolution [2]. However, their approach required assumptions that do not

reflect typical CDCL implementations, such as choosing to ignore unit propagations when preferable. Hertel et al. also showed that CDCL without restarts can effectively polynomially simulate general resolution, but required certain modifications to input formulas [14]. It was also shown that a non-restarting CDCL solver can polynomially simulate a restarting solver, but the approach requires adding additional variables to the formula as a “counter,” based on the number of variables in the original formula [4].

Recent approaches that show CDCL solving efficiently simulates general resolution require restarts. In their papers [24, 25], the authors showed that CDCL without the assumptions from [3] can polynomially simulate general resolution. The approach relies upon the notion of *1-empowerment* [23], which is the dual of *clause absorption* [1]. However, crucially, they assume that the branching and restarts in CDCL solvers are perfect (i.e., non-deterministic). In Atserias et al. [1], the authors assume randomized branching and restarts, instead of non-deterministic ones. Specifically, they demonstrate that rapidly restarting solver with sufficiently many random decisions can effectively simulate bounded-width resolution. Many questions in this context remain open. For example, can the above simulations be modified to not require restarts? Further, can we construct realistic models of CDCL solvers (with “realistic” branching and restarts) and determine their relative power vis-a-vis well-known proof systems such as general resolution.

On the empirical side, several works have studied the performance of various restart policies. Huang reported on a comprehensive evaluation of several restart policies [15], which demonstrated the strength of “dynamic” restart policies such as those based on the Luby sequence [19]. Biere performed an evaluation of restart strategies on more modern solvers [5]. Among their results, they showed that static restart policies can perform as well as dynamic strategies. Haim et al. showed that more rapid restart policies tend to require fewer conflicts before determining satisfiability, however this does not always lead to faster solving [13].

3 Background

We assume basic familiarity with the Boolean satisfiability problem, CDCL solvers and the standard notation used by solver developers and complexity theorists. For an overview we refer the reader to [6]. The CDCL model we used is the same as the one presented by Pipatsrisawat and Darwiche in [24, 25] (cf. Algorithm 1 in [24]).

We assume that Boolean formulas are given in conjunctive normal form (CNF). For a formula F , we denote its variables as $vars(F)$. A *model* for a formula F refers to a complete satisfying assignment to F . The term *trail* refers to the sequence of variable assignments (either through *decisions* made by CDCL solver’s branching heuristic, or through propagation), in the order they have been assigned, at any given point in time during the run of a solver. The *decision level* of a literal denotes how many decision variables occurred up to and including the literal on the trail. Learned clauses are derived by analyzing the *implication*

graph, which comprises the decisions and implications (propagations) that led to a conflict. The first unique implication point clause-learning scheme (denoted 1UIP) is the most common in practice [20]. We also consider the decision learning scheme (denoted DL) which creates a conflict clause from the set of decision literals which led to the conflict [30]. CDCL solvers typically *backjump* to the second highest decision level in the conflict clause after reaching a conflict, which allows the solver to quickly ignore large areas of the search space that are provably unsatisfiable. If backjumping is not allowed, the solver *backtracks* to the most recent decision level where the opposite polarity of the literal has not been explored. The learned (conflict) clause defines a cut in the implication graph; we denote the subgraph on the side of the cut which contains the conflict node as the *conflict side*, and the other side as the *reason side*.

We use the term “querying a variable” to mean that the solver assigns a value to it and then performs unit propagations until saturation (i.e., no more propagations are possible). For a set of clauses Δ and a literal l , $\Delta \vdash_1 l$ denotes that unit propagation (UP) can derive l from Δ . If $\Delta \vdash_1 \perp$ (i.e. UP derives the empty clause), then the solver is in a *1-inconsistent* or conflicting state, else it is in a *1-consistent* or non-conflicting state.

Definition 1 (Absorption [1]). *Let Δ be a set of clauses, let C be a non-empty clause and let x^α be a literal in C . Then Δ absorbs C at x^α if every non-conflicting state of the solver that falsifies the literals in $C \setminus \{x^\alpha\}$ assigns x to α . If Δ absorbs C at every literal, then Δ absorbs C .*

The intuition behind absorbed clauses is that adding an already absorbed clause C to Δ is in some sense redundant, since unit propagations that could be realized with C have already been realized by the solver using clauses in Δ .

3.1 Backdoors

Backdoors are defined with respect to *subsolvers*, which are algorithms that can solve certain classes of SAT instances in polynomial-time. Example subsolvers include the unit propagation (UP) algorithm [10, 29], which is also what we focus on as it is a standard subroutine implemented in CDCL SAT solvers. Given a partial assignment $\alpha : B \rightarrow \{0, 1\}$, the simplification of F with respect to α , denoted $F[\alpha]$, removes all clauses that are satisfied by α , and removes any literals that are falsified by α in the remaining clauses. A *strong backdoor* of a formula F is a set of variables B such that for every assignment α to B , $F[\alpha]$ can be determined to be satisfiable or unsatisfiable by the UP subsolver [29]. A set of variables B is a *weak backdoor* with respect to a subsolver S if there exists an assignment α to B such that the UP subsolver determines the formula to be satisfiable. Backdoors were further extended to allow clause-learning to occur while exploring the search space of the backdoor:

Definition 2 (Learning-sensitive (LS) backdoor [9]). *A set of variables $B \subseteq \text{vars}(F)$ is an LS backdoor of a formula F with respect to a subsolver S if there exists a search-tree exploration order such that a CDCL SAT solver without*

restarts, branching only on B and learning clauses at the leaves of the tree with subsolver S, either finds a model for F or proves that F is unsatisfiable.

Learning-sensitive with restarts (LSR) backdoors extend LS backdoors to allow restarts. By allowing restarts, the solver may learn clauses from different parts of the search-tree of B , which otherwise may not be accessible without restarting. Where appropriate, we parameterize LSR backdoors by the learning scheme: either LSR-1UIP or LSR-DL in this work. If no learning scheme is explicitly mentioned, then 1UIP is assumed.

3.2 Two Formula Families

Dilkina et al. introduced two families of formulas that were used to demonstrate an exponential separation between LS backdoors and strong backdoors [9]. We describe them here, as several variations of the formulas will be useful for deriving our results in the following sections.

The Gadget \mathcal{U} : First, we introduce the following formula gadget (we use the term gadget to refer to sub-formulas that we then use to build more complex formulas):

$$\mathcal{U}(q, a, b) := (q \vee a) \wedge (q \vee b) \wedge (q \vee \neg a \vee \neg b)$$

We describe the behavior of CDCL SAT solvers if it were to receive one of these gadgets as input: observe that if the solver were to branch on $\neg q$ (i.e., set q to false), it will immediately reach a conflict using these three clauses. Clause-learning will then derive the unit clause (q) (for both 1UIP and DL).

The formulas below are both defined on variables x_1, x_2, \dots, x_n (referred as the “ X ” variables), and also three auxiliary sets of variables $\{q_\alpha\}_{\alpha \in \{0,1\}^n}$, $\{a_\alpha\}_{\alpha \in \{0,1\}^n}$, $\{b_\alpha\}_{\alpha \in \{0,1\}^n}$. For any assignment $\alpha \in \{0,1\}^n$ let $C_\alpha = x_1^{1-\alpha_1} \vee x_2^{1-\alpha_2} \vee \dots \vee x_n^{1-\alpha_n}$ denote the clause on X variables which is uniquely falsified by the assignment α , and let $X \mapsto \alpha$ mean that we branch on each x variable, such that $x_i = \alpha_i$.

Let the parameter \mathcal{O} define an ordering over the bit-strings in $\{0,1\}^n$. In [9], this is assumed to be the lexicographic ordering, denoted as LEX . We are now ready to define below the two families of formulas that Dilkina et al. introduced in their paper.

The Family $\mathcal{F}_\mathcal{O}$ of Boolean Formulas: Consider the formula

$$\mathcal{F}_\mathcal{O} = \bigwedge_{\alpha \in \{0,1\}^n} (C_\alpha \vee \bigvee_{\alpha' \leq_\mathcal{O} \alpha} \neg q_{\alpha'}) \wedge \mathcal{U}(q_\alpha, a_\alpha, b_\alpha). \tag{1}$$

As described in the paper by Dilkina et al. [9], the smallest LS backdoor when \mathcal{O} is the lexicographic ordering, denoted \mathcal{F}_{LEX} , is the set of X variables and therefore of size n , and works as follows. To see that X is an LS backdoor, we first branch on all X variables setting them to *false*. This then propagates $\neg q_{0\dots 0}$, which leads to learning $(q_{0\dots 0})$ from the corresponding clauses in $\mathcal{U}(\dots)$.

Since the learned clause is unit, the solver backjumps to decision level 0, at which point we repeat the process by assigning X according to the next lexicographic assignment $0\dots 01$ (i.e., we set x_1, \dots, x_{n-1} to *false*, and x_n to *true*). Repeating this, we derive all (q_α) clauses in lexicographic order and eventually derive UNSAT. Note importantly that when deriving each (q_α) , all previously learned clauses $\{(q_{\alpha'}) \mid \alpha' \leq \alpha\}$ are used for propagation, so branching in lexicographic order is essential in establishing that X variables constitute the smallest LS backdoor.

The Family \mathcal{G}_O of Boolean Formulas: Now, consider the formula

$$\mathcal{G}_O = \bigwedge_{\substack{\alpha \in \{0,1\}^n, \\ \alpha \neq 1^n}} ((C_\alpha \vee \neg q_\alpha) \wedge \mathcal{U}(q_\alpha, a_\alpha, b_\alpha)) \wedge (C_{1^n} \vee \bigvee_{\alpha'} \neg q_{\alpha'}) \wedge \mathcal{U}(q_{1^n}, a_{1^n}, b_{1^n}). \quad (2)$$

We can lift the argument for formula \mathcal{F}_{LEX} to the formula \mathcal{G}_{LEX} to show that X variables constitute the smallest LS backdoor in this case as well. We iterate through all assignments to X variables in lexicographic order, learning each unit clause (q_α) along the way. However, the final assignment $\alpha = 1^n$ is treated differently, and importantly must be queried last. After learning all previous (q) clauses and assigning the X variables according to 1^n , we propagate $\neg q_{1^n}$, which again leads to conflict in the usual manner, ultimately proving the formula to be UNSAT.

4 Extending Learning-Sensitive Backdoors

Dilkina et al. introduced LS backdoors in [9], however they only considered a single CDCL solver configuration, namely, a non-restarting solver with the UIP clause-learning scheme. They demonstrated that LS backdoors may be exponentially smaller than strong backdoors using this configuration. Here, we provide several separation results, first comparing LS backdoors with and without restarts, and then various clause-learning schemes. We also demonstrate some pitfalls that may arise for a solver trying to exploit *a priori* knowledge of LSR backdoors.

4.1 Separating LS and LSR Backdoors

In this section we prove that for certain kinds of formulas the minimal LSR backdoors are exponentially smaller than the minimal LS backdoors under the assumption that the learning scheme is UIP and that the CDCL solver is only allowed to backtrack (and not backjump). Backjumping raises issues particularly when unit clauses are learned. Since the solver with backjumping will return to the second highest decision level in the clause (which defaults to level 0 in unit clauses), these conflicts effectively allow the solver to get a “free” restart.

Further, while there is a general misconception that backjumping is always better than backtracking, recent work by Nadel and Ryvchin demonstrate that solvers that occasionally backtrack instead of backjump may perform better than state of the art solvers [22]. Thus we do not find this assumption too unreasonable.

We show that the family of formulas \mathcal{F} (but for a different ordering \mathcal{O}) can be used to separate LS backdoor size from LSR backdoor size. Observe that for any ordering \mathcal{O} the variables x_1, x_2, \dots, x_n form an LSR backdoor for $\mathcal{F}_{\mathcal{O}}$.

Lemma 1. *Let \mathcal{O} be any ordering of $\{0, 1\}^n$. The X -variables form an LSR-backdoor for the formula $\mathcal{F}_{\mathcal{O}}$.*

Proof. For each assignment $\alpha \in \{0, 1\}^n$ (ordered by \mathcal{O}), assign α to the X variables by decision queries. By the structure of $\mathcal{F}_{\mathcal{O}}$, as soon as we have a complete assignment to the X variables, we will unit-propagate to a conflict and learn a q_{α} variable as a conflict clause; after that we restart. Once all of these assignments are explored we will have learned the unit clause q_{α} for every assignment α , and so we can simply query the X variables in any order (without restarts) to yield a contradiction, since every assignment to the X variables will falsify the formula.

Note that the formula $\mathcal{F}_{\mathcal{O}}$ depends on $N = O(2^n)$ variables, and so the size of this LSR backdoor is $O(\log N)$. Furthermore, observe that the X variables will also form an LS backdoor if we can query the assignments $\alpha \in \{0, 1\}^n$ according to \mathcal{O} without needing to restart — for example, if \mathcal{O} is the lexicographic ordering. This suggests the following definition, which captures the orderings \mathcal{O} of $\{0, 1\}^n$ that can be explored by a CDCL algorithm without restarts:

Definition 3. *Let \mathcal{T}_X be the collection of all depth- n decision trees on X variables, where we label each leaf ℓ of a tree $T \in \mathcal{T}_X$ with the assignment $\alpha \in \{0, 1\}^n$ obtained by taking the assignments to the X variables on the path from the root of T to ℓ . For any $T \in \mathcal{T}_X$, let $\mathcal{O}(T)$ be the ordering of $\{0, 1\}^n$ obtained by reading the assignments labeling the leaves of T from left to right.*

To get some intuition for our lower-bound argument, consider an ordering $\mathcal{O}(T)$ for some decision tree $T \in \mathcal{T}_X$. By using the argument in Lemma 1 the formula $\mathcal{F}_{\mathcal{O}(T)}$ will have a small LS backdoor, obtained by querying the X variables according to the decision tree T . Now, take any two assignments $\alpha_i, \alpha_j \in \mathcal{O}(T)$ and let $\mathcal{O}(T)'$ be the ordering obtained from $\mathcal{O}(T)$ by swapping the indices of α_i and α_j . If we try and execute the same CDCL algorithm without restarts (corresponding to the ordering $\mathcal{O}(T)$) on the new formula $\mathcal{F}_{\mathcal{O}(T)'}$, the algorithm will reach an inconclusive state once it reaches the clause corresponding to α_j in $\mathcal{O}(T)'$ since at that point the assignment to the X variables will be α_i . Thus, it will have to query at least one additional variable (for instance q_{α_j}), which increases the size of the backdoor by one. We can generalize the above argument to multiple “swaps” — the CDCL algorithm without restarts querying the variables according to $\mathcal{O}(T)$ would then have to query one extra variable for every q_{α} which is “out-of-order” with respect to $\mathcal{O}(T)$.

This discussion leads us to the following complexity measure: for any ordering $\mathcal{O} \in \{0, 1\}^n$ (not necessarily obtained from a decision tree $T \in \mathcal{T}_X$) and any ordering of the form $\mathcal{O}(T)$, let

$$d(\mathcal{O}, \mathcal{O}(T)) = |\{\alpha' \in \{0, 1\}^n \mid \exists \alpha \in \{0, 1\}^n : \alpha' <_{\mathcal{O}} \alpha, \alpha <_{\mathcal{O}(T)} \alpha'\}|.$$

Informally, $d(\mathcal{O}, \mathcal{O}(T))$ counts the number of elements of \mathcal{O} which are “out-of-order” with respect to $\mathcal{O}(T)$ as we have discussed above. We are able to show that the above argument is fully general:

Lemma 2. *Let \mathcal{O} be any ordering of $\{0, 1\}^n$, and let \mathcal{T}_X denote the collection of all complete depth- n decision trees on X variables. Then any learning-sensitive backdoor of $\mathcal{F}_{\mathcal{O}}$ has size at least*

$$\min_{T \in \mathcal{T}_X} d(\mathcal{O}, \mathcal{O}(T)).$$

This reduces our problem to finding an ordering \mathcal{O} for which *every* ordering of the form $\mathcal{O}(T)$ has many elements which are “out-of-order” with respect to \mathcal{O} (again, intuitively for every mis-ordered element in the LS backdoor we will have to query at least one more variable from Q).

Lemma 3. *For any $n > 3$ there exists an ordering \mathcal{O} of $\{0, 1\}^n$ such that for every decision tree $T \in \mathcal{T}_X$ we have*

$$d(\mathcal{O}, \mathcal{O}(T)) \geq 2^{n-2}.$$

Proof Sketch. We define the ordering, and leave the full proof of correctness to the companion supplemental material. Let $\beta_1, \beta_2, \dots, \beta_N$ be the lexicographic ordering of $\{0, 1\}^n$, and for any string β_i define $\bar{\beta}_i$ to be the string obtained by flipping each bit in β_i . Then let \mathcal{O} be the ordering

$$\beta_1, \bar{\beta}_1, \beta_2, \bar{\beta}_2, \dots, \beta_{N/2}, \bar{\beta}_{N/2}.$$

Theorem 1. *For every $n > 3$, there is a formula F_n on $N = O(2^n)$ variables such that the minimal LSR backdoor has $O(\log N)$ variables, but every LS backdoor has $\Omega(N)$ variables.*

4.2 The Effect of Clause-Learning Schemes

We next show that the size of the minimal LSR backdoor is dependent on the solver’s underlying clause-learning scheme. We draw comparisons between the 1UIP and DL schemes. Note that for all following results, we allow backjumping (as opposed to just backtracking as in the previous subsection). A takeaway from these results is that the LSR backdoor gives us a deeper theoretical understanding of why DL learning schemes are weaker than 1UIP ones.

Theorem 2. *Let F be a formula with an LSR-DL backdoor of size n . Then the smallest LSR-1UIP backdoor for F has size at most n .*

Proof. Consider the sequence of learned clauses in the proof that witnesses the smallest LSR-DL backdoor. Then the DL-solver must have branched on all the variables in the learned clauses given the nature of the DL scheme. Let B be this set of variables. Then we can absorb them one-by-one by only branching on the variables in those clauses. (This result is discussed further in Theorem 5).

Theorem 3. *There exists an infinite family of formulas such that the smallest LSR-DL (resp. LS-DL) backdoor for each instance is exponentially larger than the smallest LSR-1UIP (resp. LS-1UIP) backdoor.*

Proof. We show this using the formula family \mathcal{F}_{LEX} . The result follows analogously to the separation of LS-1UIP backdoors and strong backdoors in [9]. We have already demonstrated that the smallest LSR-1UIP backdoor is of size $|X| = n$.

Since each formula in \mathcal{F}_{LEX} is minimally unsatisfiable, in order to derive UNSAT we must “make use” of each clause through some propagation. Let C_{long} be the clause with the largest number of q_α literals. If our branching sequence only branched previously on variables in X , then all learned clauses will only include variables in X , and in particular could not propagate the Q variables in C_{long} . The only way we could have derived (q_α) clauses previously is through branching on them, which would also increase the size of the backdoor. Thus, we must branch on at least the n variables from X and $2^n - 1$ variables from Q in order to propagate on C_{long} .

4.3 Properties of LS and LSR Backdoors

In the case of traditional strong or weak backdoors with UP as the subsolver, for a given formula F , it is easy to show that adding clauses to F can only decrease the size of the backdoor. This is not the case for LS and LSR backdoors. Note that we believe this result relates to the notion of *natural proof systems* – proof systems such that if unit clauses are added to the formula, it will not increase the size of the smallest resolution proof [3]. It was shown in their work that, under their theoretical model, clause learning without restarts is either *not* natural or not as powerful as general resolution.

Observation 1. Given formulas F_1 and F_2 , the formula $F_1 \wedge F_2$ may have a larger LSR (or LS) backdoor than either individual formula.

Example 1. Consider $F_1 \in \mathcal{F}_{LEX}$, and let F_2 be the single unit clause (x_1) . Note that F_2 subsumes the first half of the clauses defined over C_α ’s (since we are using the lexicographic ordering). Therefore, we can no longer utilize those clauses to derive conflicts, since the solver will never set x_1 to false. It is easy to show that the solver must begin branching on variables not in X in order to solve the formula, and that the number of q variables that must be necessarily branched on is larger than $|X|$.

Next, we show that even if the solver is given a perfect branching sequence witnessing an LSR backdoor, there exist formulas where the solver may still need

to branch on additional variables. Our result relies on the following probabilistic assumption:

Definition 4 (Uniform Conflict Choice (UCC) Assumption). *Let Δ be a set of clauses and D be a sequence of decisions, such that for any proper prefix P of D , $\Delta \wedge P$ is 1-consistent, but $\Delta \wedge D$ is 1-inconsistent, i.e., it causes a conflict. Further, assume that there are n unique conflict clauses that can be derived after branching on D , depending on the order in which literals are propagated. The Uniform Conflict Choice assumption states that the solver always chooses the conflict clause to learn uniformly at random from all possible conflict clauses.*

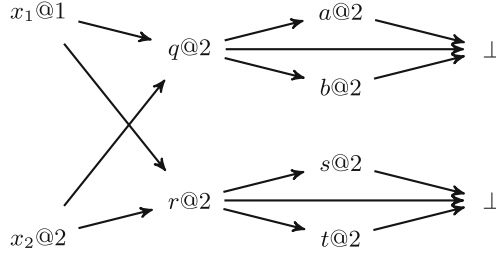


Fig. 1. Example of multiple conflicts after making decisions x_1 and x_2 . The number after the ‘@’ symbol denotes the decision level of the literal.

Example 2. Consider Fig. 1. Depending on whether q or r is propagated first, a solver using *1UIP* may learn the unit clause $(\neg q)$ or $(\neg r)$, respectively. Under the UCC assumption, each has a 50% likelihood of being derived. Since solvers typically only learn one clause per conflict, after one of the two clauses is learned, the solver will backjump to decision level 0 (since both clauses are unit), ignoring the other possible clause.

Theorem 4. *There exists an infinite family of formulas such that for any $\delta > 0$, the probability of realizing the minimal LS backdoor (or LSR backdoor), even given the perfect branching sequence, is less than δ , under the UCC assumption.*

Proof. We construct a new family of formulas \mathcal{G}_{2LEX} by starting with \mathcal{G}_{LEX} and adding a duplicate set of clauses, where the X variables are reused in these clauses, but each q_α , a_α , and b_α is replaced by a fresh r_α , s_α , and t_α , respectively:

$$\begin{aligned}
 \mathcal{G}_{2\mathcal{O}} = & \bigwedge_{\alpha \in \{0,1\}^n, \alpha \neq 1^n} ((C_\alpha \vee \neg q_\alpha) \wedge \mathcal{U}(q_\alpha, a_\alpha, b_\alpha)) \wedge \\
 & \bigwedge_{\alpha \in \{0,1\}^n, \alpha \neq 1^n} ((C_\alpha \vee \neg r_\alpha) \wedge \mathcal{U}(r_\alpha, s_\alpha, t_\alpha)) \wedge \\
 & (C_{1^n} \vee \bigvee_{\alpha'} \neg q_{\alpha'}) \wedge \mathcal{U}(q_{1^n}, a_{1^n}, b_{1^n}) \wedge \\
 & (C_{1^n} \vee \bigvee_{\alpha'} \neg r_{\alpha'}) \wedge \mathcal{U}(r_{1^n}, s_{1^n}, t_{1^n}).
 \end{aligned} \tag{3}$$

Note that the first and third lines are exactly the clauses from $G_{\mathcal{O}}$. We first show that, as in the case for \mathcal{G}_{LEX} , the X variables constitute an LS backdoor for \mathcal{G}_{2LEX} . Let P be the branching sequence that witnesses X as an LS backdoor for \mathcal{G}_{LEX} (i.e. by branching on X lexicographically). For each assignment α to X ($\alpha \neq 1^n$), the solver can derive one of two conflict clauses depending on the order of propagations: either (q_α) or (r_α) (as in Example 1). If for every $\alpha \neq 1^n$ the solver derives some q_α , then the solver can derive the same proof as derived for the \mathcal{G}_{LEX} formula. This effectively ignores any clauses that contain some r_α . The same holds if r_α is always chosen, and the clauses with q_α are ignored. Thus, the branching sequence P witnesses that X is an LS backdoor for \mathcal{G}_{2LEX} . Further, it is clear that branching on any q, r, a, b, s , or t variables cannot reduce the size of the backdoor through a similar argument as in the case for \mathcal{G}_{LEX} , and also given the fact that no q or r variables share a clause. Thus X is the smallest backdoor for \mathcal{G}_{2LEX} .

Suppose that instead of either always learning q_α , or always learning r_α , that a mix of the two are learned. Then, when the final lexicographic assignment is reached, which sets $X \mapsto 1 \dots 1$, we are not able to propagate the final literal (either q_{1^n} or r_{1^n}), since the clauses listed on lines 3 and 4 of Eq. 3 will have multiple unassigned literals. Thus, the solver is forced to branch on q or r literals to derive UNSAT.

It remains to compute the probability of this occurring under the UCC assumption. Given $|X| = n$, there are 2^{n-1} assignments to X that occur before the conflict involving C_{1^n} , and for each assignment α , we can learn either (q_α) or (r_α) . Then the likelihood of picking either all q_α 's or all r_α 's is $1/2^{n-2}$. Given a fixed δ , choosing any $n \geq \lceil \log_2(1/\delta) \rceil + 2$ completes our result.

5 Relating LSR Backdoors to CDCL Proofs

We next show connections to LSR backdoors and the proofs generated by CDCL solvers on unsatisfiable instances. Specifically, we show that if B is the union of all variables found in the “useful clauses” of the proof, then B is an LSR backdoor for the formula. More importantly, we show that frequent restarts often result in smaller and more “local” proofs.

Definition 5 (Useful clauses). *Let P be the proof of unsatisfiability constructed by the SAT solver represented as a graph G , such that nodes represent clauses, input clauses have no incoming edges, and an edge exists from C_1 to C_2 iff the clause C_1 was in the implication graph used to derive C_2 . (Additional edges are needed to account for extra components of real-world solvers, such as clause minimization.) The final node added to the graph is the empty clause E . Then, if we reverse all edges in the graph, the useful clauses correspond to the set of nodes reachable from E .*

Theorem 5. *Let S be a CDCL solver that has been used to determine a formula F is unsatisfiable. Let Δ be the set of useful clauses learned by S while solving F . Then a fresh CDCL solver S' can absorb all clauses in Δ , thus deriving UNSAT,*

by only branching on the variables in Δ . (By fresh we mean that the solver starts from an initial state on the input F , and branches only on the variables in Δ).

Proof. The result follows similarly to the result of [24], where they show that CDCL can simulate general resolution. If every clause in Δ is absorbed, we are done. Otherwise, there must exist some $C \in \Delta$ that is both 1-empowering and 1-provable (refer [24]). We can absorb C by a sequence of restarts and decisions only on variables in C (cf. proof of Proposition 2 in [24]). This process repeats until all clauses in Δ are absorbed.

The result can be easily extended to consider satisfiable instances, but here we focus on unsatisfiable ones. We use the number of variables spanning the useful clauses to intuitively measure an aspect of locality of CDCL proofs, in the sense that proofs which spans fewer variables can be seen as more local. Theorem 5 indicates that this set of spanning variables constitutes an (not necessarily minimal) LSR backdoor witnessing the proof of unsatisfiability for the formula.

Table 1. Locality of proofs through the lens of LSR backdoors. Values are normalized by the number of variables in each instance. Standard deviations are given in parentheses.

Heuristic		Agile			
Branching	Restart	LSR	All Decisions	Time (s)	Conflicts
LRB	Luby	0.21 (0.08)	0.38 (0.10)	0.45 (1.81)	13392.56 (48874.86)
	Always	0.15 (0.05)	0.49 (0.13)	0.31 (1.21)	9320.84 (31384.02)
	Never	0.34 (0.15)	0.39 (0.11)	1.29 (4.25)	30450.00 (91745.34)
VSIDS	Luby	0.23 (0.09)	0.40 (0.11)	0.18 (0.73)	7783.51 (25684.43)
	Always	0.14 (0.05)	0.45 (0.12)	0.16 (0.54)	6836.73 (20516.48)
	Never	0.32 (0.14)	0.37 (0.10)	1.10 (5.23)	32665.29 (127482.25)
Random	Luby	0.76 (0.25)	0.94 (0.11)	3.12 (9.17)	52963.82 (138268.52)
	Always	0.29 (0.11)	0.96 (0.09)	2.96 (9.08)	51113.82 (139041.44)
	Never	0.75 (0.24)	0.93 (0.12)	7.63 (13.86)	107275.56 (174531.05)

Table 2. Results for application instances.

Heuristic		Application			
Branching	Restart	LSR	All Decisions	Time (s)	Conflicts
LRB	Luby	0.62 (0.35)	0.61 (0.35)	526.64 (931.93)	1728644.37 (3476414.24)
	Always	0.59 (0.35)	0.68 (0.33)	837.13 (1547.10)	2347710.94 (3935311.45)
	Never	0.62 (0.35)	0.60 (0.34)	682.98 (1148.91)	2544999.84 (5911701.24)

5.1 Computing LSR Backdoors Using the *LaSeR* Tool

In this Section, we discuss a tool called *LaSeR*, built on top of MapleSat [18], to compute the set of variables spanning the proof in each run. The tool runs MapleSat as normal, but maintains additional logging to compute the backdoor.

For every conflicting state, let C' denote the clause that will be learned through conflict analysis. We let $R_{C'}$ be the set of clauses on the conflict side of the implication graph used to derive C' where $R_{C'}^* = R_{C'} \cup \bigcup_{C \in R_{C'}} R_C^*$ recursively defines the set of clauses needed to derive C' (where $R_{original_clause}^* = \emptyset$). For every learnt clause we define $D_{C'}^* = vars(C') \cup \bigcup_{C \in R_{C'}^*} D_C^*$, where $D_{original_clause}^* = \emptyset$, as the set of variables in the clause itself as well as any learnt clause used in the derivation of the clause (recursively). Intuitively, $D_{C'}^*$ is a sufficient set of *dependency variables*, such that a fresh SAT solver can absorb C' by only branching on variables in the set. For a set of clauses Δ , we let $R_{\Delta}^* = \bigcup_{C \in \Delta} R_C^*$ and $D_{\Delta}^* = \bigcup_{C \in \Delta} D_C^*$. Let Δ_{\perp} be the set of clauses involved in the final conflict, i.e., when the solver is about to derive UNSAT. Our invariant ensures that $\bigcup_{C \in \Delta_{\perp}} D_C^*$ constitutes a [not-necessarily minimal] LSR backdoor.

5.2 Empirical Results

We compare several solving and restart heuristics through the lens of this spanning variables metric, which we will refer to simply as the LSR backdoor of the proof (LSR in Tables 1 and 2). Our experiments are conducted over all unsatisfiable instances from both the Application track of the SAT competition from 2009–2014, as well as the Agile 2016 instances. The Agile track contains smaller instances generated from the quantifier-free bit-vector formulas derived from the SAGE whitebox fuzzer [12]. The timeout in the SAT competition for these instances was only 60 s.

We consider three restart policies: (1) the Luby heuristic; (2) restarting after every conflict (“Always”); and (3) never restarting. For the Agile instances, we considered three branching heuristics: LRB [18], VSIDS [21], and random branching (with phase-saving polarity selection), thus totaling 9 solver configurations in combination with the restart policies. For the Application instances, we did not include VSIDS or random branching in our experiments due to the cost of computation and to avoid the random branching heuristic greatly limiting our set of usable instances. We allotted 10,000 s for each Application instance, and 300 s for each Agile instance. Experiments were run on an Azure cluster, where each node contained two 3.1 GHz processors and 14 GB of RAM. Each experiment was limited to 6 GB. We only include instances where we could compute data for all heuristics being considered, in total, 1168 Agile instances, and 81 Application instances. For each instance, the size of the LSR backdoor is normalized by the total number of variables.

Tables 1 and 2 depict the results. On average, the always-restart policy seems to produce more local proofs than the other policies, regardless of the branching heuristic. This may provide further explanation as to why restarts are useful in practice, particularly on unsatisfiable instances. Interestingly, the always-restart

policy ends up requiring the most time and conflicts to solve the Application instances; this may indicate that the usefulness of this locality is dependent on the types of instances. We also wish to emphasize that although the average LSR ratio is only 0.03 smaller for always-restart than the other policies on Application instances, this amounts to approximately 390 variables on average.

Finally, we compare our above approach to computing LSR backdoors to the previously proposed “All Decisions” approach to computing LS backdoors. In [9], the authors compute LS backdoors by running a randomized non-restarting CDCL solver to completion and recording the set of all variables branched upon during search. This set constitutes an LS backdoor. The process is repeated many times to try to find small backdoors. Due to the number of instances we consider, we only use one run of the solver for each heuristic being considered. Tables 1 and 2 compare our above LSR approach to the set of all decision variables (computed on the same solver run with restarts). Since many clauses learned during search are not useful for the proof, the all-decisions approach records many unnecessary decisions that are ultimately not useful. The result does not hold on many types of crafted instances however, particularly when the formula is designed to intrinsically require proofs spanning many variables. Nonetheless, our approach seems to work for certain classes of instances found in industrial settings.

6 Conclusions and Future Work

In this paper, we explored several important questions within the context of the broad research program of trying to understand why Boolean SAT solvers are so efficient for industrial instances obtained from verification, program analysis, testing, and synthesis. All the questions we explored relate to restart techniques in SAT solvers, new kinds of restart-aware backdoors we introduced, and perhaps most importantly a characterization of the properties (e.g., locality) of proofs produced by SAT solvers through the lens of such backdoors.

Specifically, we introduced the notion of LSR backdoors, and demonstrated an exponential separation from LS backdoors (which in turn were shown to be exponentially smaller than strong backdoors for certain class of instances in previous work by Dilkina et al.). A takeaway of this result is that clause learning together with restarts is capable of exploring the search space in ways not possible with clause learning alone.

We further showed that LSR-UIP backdoors may be exponentially smaller than LSR-DL backdoors. The order in which the search space is explored is crucial when branching over both LS and LSR backdoors, and we demonstrated several issues that may arise during the search. Empirically, we demonstrated that rapid restart strategies tend to produce significantly more local proofs than other strategies on industrial instances.

Going forward, we would like to refine our empirical results further by comparing uniform restart policies (e.g., restarting every k conflicts or geometric restarts) that are less “extreme” against the always-restart-at-conflicts policy.

We also plan to refine our notion of locality in proofs by considering the structure of (e.g., the variable-incidence graph) of Boolean formulas. Another line of future work is to answer the big open question, namely, are CDCL SAT solvers with restarts a more powerful proof system than CDCL without restarts?

References

1. Atserias, A., Fichte, J.K., Thurley, M.: Clause-learning algorithms with many restarts and bounded-width resolution. *J. Artif. Intell. Res.* **40**, 353–373 (2011)
2. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res.* **22**, 319–351 (2004)
3. Beame, P., Kautz, H., Sabharwal, A.: Understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res.* **22**, 319–351 (2014)
4. Beame, P., Sabharwal, A.: Non-restarting sat solvers with simple preprocessing can efficiently simulate resolution. In: *AAAI Conference on Artificial Intelligence*, pp. 2608–2615. AAAI Press (2014)
5. Biere, A., Fröhlich, A.: Evaluating CDCL restart schemes. In: *Pragmatics of SAT Workshop* (2015)
6. Biere, A., Heule, M., van Maaren, H.: *Handbook of satisfiability*, vol. 185. IOS Press (2009)
7. Buss, S.R., Hoffmann, J., Johannsen, J.: Resolution trees with lemmas: resolution refinements that characterize DLL algorithms with clause learning. *arXiv preprint arXiv:0811.1075* (2008)
8. Dilkina, B., Gomes, C.P., Malitsky, Y., Sabharwal, A., Sellmann, M.: Backdoors to combinatorial optimization: feasibility and optimality. In: van Hoeve, W.-J., Hooker, J.N. (eds.) *CPAIOR 2009. LNCS*, vol. 5547, pp. 56–70. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01929-6_6
9. Dilkina, B., Gomes, C.P., Sabharwal, A.: Backdoors in the context of learning. In: Kullmann, O. (ed.) *SAT 2009. LNCS*, vol. 5584, pp. 73–79. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_9
10. Dilkina, B., Gomes, C.P., Sabharwal, A.: Tradeoffs in the complexity of backdoors to satisfiability: dynamic sub-solvers and learning during search. *Ann. Math. Artif. Intell.* **70**(4), 399–431 (2014). <https://doi.org/10.1007/s10472-014-9407-9>
11. Ganian, R., Ramanujan, M.S., Szeider, S.: Backdoor treewidth for SAT. In: Gaspers, S., Walsh, T. (eds.) *SAT 2017. LNCS*, vol. 10491, pp. 20–37. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_2
12. Godefroid, P., Levin, M.Y., Molnar, D.A., et al.: Automated whitebox fuzz testing. In: *Network and Distributed System Security Symposium*, pp. 151–166. Internet Society (2008)
13. Haim, S., Heule, M.: Towards ultra rapid restarts. *arXiv preprint arXiv:1402.4413* (2014)
14. Hertel, P., Bacchus, F., Pitassi, T., Van Gelder, A.: Clause learning can effectively p-simulate general propositional resolution. In: *AAAI Conference on Artificial Intelligence*, pp. 283–290. AAAI Press (2008)
15. Huang, J.: The effect of restarts on the efficiency of clause learning. In: *International Joint Conference on Artificial Intelligence*, pp. 2318–2323. AAAI Press (2007)
16. Kilby, P., Slaney, J., Thiébaux, S., Walsh, T.: Backbones and backdoors in satisfiability. In: *AAAI Conference on Artificial Intelligence*, pp. 1368–1373. AAAI Press (2005)

17. Li, Z., van Beek, P.: Finding small backdoors in SAT instances. In: Butz, C., Lingras, P. (eds.) AI 2011. LNCS (LNAI), vol. 6657, pp. 269–280. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21043-3_33
18. Liang, J.H., Ganesh, V., Poupard, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 123–140. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_9
19. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. *Inf. Process. Lett.* **47**(4), 173–180 (1993)
20. Marques-Silva, J.P., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. *IEEE Trans. Comput.* **48**(5), 506–521 (1999)
21. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Design Automation Conference, pp. 530–535. ACM (2001)
22. Nadel, A., Ryvchin, V.: Chronological backtracking. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) SAT 2018. LNCS, vol. 10929, pp. 111–121. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_7
23. Pipatsrisawat, K., Darwiche, A.: A new clause learning scheme for efficient unsatisfiability proofs. In: AAAI Conference on Artificial Intelligence, pp. 1481–1484 (2008)
24. Pipatsrisawat, K., Darwiche, A.: On the power of clause-learning SAT solvers with restarts. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 654–668. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_51
25. Pipatsrisawat, K., Darwiche, A.: On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.* **175**, 512–525 (2011). <https://doi.org/10.1016/j.artint.2010.10.002>
26. Ruan, Y., Kautz, H., Horvitz, E.: The backdoor key: a path to understanding problem hardness. In: AAAI Conference on Artificial Intelligence, pp. 124–130. AAAI Press (2004)
27. Samer, M., Szeider, S.: Backdoor trees. In: AAAI Conference on Artificial Intelligence, pp. 363–368. AAAI Press (2008)
28. Gelder, A.: Pool resolution and its relation to regular resolution and DPLL with clause learning. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 580–594. Springer, Heidelberg (2005). https://doi.org/10.1007/11591191_40
29. Williams, R., Gomes, C., Selman, B.: On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. In: International Conference on Theory and Applications of Satisfiability Testing, pp. 222–230. Springer (2003). <https://doi.org/10.1.1.128.5725>
30. Zhang, L., Madigan, C.F., Moskewicz, M.H., Malik, S.: Efficient conflict driven learning in a boolean satisfiability solver. In: Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, pp. 279–285. IEEE Press (2001)

Applications Track



Process Plant Layout Optimization: Equipment Allocation

Gleb Belov¹(✉), Tobias Czauderna¹, Maria Garcia de la Banda¹,
Matthias Klapperstueck¹, Ilankaikone Senthoooran¹, Mitch Smith²,
Michael Wybrow¹, and Mark Wallace¹

¹ Faculty of Information Technology, Monash University, Melbourne, Australia
{gleb.belov,tobias.czauderna,maria.garciadelabanda,matthias.klapperstueck,
ilankaikone.senthoooran,michael.wybrow,mark.wallace}@monash.edu

² Woodside Energy Ltd., Perth, Australia
mitch.smith@monash.edu

Abstract. Designing the layout of a chemical plant is a complex and important task. Its main objective is to find a most economical spatial arrangement of the equipment and associated pipes that satisfies construction, operation, maintenance and safety constraints. The problem is so complex it is still solved manually, taking multiple engineers many months (or even years) to complete. This paper provides (a) the most comprehensive model ever reported in the literature for spatially arranging the equipment, and (b) a Large Neighbourhood Search framework that enables complete solvers explore much larger neighbourhoods than previous approaches to this problem. The two contributions are part of a system being developed in collaboration with Woodside Energy Ltd. for arranging their Liquefied Natural Gas plants. The results are indeed so promising that Woodside are actively exploring its commercialisation.

1 Introduction

A chemical process plant produces chemicals by transforming or separating materials as they pass through different equipment via connecting pipes [10]. These plants are common in many industries, such as oil and gas, and are very costly to design, build and maintain, requiring multibillion-dollar budgets. When designing the layout of a new plant, the objective is to find a most economical spatial arrangement of the equipment and associated pipes that satisfies construction, operation, maintenance, and safety constraints.

High-quality layout can have a very significant impact on the cost of these plants. It can considerably reduce the cost of the pipes and associated support structures, which are known to take the largest share: up to 80% of the purchased equipment cost or 20% of the fixed-capital investment [12]. It also greatly reduces the total amount of space/volume needed, which is crucial for offshore plants. However, finding high-quality plant layouts is remarkably difficult due to the size of these plants and the complexity of the associated constraints. As a result,

layouts are still designed manually, taking multiple engineers many months (or even years) to complete. This process is inefficient, costly and the results may vary in quality, since they largely depend on the experience of the piping and layout engineers. For this reason, Woodside Energy Ltd. funded our project to explore the use of optimisation and visualisation technology in improving the current layout design process for their Liquefied Natural Gas plants.

Due to the complexity of this problem, most methods published in this area divide it into independently solved two phases: the first phase spatially places the equipment, while the second routes the connecting pipes. However, these methods (e.g., [6, 14, 17, 18]) are too simplistic to meet industry requirements and/or do not scale. Our work aims to produce a comprehensive solution that satisfies real-world needs. We reported a model for the pipe routing phase of the problem in [3]. This paper presents two further contributions. The first one is the most realistic model ever reported for solving the equipment allocation phase. Prior to this, the most complete approach [6] only considered non-overlapping constraints and the minimisation of approximate pipe lengths, elevation and footprint. In addition to these, our model handles constraints on equipment alignment, simplified maintenance access, and support structures. While vital to model the real problem, all this (particularly the maintenance constraints) considerably increases the complexity of the model and, thus, decreases the scalability of the approach for complete search methods. The second contribution helps in this regard: a Large Neighbourhood Search [16] (LNS) framework that uses a modified neighbourhood definition and warm-start for several complete solvers (via the MiniZinc [11] modelling language). The resulting framework is highly efficient and can explore larger neighbourhoods than the only previous LNS approach [18] for this problem.

The model and LNS framework are parts of a much larger system being developed with Woodside Energy Ltd. This system aims at transforming the way in which Woodside engineers approach plant-layout design, by allowing them to get a global view of the plant layout quickly, and easily compare different design decisions. As our experimental evaluation shows, our approach provides us with the solution quality and scalability required for this application. The results are so promising Woodside is actively exploring commercialisation of the system.

2 Literature Review

To solve the plant layout problem we are required to find 3D location coordinates for the equipment and connecting pipes within a plant's volume (referred to as the *container space*), in such a way as to minimise the total cost of the plant while, at the same time, ensuring its safety and correct functionality. For small problem instances, one can apply integrated approaches (e.g., [14]) that simultaneously place the equipment and route the pipes. For larger, more realistic instances, current integrated approaches do not scale. For example, [14] fails to find any solution for plants with just 10 pieces of equipment and 8 pipes.

As a result, methods to solve the plant layout problem often divide it into two phases [6]. The first one finds the position and orientation of each piece of equipment, minimising an approximate total cost for the plant. The second phase looks for an optimal pipe routing to connect the already positioned equipment. There has been a significant amount of research devoted to the first phase (e.g., [6, 14, 17, 18]), which is the focus of our paper. However, most methods are too simplistic, concentrating mainly on (often 2D) non-overlapping constraints. No method we know of considers maintenance access constraints, alignment constraints, or the cost of (not) using provided support structures for elevated equipment.

In addition, scalability is a difficult issue due to the high combinatorial nature of the problem, given by the packing requirements. Approaches that use complete methods (e.g., [6, 14]) quickly become prohibitive as the number of objects grow. Others achieve scalability by giving up on completeness and, for example, iteratively constructing layouts that extend an initial, incomplete plant with more and more equipment [17], or using a quadratic force minimisation model of the problem that tries to compress an initial (complete) layout by minimizing “forces” between connected pieces of equipment [9]. As near-optimality is important for Woodside, we looked at LNS solutions to the problem and only found [18], which uses LNS to solve a simple 2D version of the problem.

The closest commercial product for plant layout is ASD Global’s OptiPlant [1], which performs automated pipe routing and can generate an initial 3D plant layout from an equipment list. However, this product only deals with phase two (pipe routing), requiring users to specify equipment positions.

3 Full System and Key Role of Constraint Programming

As mentioned before, this paper describes components of a much larger system being developed in collaboration with Woodside Energy Ltd. The system has a 2D visual interface (see Fig. 1) for users to specify the input data needed for the optimisation process. This information is stored in a JSON file and, when users press the “Launch Optimisation” button, is sent to a C++ program that generates the MiniZinc model for the first phase (described in this paper). MiniZinc compiles this model to the target solver and executes it to obtain the final positions and rotations of the equipment, writing these as additional entries in JSON format. Then, the C++ program uses the stored solution to generate a MiniZinc model for the second phase (described in [3]), compiles this to the target solver and executes it to obtain the routing of every pipe, again storing this in JSON format.

Once the equipment is placed and the pipes routed, a Python extension to FreeCAD generates a 3D model of the solution with structural information for navigating the equipment and pipes in the plant. An interactive 3D visualisation of the 3D model (see Fig. 2) enables engineers to explore the produced layout collaboratively, evaluate and validate the proposed solution in a familiar way, and compare different solutions. Now that our LNS framework can produce high quality solutions in a relatively short amount of time (see Sect. 6), the aim is to support engineers in interactively re-optimising a given solution.

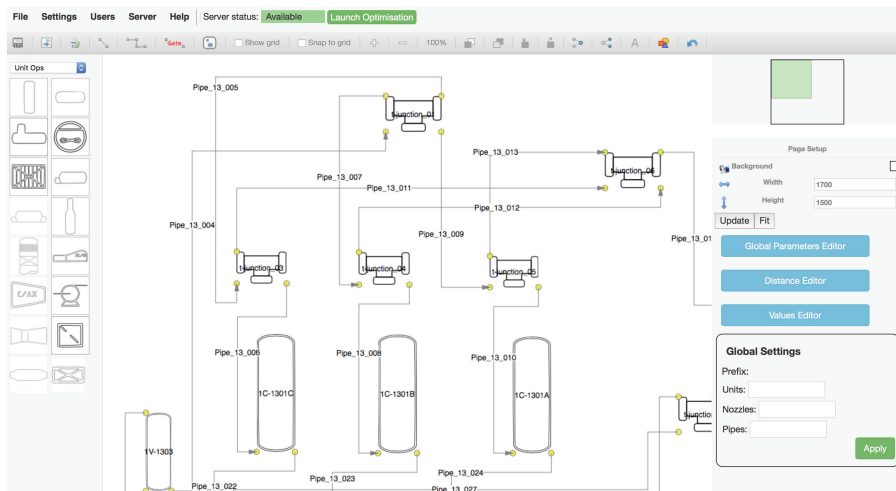


Fig. 1. Process Editor interface for plant engineers to specify all input data.

The use of a constraint programming modelling language (MiniZinc) has been critical to the success of our project for three main reasons. First, it has allowed us to quickly modify the models and try different modelling alternatives. This has been invaluable as our lack of knowledge regarding chemical plants often created issues that were only resolved by constantly exploring model changes with Woodside’s engineers. Further model changes occurred as Woodside’s engineers became more familiar with the possibilities offered by optimisation technology and we could increase the amount and quality of the constraints. Second, it has allowed us to easily compare the efficiency of different solvers. This has been particularly important since, as the models evolved, the efficiency of the solvers varied greatly. Being able to quickly determine which solver is faster for the current model allows us to give Woodside’s engineers the best efficiency. Finally, it has allowed us to quickly build a system that far exceeds the capabilities of those currently available both in the literature and commercially. As a result, Woodside engineers can now obtain one or more near-optimal layouts for a plant significantly faster than before, and can already explore the consequences of applying simple modifications to the data (e.g., costs and safety distances).

4 An Optimization Model for Equipment Allocation

The model provided in this paper describes the first phase of the process plant layout problem. That is, determining the 3D position coordinates and the orientation of the given equipment within a given *container space*, that (a) satisfy safety, maintenance and alignment constraints and (b) minimise the costs of the land needed (the *footprint*), the supporting equipment, and the connecting pipes (see Fig. 2 for a final solution to our small benchmark).

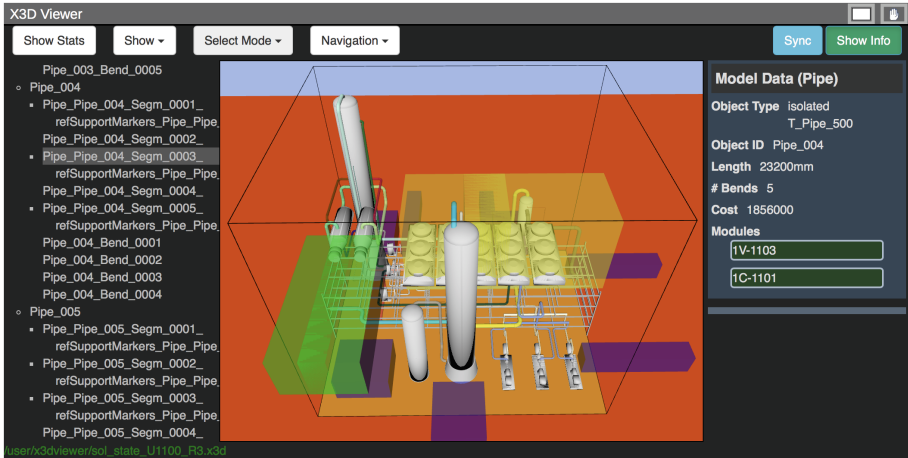


Fig. 2. Visualisation of the final layout for Unit 1000 as a 3D model. Coloured boxes indicate different kinds of maintenance access zones (e.g., purple are truck access). (Color figure online)

Our model considers the container space as a discretized cuboid with front-left-bottom (FLB) corner at coordinates $(0, 0, 0)$, and a (maximum) user-defined length, width and height corresponding to axes x, y and z , respectively, where $x \times y$ defines the footprint. The exact shape of each piece of *equipment* is abstracted by a bounding *box*, also represented by the position of its FLB corner and its user-defined (fixed) length, width and height. Boxes are positioned into one of four different *horizontal rotations* around the vertical axis: $0^\circ, 90^\circ, 180^\circ$ and 270° .

Each *nozzle* connecting a pipe with diameter d to equipment b at a position with centre $\bar{p} \in \mathbb{Z}^3$ relative to the equipment box's FLB corner, is represented as a point located $3d$ units perpendicularly away from the surface containing \bar{p} . This roughly corresponds to the usual bend position of a nozzle segment. Every pipe has a user-defined cost factor (per length of unit) that determines its cost. In this phase, pipe length is approximated by the Manhattan distance obtained from the position of its two connecting nozzles. The splitting of a pipe into two (or merging of two pipes into one) is modelled by a special rotatable *t-junction* box that inherits the pipe properties (e.g., size and safety distances).

User-defined *safety distances* between all equipment must be enforced. Default safety distances are specified between equipment *classes* (each equipment belongs to a class), and can be modified by providing a specific safety distance between any two. Note that safety distances are *directed*: that from A to B, where A is before B w.r.t. their projection on axis c , might be different than that from B to A. This might be useful for expressing vertical relative positions, for example, between fin-fans and other equipment, as well as horizontal positions from high-risk equipment in areas with well-defined wind directions.

Some equipment (such as the pipe *rack*) can support other equipment without further capital costs. Thus, equipment might be positioned on the ground (at no cost), on some other equipment (again at no cost), or “in the air”, representing the fact that a supporting structure must be built. The cost of this structure is approximated using the cost (per height unit) associated to each piece of equipment (see *bph* below). Some equipment is allowed to protrude by a given amount (its *support margin*) over the sides of its support box. If the amount is negative, the box should be that far inside the support box’s sides.

Some equipment should be located at a certain minimal level above the baseline of another. For example, a vessel might need to be a certain distance above a pump, if its flow into the pump needs to satisfy the *Net Positive Suction Head (NPSH)* regulation. We formulate the corresponding constraint in terms of minimal height differences. Note that they cannot be expressed by safety distances, as the size of the lower object might be larger than the elevation distance.

Some equipment has particular *maintenance access* requirements, such as the need to be accessible from above/below, or by a truck (requiring a big empty space to be attached to it and accessible from the road). The former is modelled by a constraint that ensures no other equipment is positioned above or below. The latter is modelled using additional (slave) boxes that satisfy the requirements in terms of size and location relative to the (master) equipment boxes. Currently, our model handles three types of relative locations:

- *Rigid (fixed) attachment*: slave is at a specified position relative to the master and the whole combination rotates together. Examples: access must be provided to a particular side of the equipment.
- *Rotatable attachment*: as before, but slave can rotate around the master. Example: access must be provided to any side of the equipment.
- *Multi-zone disjunctive attachment*: slave located in one of the given zones and orientated as the master. Example: aligning a pipe header to a pump group can be modelled by attaching one or several zone boxes to the pump group and requiring the header to be in one of these zones.

The rest of the section provides a summary of the parameters (input and derived data), variables, constraints and objective function used in our model.

4.1 Input and Derived Data

Input Index Sets.

- $\mathcal{B} = \{1, \dots, N\mathcal{B}\}$: set of boxes that need to be allocated.
- $\mathcal{P} = \{1, \dots, N\mathcal{P}\}$: set of pipes that connect the equipment.
- $\mathcal{MAZ} \subset \mathcal{B}$: subset of boxes that are maintenance access zones.
- $\mathcal{B}^{Supp} \subseteq \mathcal{B} \cup \{N\mathcal{B} + 1\}$: subset of boxes that can support other boxes without further construction costs ($N\mathcal{B} + 1$ represents the ground).
- $\mathcal{ATTZN} \subset \mathcal{B}$: subset of multi-zone attachment zones.

We currently assume there are two nozzles per pipe, and use input set \mathcal{P} to construct the set $\mathcal{NZ} = \{(p, k) | p \in \mathcal{P}, k \in \{1, 2\}\}$ of nozzles. We also use index set $\mathcal{OH} = \{1, 2, 3, 4\}$ to represent rotations $\{0^\circ, 90^\circ, 180^\circ, 270^\circ\}$, respectively.

Input Data.

- $\overline{W}_i^0 \in \mathbb{Z}^{\times 3}$: x , y and z sizes of box $i \in \mathcal{B}$ while in horizontal rotation 0° .
- $\overline{XFLB}_i^{\text{LB}}, \overline{XBRT}_i^{\text{UB}} \in \mathbb{Z}^3$: lower bound of the FLB corner of box $i \in \mathcal{B}$, and upper bound of its back-right-top (BRT) corner, respectively. Often determined by the container space.
- $\mathcal{BOH}_i \subseteq \mathcal{OH}$: set of allowed horizontal orientations for box $i \in \mathcal{B}$.
- $bsd_{i,j}^H, bsd_{i,j}^V \in \mathbb{Z}$: horizontal and vertical directed safety distances, respectively, between boxes $i, j \in \mathcal{B}$.
- \mathcal{NPSH} : set of tuples (i, j, h) , where $i, j \in \mathcal{B}$ and $h \in \mathbb{Z}$, indicating that the base of box j must be above the base of i by at least h units.
- $suppMrg_i \in \mathbb{Z}$: support margins for box $i \in \mathcal{B}$.
- $bph_i \in \mathbb{Z}$: height support penalty for box $i \in \mathcal{B}$, in \$ per height unit.
- $batt_i \in \{0, 1, 2, 3\}$: attachment type of box $i \in \mathcal{B}$, where 0 indicates none, 1 rigid, 2 rotatable, and 3 multi-zone.
- $batm_i \in \mathcal{B}$: attachment master box of slave box $i \in \mathcal{B}$ ($batm_i = i$ if $batt_i = 0$).
- $\overline{batp}_i \in \mathbb{Z}^3$: attachment point of box $i \in \mathcal{B}$ relative to its master's FLB corner in orientation 0° , for rigid and rotatable attachment types.
- $\mathcal{Batz}_i \subset \mathcal{B}$: set of possible location zones (i.e., boxes, typically from \mathcal{ATTZN}) for the FLB corners of slave box $i \in \mathcal{B}$ with multi-zone attachment.
- $nzBox_i \in \mathcal{B}$: master box of nozzle $i \in \mathcal{NZ}$.
- $\overline{nzPoz}_i^0 \in \mathbb{Z}^3$: position of nozzle $i \in \mathcal{NZ}$ in its master box (orientation 0°).
- $plc_i \in \mathbb{Z}$: cost factor for pipe $i \in \mathcal{P}$, in \$ per length unit.
- $fpc_c \in \mathbb{Z}, c \in \{x, y\}$: cost factor for perimeter length and width, respectively, in \$ per length unit.

4.2 Decision Variables

A solution to an instance of our model is expressed in terms of the values of the following decision variables (the first two groups functionally define all others):

- $r_i^F \in \mathcal{BOH}_i$: final orientations of each box $i \in \mathcal{B}$.
- $\overline{XFLB}_i, \overline{XBRT}_i = \overline{XFLB}_i + \overline{W} \in \mathbb{Z}^3$ positions of the FLB and BRT corners of each box $i \in \mathcal{B}$, respectively.
- $\overline{W}_i^F \in \mathbb{Z}^3$: final sizes of each box $i \in \mathcal{B}$ according to its final rotation.
- $relPos_{ijc} \in \{0, 1\}$ relative position variable (directed separation flag) for pair of boxes $i, j \in \mathcal{B}$ along coordinate axis $c \in \{x, y, z\}$. It holds: $relPos_{ijc} = 1$ iff box j is after i in projection on axis c , obeying their safety distance.
- $suppIdx_i \in \mathcal{B}^{supp}$: supporting box of each box $i \in \mathcal{B}$ s.t., $bph_i > 0$.
- $suppCost_i \in \mathbb{Z}$: computed support cost for each box $i \in \mathcal{B}$.
- $\overline{nzPos}_i \in \mathbb{Z}^3$: absolute position of each nozzle $i \in \mathcal{NZ}$.
- $pLen_i \in \mathbb{Z}^3$: approximated length along each axis, for each pipe $i \in \mathcal{P}$.
- $fpc_c \in \mathbb{Z}, c \in \{x, y\}$: footprint length and width, respectively.
- $obj \in \mathbb{Z}$: objective function value.

4.3 Functions

The constraints in our model use the following five functions. Function $\text{getBoxSafety} : \mathcal{B} \times \mathcal{B} \times \{x, y, z\} \rightarrow \mathbb{Z}$ returns the minimal positive separation from box $i \in \mathcal{B}$ to box $j \in \mathcal{B}$ along axis c :

$$\text{getBoxSafety}(i, j, c) = \begin{cases} \text{bsd}_{i,j}^H, & c \text{ is } x \text{ or } y \\ \text{bsd}_{i,j}^V, & c \text{ is } z \end{cases} \quad (1)$$

Function $\text{hFindFLBCorner} : \mathbb{Z}^3 \times \mathcal{OH} \rightarrow \mathbb{Z}^3$ returns the new position of the FLB corner of a box with sizes $\overline{W} = (W_x, W_y, W_z)$, once it is rotated according to r . It uses the *element* constraint [2] to select a matrix column using r as index:

$$\text{hFindFLBCorner}(\overline{W}, r) = \begin{pmatrix} 0 & W_y & W_z & 0 \\ 0 & 0 & W_y & W_x \\ 0 & 0 & 0 & 0 \end{pmatrix}_{.r} \quad (2)$$

Note that the above function and several of the constraints defined later (e.g., (9) and (13b)) are non-linear and, thus, not directly supported by MIP solvers. The MIP interface of MiniZinc [4] handles their MIP decomposition.

Function $\text{hRotateB} : \mathbb{Z}^3 \times \mathcal{OH} \rightarrow \mathbb{Z}^3$ returns the new sizes of a box with sizes $\overline{W} = (W_x, W_y, W_z)$, rotated according to r .

$$\text{hRotateB}(\overline{W}, r) = \begin{pmatrix} W_x & W_y & W_x & W_y \\ W_y & W_x & W_y & W_x \\ W_z & W_z & W_z & W_z \end{pmatrix}_{.r} \quad (3)$$

Function $\text{hRotateBWB} : \mathbb{Z}^3 \times \mathbb{Z}^3 \times \mathbb{Z}^3 \times \mathcal{OH} \rightarrow \mathbb{Z}^3$ receives the sizes \overline{W}_s and \overline{W}_m of boxes s and m , respectively, the point \overline{P} where the FLB corner of s is rigidly attached to m (relative to m 's FLB corner, which is always $(0,0,0)$) in their default orientation, and rotation r . Returns the relative position of the FLB corner of s to that of m , once both are rotated by r around m 's centre.

$$\text{hRotateBWB}(\overline{W}_s, \overline{W}_m, \overline{P}, r) = \begin{pmatrix} P_x & W_{my} - P_y - W_{sy} & W_{mx} - P_x - W_{sx} & P_y \\ P_y & P_x & W_{my} - P_y - W_{sy} & W_{mx} - P_x - W_{sx} \\ P_z & P_z & P_z & P_z \end{pmatrix}_{.r} \quad (4)$$

Function $\text{hRotateBAB} : \mathbb{Z}^3 \times \mathbb{Z}^3 \times \mathbb{Z}^3 \times \mathcal{OH} \times \mathcal{OH} \rightarrow \mathbb{Z}^3$ receives the sizes \overline{W}_s and \overline{W}_m of boxes s and m , respectively, the point \overline{P} where the FLB corner of s is attached to m (relative to m 's FLB corner, which is always $(0,0,0)$) in their default orientation, and two rotations r_s and r_m . For efficiency, it returns an approximation of the relative position of the FLB corner of s relative to m , once s and m are rotated according to r_s and r_m , respectively, around the centre of m . The approximation is done by “shrinking” the master box to a square footprint,

which is acceptable as long as we attach spacious MAZ.

$$\begin{aligned} \text{hRotateBAB}(\overline{W}_s, \overline{W}_m, \overline{P}, r_s, r_m) &= \begin{pmatrix} \lfloor W_{mx}^m/2 \rfloor - \lceil w/2 \rceil \\ \lfloor W_{my}^m/2 \rfloor - \lceil w/2 \rceil \\ +0 \end{pmatrix} + \\ \text{hRotateBWB} \left(\overline{W}_s, \begin{pmatrix} w \\ w \\ W_{sz} \end{pmatrix}, \overline{P} + \begin{pmatrix} \lceil w/2 \rceil - \lfloor W_{mx}/2 \rfloor \\ \lceil w/2 \rceil - \lfloor W_{my}/2 \rfloor \\ 0 \end{pmatrix}, r_s \right) & \quad (5) \end{aligned}$$

where $w = \min\{W_{mx}, W_{my}\}$ is the minimum of the master's horizontal sizes and $\overline{W}_m^r = \text{hRotateB}(\overline{W}_m, r_m)$ are the sizes of the master rotated according to r_m .

Function $\text{hRotatePWB} : \mathbb{Z}^3 \times \mathbb{Z}^3 \times \mathcal{OH} \rightarrow \mathbb{Z}^3$ receives a rotation r and a point \overline{P} rigidly attached to a box with sizes \overline{W} . Returns the relative position of \overline{P} to the box's FLB corner, once both are rotated by r around the box's centre.

$$\text{hRotatePWB}(\overline{P}, \overline{W}, r) = \text{hRotateBWB}(\overline{0}, \overline{W}, \overline{P}, r) \quad (6)$$

4.4 Constraints and Objective Function

Box sizes: can be obtained from the original sizes and the final rotations:

$$\overline{W}_b^F = \text{hRotateB}(\overline{W}_b^0, r_b^F), \quad b \in \mathcal{B} \quad (7)$$

Box position: should satisfy the given bounds.

$$\overline{XFLB}_b^{\text{LB}} \leq \overline{XFLB}_b, \quad \overline{XRBT}_b \leq \overline{XRBT}_b^{\text{UB}}, \quad b \in \mathcal{B} \quad (8)$$

Box disjointness: only needs to be enforced between boxes that are not maintenance access zones (which are allowed to overlap), and are not attached to each other. If we had equal safety distances, we could have enforced disjointness by using the `diffnk` global constraint [2]. Since this is not the case, we enforce the disjointness of boxes $i, j \in \mathcal{B}$ similarly to [6], as follows. First, we reify the existence of the appropriate safety distance from i to j in each axis c as:

$$\begin{aligned} \text{relPos}_{ijc} = 1 &\leftrightarrow \\ \left\{ \begin{array}{l} \text{True}, \\ \overline{XBRT}_{ic} + \text{getBoxSafety}(i, j, c) \leq \overline{XFLB}_{jc}, \end{array} \right. & \begin{array}{l} i = j \vee \{i, j\} \subseteq \text{MAZ} \\ \vee i \text{ and } j \text{ are attached} \\ \vee i \in \text{ATTZN} \vee j \in \text{ATTZN} \\ \text{otherwise,} \end{array} \\ & \quad i, j \in \mathcal{B}, \quad c \in \{x, y, z\} \end{aligned} \quad (9)$$

Then, for each pair of boxes $i < j \in \mathcal{B}$, we demand the existence of such safety distance in at least one coordinate direction, positive or negative:

$$\bigvee_{c=1}^3 (\text{relPos}_{ijc} \vee \text{relPos}_{jic}), \quad i < j \in \mathcal{B} \quad (10)$$

This allows us to easily model the “none above/below” constraints by providing big enough vertical separations (ensuring they do not fit above each other).

Minimal Height Separation Constraints: they demand the base of box j to be above the base of box i by at least h units:

$$\overline{XFLB}_{iz} + h \leq \overline{XFLB}_{jz}, \quad (i, j, h) \in \mathcal{NPSH} \quad (11)$$

Support Constraints: Box $i \in \mathcal{B}$ with positive height support cost ($bph_i > 0$) is considered as being supported by another box $j \in \mathcal{B}^{Supp} \setminus \{N^{\mathcal{B}} + 1\}$ (not the ground) if i 's “core footprint” is contained in the footprint of j :

$$\begin{aligned} suppIdx_i = j &\leftrightarrow \\ \{ \overline{XFLB}_{ic} + suppMrg_i \geq \overline{XFLB}_{jc} \wedge \overline{XBR}_{ic} - suppMrg_i \leq \overline{XBR}_{jc} \}, \\ c \in \{x, y\}, i \in \mathcal{B}, bph_i > 0, j \in \mathcal{B}^{Supp} \setminus \{N^{\mathcal{B}} + 1\} \end{aligned} \quad (12)$$

The chosen form of the constraint implies that the support objects cannot be stacked if they already support something (as $suppIdx_i$ can only have one value). For efficiency, the domain of $suppIdx_i$ should be a priori reduced if possible. Finally, the support costs penalize being higher than the support object:

$$suppCost_i \geq 0, \quad (13a)$$

$$suppCost_i \geq bph_i (\overline{XFLB}_{iz} - (\overline{XFLB}_{.z} ++ (0))_{suppIdx_i}), \quad i \in \mathcal{B} : bph_i > 0 \quad (13b)$$

which assumes the ground at level 0. This allows the modelling of objects that overlap with their supports, as it is the case with the equipment placed in racks.

Box Attachment and MAZ: For a rigidly attached box (attachment type $batt_b = 1$), its FLB corner is computed from that of the master and the position of the attachment point, once rigidly rotated with the master to its final position.

$$\overline{XFLB}_b = \overline{XFLB}_{batm_b} + hRotateBWB(\overline{W}_b^0, \overline{W}_{batm_b}^0, \overline{batp}_b, r_b^F), \quad \forall b : batt_b = 1 \quad (14a)$$

Moreover its final orientation must be equal to that of its master $batm_b$:

$$r_b^F = r_{batm_b}^F, \quad b \in \mathcal{B} : batt_b = 1 \quad (14b)$$

For rotatable, attached boxes ($batt_b = 2$), where the master has a square footprint, the modelling is the same as (14a). For rotation around a non-square-footprint master, we only provide an approximation (see `hRotateBAB`):

$$\begin{aligned} \overline{X}_b = \overline{X}_{batm_b} + hRotateBAB(\overline{W}_b^0, \overline{W}_{batm_b}^0, \overline{batp}_b, r_b^F, r_{batm_b}^F), \\ b : batt_b = 2, \overline{W}_{batm_b,1}^0 \neq \overline{W}_{batm_b,2}^0 \end{aligned} \quad (15)$$

For multi-zone attachment ($batt_b = 3$), we require the slave's FLB corner to be in one of the specified zones (boxes in $\mathcal{B}atz_b$) and the rotation to be the same as that of the master ($batm_b$). This translates into the following system:

$$r_b^F = r_{batm_b}^F, \quad (16a)$$

$$\exists i \in \mathcal{B}atz_b : \overline{XFLB}_b \in [\overline{XFLB}'_i, \overline{XFLB}'_i + \overline{W}_i], \quad b : batt_b = 3 \quad (16b)$$

where

$$\overline{XFLB}'_i = \overline{XFLB}_i + \text{hFindFLBCorner}(\overline{W}_b^0, r_b^F), \quad i \in \mathcal{B}atz_b \quad (16c)$$

are the location zones' origins corrected for the slave's rotation.

Pipe Symmetry. A set of pipes $\mathcal{S} \subset \mathcal{P}$ might need to be symmetric, e.g., due to restrictions on the associated equipment. Our phase one model approximates pipe symmetry by demanding the nozzle distances \overline{pLen}_p of all pipes $p \in \mathcal{S}$ to be equal. Actual symmetry is enforced during pipe routing in the second phase.

Pipe Cost Approximation: uses the nozzle positions, which are computed using its master box's position and orientation:

$$\begin{aligned} \overline{nzPos}_{(p,i)} &= \overline{XFLB}_{nzBox_{(p,i)}} + \\ &\text{hRotatePWB}(\overline{nzPos}_{(p,i)}^0, \overline{W}_{nzBox_{(p,i)}}^0, r_{nzBox_{(p,i)}}^F), \quad p \in \mathcal{P}, \quad i \in \{1, 2\} \end{aligned} \quad (17)$$

This allows us to compute the pipe end differences and their absolute values:

$$\overline{pLen}_p = |\overline{nzPos}_{(p,2)} - \overline{nzPos}_{(p,1)}|, \quad p \in \mathcal{P} \quad (18)$$

Footprint Cost Approximation: similar to [6], we approximate footprint cost by penalizing perimeter length. We measure the footprint as including all physical boxes (i.e., no MAZ and zones) and, for each direction x or y , we only include boxes whose corresponding coordinate is not a priori fixed, as follows:

$$\begin{aligned} fps_c &= \max\{\overline{XFLB}_{bc} | b \in \mathcal{B}\mathcal{F}_c\} - \min\{\overline{XBRT}_{bc} | b \in \mathcal{B}\mathcal{F}_c\}, \\ \text{where } \mathcal{B}\mathcal{F}_c &= \{b \in \mathcal{B} | \overline{XFLB}_{bc} \neq \text{const}\} \setminus \mathcal{M}\mathcal{A}\mathcal{Z} \setminus \mathcal{A}\mathcal{T}\mathcal{T}\mathcal{Z}\mathcal{N}, \quad c \in \{x, y\} \end{aligned}$$

Objective Function: sum of the piping, footprint and support costs.

$$obj = \sum_{p \in \mathcal{P}, c \in \{x, y, z\}} plc_p \overline{pLen}_{pc} + \sum_{b \in \mathcal{B}} \text{suppCost}_b + \sum_{c \in \{x, y\}} fps_c \cdot fps_c \quad (19)$$

Optimisations: We reduce the domain of each suppIdx_i by removing boxes that are either too small to support box i , or (for a given neighbourhood) known to be located in a different area of the plant. We remove any non-overlapping constraints for pairs of boxes known not to overlap (for a given neighbourhood).

5 Overall Approach and Implementation

LNS is a meta-heuristic search method that, from an initial *seed* solution, iteratively relaxes part of the current solution and re-optimises the corresponding sub-problem obtaining a new solution. Our implementation uses a C++ program that, in each iteration, creates a new MiniZinc model for the neighbourhood, compiles it and executes it with a CP or MIP solver, using warm-starts when possible. The following describes these steps in more detail.

Constructing a Seed Solution: In order to have some control over the amount of time invested in finding a seed solution, we take two steps. First, we run the chosen solver until a feasible solution is found. Then we warm-start the same solver with that solution and run it with a given time limit.

Selecting the Boxes to Be Relaxed: When relaxing a solution, and given input parameters $L \leq U$, our LNS first selects a subset of boxes $\mathcal{B}^{N\mathcal{B}\mathcal{H}} \subset \mathcal{B}$ that will be relaxed (i.e., allowed to move freely) as follows: First, as long as we have not yet selected L boxes, a new box i is selected from set \mathcal{B} and added to set $\mathcal{B}^{N\mathcal{B}\mathcal{H}}$. For sequential LNS, i is a next biggest equipment box (starting from a new one for each $\mathcal{B}^{N\mathcal{B}\mathcal{H}}$); for random LNS, i is selected randomly. Next, all slave boxes attached to i and all boxes connected to i via pipes are added to $\mathcal{B}^{N\mathcal{B}\mathcal{H}}$, stopping if its cardinality reaches U . We repeat the process until the minimum number L of boxes is reached. This is different from [18] which constructs $\mathcal{B}^{N\mathcal{B}\mathcal{H}}$ by choosing the boxes based on various probabilistic selection schemes, from random to those considering the number of box connections, the cost of the box, or all boxes connected to the selected links.

Defining the Neighbourhood: Once $\mathcal{B}^{N\mathcal{B}\mathcal{H}}$ is selected, we define the *neighbourhood of the current solution* (subset of solutions to be explored in this iteration) by fixing the orientations of all boxes not in $\mathcal{B}^{N\mathcal{B}\mathcal{H}}$, and tightening their separation constraints (10). The tightening is done for all pairs of boxes $i, j \in \mathcal{B} \setminus \mathcal{B}^{N\mathcal{B}\mathcal{H}}$ such that $i < j$, by enforcing their relative position along one of the six directions where they were most separated in the last solution. That is, by setting $relPos_{abc} = 1$ for one of the six tuples in $\{(a, b, c) | \{a, b\} = \{i, j\}, c \in \{x, y, z\}\}$, one with the maximum value for $\overline{XFLB}_{bc} - \overline{XBRT}_{ac} - \text{getBoxSafety}(a, b, c)$. The other five relative position variables (and, consequently, their reification constraints (9)) for i, j are omitted, significantly simplifying the model and enhancing the solution space. This differs from the neighbourhood described in [18] for a 2D version of the problem, which also fixes the relative position variables of each pair of boxes not in $\mathcal{B}^{N\mathcal{B}\mathcal{H}}$, but does so for all four possible separation directions. By fixing only the largest-separation relation variable, our LNS obtains larger neighbourhoods and simpler models.

6 Evaluation

We have evaluated the practicality of our system by executing it as an 8-thread process on an Intel(R) Core(TM) i7-4771 CPU @ 3.90 GHz on two benchmarks. The first one extends the default benchmark of [3], which models the *acid gas removal-1100* unit of an existing plant, by adding maintenance access zone boxes and pipe t-junctions, yielding 39 boxes and 47 pipes. The second benchmark is new and models the combined *dehydration-1300* unit, *mercury removal-1500* unit and *propane circuit of the liquefaction-1400* unit of the same plant. We believe this is the largest plant layout benchmark ever considered in the literature.

Its *container cuboid* is sized $250 \times 100 \times 40$ m length by width by height, discretized by 200 mm, yielding $1250 \times 500 \times 200$ position points along axes x, y, z , respectively. It has 85 pipes in \mathcal{P} , with diameters D_p between 50 and 1400 mm, and 76 boxes in \mathcal{B} including: 12 columns and vessels, with heights between 1.5 and 26 m; 2 heat exchanger groups and 8 individual heat exchangers including 4 fin-fan blocks (the groups have size $22 \times 4 \times 6$ m, while the sizes of the individual ones range from $6.5 \times 1.5 \times 1.5$ m to $173 \times 15 \times 5$ m); 1 source and 9 sink points connecting the equipment to the outside; 2 pipe racks of size $75 \times 15.5 \times 18$ m and $186 \times 15.5 \times 18$ m, where levels at heights 3, 6, 9, and 12 m provide support without cost; a pump group of size $4 \times 1 \times 4.5$ and two compressors of sizes $9 \times 3 \times 3$ and $7 \times 5.5 \times 5.5$ m; 3 general equipment of sizes $7.5 \times 5.5 \times 5.5$, $12 \times 10 \times 39$ and $25 \times 27 \times 13$ m; 2 small mixers of size $0.7 \times 0.6 \times 0.6$ m each; 7 strainers of sizes from $1 \times 0.5 \times 0.5$ to $5.5 \times 1.5 \times 1.5$ m; 17 pipe t-junctions; and 10 maintenance access zones (4 truck, 4 landing and 2 extract zones).

The use of MiniZinc allowed us to try several solvers, including two state-of-the-art MIP solvers (Gurobi 7.5.2 [7] and IBM ILOG CPLEX 12.8 [8]) and the two CP solvers that gave the best performance for phase two in [3] (Chuffed [5] and Gecode 6.0 [15]). For MIP solvers we *warm-start* the solver on each neighbourhood, i.e., the last solution is not “destroyed” [13] by demanding a strictly better objective value, but provided as a solution hint to the solver [7,8]. This considerably sped-up MIP by allowing the efficient solving of much larger neighbourhoods than in [18]. Note that only the variable values are provided for a warm start.

Table 1 provides the results for our two benchmarks (denoted as Unit1100 and Unit1300+) using the MIP solvers, with parameters $L=15$ and $U=20$ for building the neighbourhoods. For each solver and type of neighbourhood, it shows the time in getting the first solution and its associated objective value ($\times 10^3$); the objective value and associated gap of the solution found by the warm-started solver with the given timeout (30, 60 and 120 seconds for Unit1100, and 60, 300 and 600 for Unit1300+), and the objective value and associated gap of the solution found after performing all LNS iterations (20 for Unit1100, 50 for Unit1300+). Note that the gap shown is computed using the best lower bound found in any run. For Unit1100 this is the optimal value, 546797, which is found by Gurobi in 802 seconds. For Unit1300+ it is 1440125.44 after two days of computation with 8 threads, and might still be suboptimal.

Table 1. Unit 1100 with 20 LNS iterations and Unit 1300+ with 50 LNS iterations

Solver	First Solution		Sequential LNS						Random LNS					
	Time (sec)	Total Cost (10 ³)	Restart			End (after LNS)			Restart			End (after LNS)		
			Time limit (sec)	Obj (UB) (10 ³)	Gap (%)	End Time (sec)	Obj (UB) (10 ³)	Gap (%)	Time limit (sec)	Obj (UB) (10 ³)	Gap (%)	End Time (sec)	Obj (UB) (10 ³)	Gap (%)
Unit1100 CPLEX	13	751	30	728	24.90	186	561	2.46	30	728	24.90	220	618	11.54
			60	673	18.75	260	556	1.60	60	673	18.75	188	558	1.97
			120	654	16.42	294	556	1.66	120	654	16.42	275	555	1.53
	18	660	180	572	4.46	303	558	2.07	180	630	13.17	370	557	1.81
			30	585	6.48	146	550	0.53	30	585	6.48	122	555	1.53
			60	567	3.53	173	550	0.53	60	567	3.53	152	555	1.53
180	660	120	567	3.53	234	550	0.53	120	567	3.53	212	555	1.53	
		180	567	3.51	295	550	0.53	180	567	3.51	276	555	1.53	
		60	2698	47.23	3818	1597	10.87	60	2525	43.60	3901	1658	14.11	
Unit1300+ CPLEX	146	2721	300	2336	39.06	3967	1657	14.10	300	2277	37.48	4152	1652	13.84
			600	2283	37.63	4348	1654	13.93	600	2229	36.13	4446	1613	11.74
			60	2105	32.37	4174	1614	11.79	60	2164	34.20	4273	1553	8.35
	611	2480	300	1980	28.11	4453	1552	8.24	300	2038	30.14	4492	1555	8.47
			600	1785	20.22	4739	1600	11.02	600	1849	23.01	4888	1609	11.53
			600	1785	20.22	4739	1600	11.02	600	1849	23.01	4888	1609	11.53

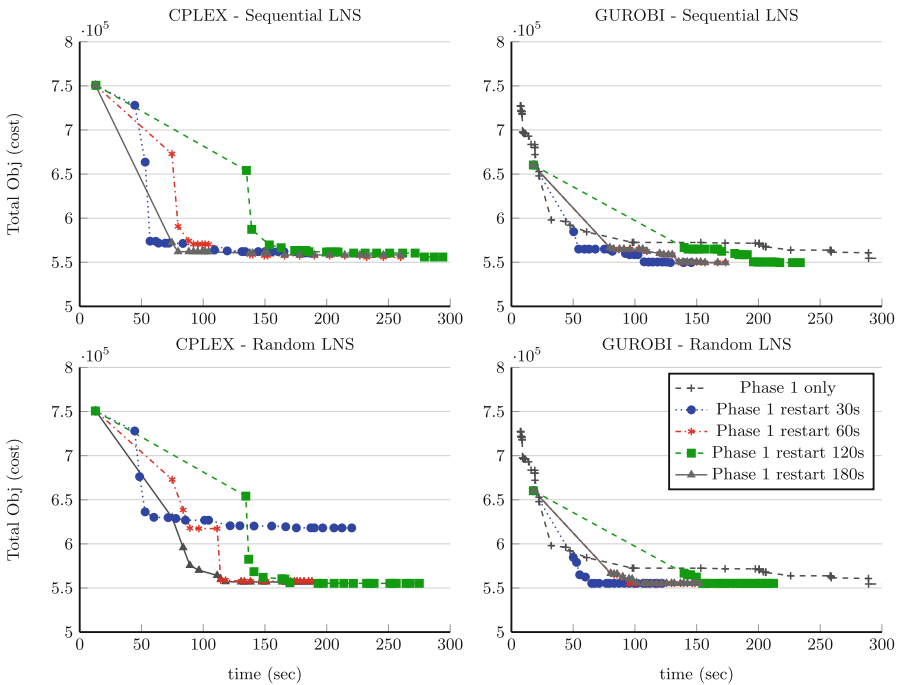


Fig. 3. Intermediate MIP solutions for Unit1100 with and without LNS. (Color figure online)

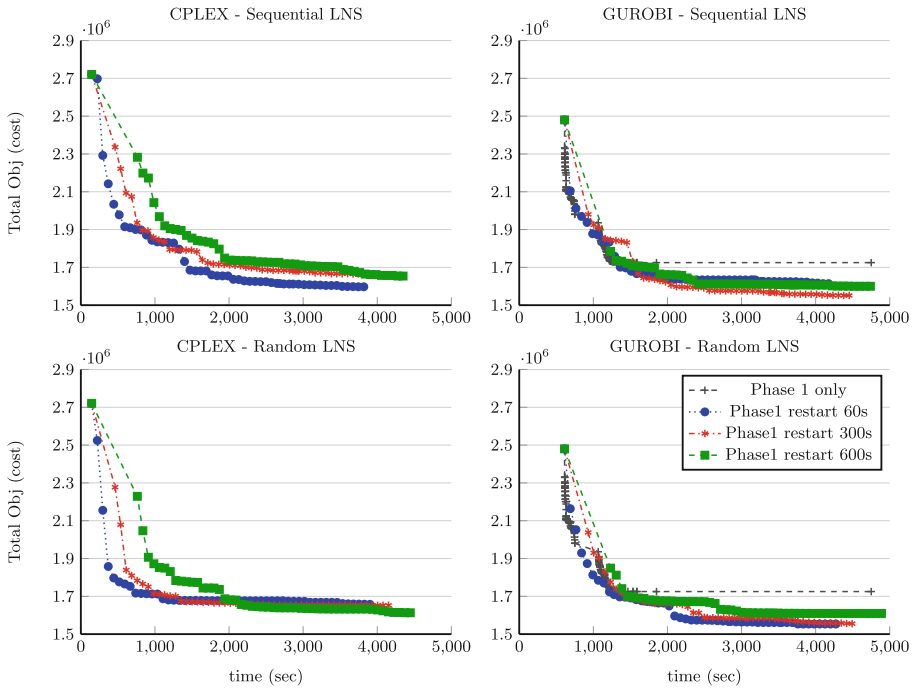


Fig. 4. Intermediate MIP solutions for Unit1300+ with and without LNS

In addition, Figs. 3 and 4 show information regarding the value of the objective function and the time (also in seconds) at which the associated solution was found for each of the 20 and 50 LNS iterations summarised in the above Table (all starting from the initial solution found by each solver). The rightmost figures for each solver also show the solutions found by that solver without LNS in the same time-frame. As the figures show, Gurobi consistently performs better than CPLEX for our model, and seems to perform most efficiently when given a short time to improve the initial feasible solution to build the seed. Its combination with LNS allows us to provide Woodside engineers with high quality solutions in under 2 minutes for Unit1100 and under 30 minutes for Unit1300+. This is quite pleasing as it is well within the expectations of Woodside, not only for obtaining a first solution, but also for performing interactive re-optimisations. Still, we would like to reduce further the time taken for the second benchmark. Thus, we plan to explore the use of other neighbourhoods, in combination with hierarchical approaches to decompose big plants.

The experiments with CP solvers were not as successful. For Chuffed we were not able to get a first solution in 1 hour, even for the smaller benchmark and trying with a variety of searches (free, model, alternating, etc.). Thus, we seeded it with a solution from Gurobi. Then, the best results were obtained when warm-started with an upper bound on the objective, allowed to alternate

between free/user-defined search (-f solving option), and the neighbourhood size was reduced to 5–15. Even then, after 50 iterations it was only able to return an objective of 574247 (and it took 4739 seconds). For Unit1300+ it never returned a solution better than the seed. For Gecode we were again not able to find a first solution for any of the two benchmarks, and seeding it with Gurobi (and setting an upper bound for the objective function) did not produce any improvement, perhaps due to its reliance on the search specified by the model.

7 Conclusions

We have presented the most realistic model ever described in the literature to solve phase one of the plant-layout problem, which positions the equipment ensuring it satisfies directional safety distances, equipment alignment, and various types of (rigid, rotatable and multi-zone) maintenance access constraints in such a way as to minimise the piping and support costs of all equipment and the overall footprint of the plant. Making the model sufficiently realistic to satisfy industry standards has been very challenging and considerably increased the search space and, thus, the time taken by the solvers to find a high-quality solution. Thus, we also developed and implemented an LNS framework that can explore larger neighbourhoods than any previous approach for this problem, thanks to the use of complete solvers able to explore the neighbourhoods efficiently. Our experimental results show that the combination of MIP solvers with LNS provides Woodside engineers with high quality solutions in under 2 and 30 minutes for our two benchmarks, respectively. The use of a constraint programming modelling language (MiniZinc) was critical to be able to modify the model as often as required, and execute it with the most efficient solver for that model.

Acknowledgements. Funded by Woodside Energy Ltd. and the Australian Research Council grant DP180100151. We thank our Woodside collaborators, particularly Solomon Faka and Michelle Frayne, for the many useful discussions.

References

1. AMEC Paragon launches optimized FEED design process. *Zeus Technology Magazine*, **4**(2), 1–3 (2009)
2. Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T.: Global constraint catalogue: Past, present and future. *Constraints* **12**(1), 21–62 (2007)
3. Belov, G., et al.: An optimization model for 3D pipe routing with flexibility constraints. In: Beck, J.C. (ed.) *CP 2017*. LNCS, vol. 10416, pp. 321–337. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66158-2_21
4. Belov, G., Stuckey, P.J., Tack, G., Wallace, M.: Improved linearization of constraint programming models. In: Rueher, M. (ed.) *CP 2016*. LNCS, vol. 9892, pp. 49–65. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44953-1_4
5. Chu, G.G.: Improving combinatorial optimization. Ph.D. thesis (2011)

6. Guirardello, R., Swaney, R.E.: Optimization of process plant layout with pipe routing. *Comput. Chem. Eng.* **30**(1), 99–114 (2005)
7. Gurobi Optimization, Inc.: Gurobi Optimizer Reference Manual Version 7.5. Gurobi Optimization, Houston, Texas (2017)
8. IBM: IBM ILOG CPLEX Optimization Studio. CPLEX User's Manual (2017)
9. Kar, Y.T., Shi, G.L.: A hierarchical approach to the facility layout problem. *Int. J. Prod. Res.* **29**(1), 165–184 (1991)
10. Mecklenburgh, J.C.: *Process Plant Layout*. Halsted Press; Wiley, New York (1985)
11. Nethercote, N., et al.: MiniZinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_38
12. Peters, M.S., Timmerhaus, K.D.: *Plant Design and Economics for Chemical Engineers*, 5th edn. McGraw-Hill Book Company, New York (2004)
13. Pisinger, D., Sigurd, M.M.: Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem. *INFORMS J. Comput.* **19**(1), 36–51 (2007)
14. Sakti, A., Zeidner, L., Hadzic, T., Rock, B.S., Quartarone, G.: Constraint programming approach for spatial packaging problem. In: Quimper, C.-G. (ed.) CPAIOR 2016. LNCS, vol. 9676, pp. 319–328. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33954-2_23
15. Schulte, C., Tack, G., Lagerkvist, M.Z.: *Modeling and programming with Gecode* (2017). www.gecode.org
16. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M., Puget, J.-F. (eds.) CP 1998. LNCS, vol. 1520, pp. 417–431. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49481-2_30
17. Xu, G., Papageorgiou, L.G.: A construction-based approach to process plant layout using mixed-integer optimization. *Ind. Eng. Chem. Res.* **46**(1), 351–358 (2007)
18. Xu, G., Papageorgiou, L.G.: Process plant layout using an improvement-type algorithm. *Chem. Eng. Res. Des.* **87**(6), 780–788 (2009)



A Constraint Programming Approach for Solving Patient Transportation Problems

Quentin Cappart^{1,2,3(✉)}, Charles Thomas¹, Pierre Schaus¹,
and Louis-Martin Rousseau^{2,3}

¹ Université catholique de Louvain, Louvain-la-Neuve, Belgium
{charles.thomas,pierre.schaus}@uclouvain.be

² Ecole Polytechnique de Montréal, Montréal, Canada
quentin.cappart@polymtl.ca

³ Interuniversity Research Centre on Enterprise Networks,
Logistics and Transportation (CIRRELT), Montréal, Canada
louis-martin.rousseau@cirrelt.ca

Abstract. The Patient Transportation Problem (PTP) aims to bring patients to health centers and to take them back home once the care has been delivered. All the requests are known beforehand and a schedule is built the day before its use. It is a variant of the well-known Dial-a-Ride Problem (DARP) but it has nevertheless some characteristics that complicate the decision process. Three levels of decisions are considered: selecting which requests to service, assigning vehicles to requests and routing properly the vehicles. In this paper, we propose a Constraint Programming approach to solve the Patient Transportation Problem. The model is designed to be flexible enough to accommodate new constraints and objective functions. Furthermore, we introduce a generic search strategy to maximize efficiently the number of selected requests. Our results show that the model can solve real life instances and outperforms greedy strategies typically performed by human schedulers.

1 Introduction

Over the years, there is an increasing demand for transports by disabled and invalid people requiring health care but that do not have the ability to go to hospitals by themselves. In this context, organizations managing the transportation of patients from their home to health centers are present in many cities. Their goal is to provide a door-to-door transportation service to a set of patients on a daily basis. Most of them are non-profit organizations that often have limited resources. Besides, they often do not have an expertise on decision support tools in order to assist them in their operations. This leads to sub-optimal decisions in most cases which has a direct negative impact on the patients and also leads to financial losses. Therefore, minimizing the operational costs while maintaining a sufficient quality of service is highly desirable and both aspects must be properly balanced.

The problem considered in this paper has been proposed by a non-profit organization operating at Liège (Belgium) which provides a range of home help services. One of them is transportation of people for medical appointments. We refer to it as the Patient Transportation Problem, which is a specific case of the well-known Dial-a-Ride Problem (DARP) [1]. The goal of this last problem consists in designing routes and schedules for a set of users who specify pickup and delivery requests between origins and destinations. It is especially present for the transportation services in the medical domain [2–4].

However, a tremendous amount of variants are possible and have been extensively studied in the literature: the fleet can be composed of several vehicles [5] that can be heterogeneous [6], users can have different characteristics [7], availability of vehicles can be constrained [8], patients can require a return trip [9], several depots can be present [10], etc. A large scope of objective functions can also be considered such as minimizing the waiting time of users or maximizing the number of accepted requests. Multi-objective approaches have also been introduced [11]. Besides, the problem can either be solved offline [12] or online [13]. In the former case, all the requests are known in advance whereas they appear gradually in real-time in the latter. Aforementioned references are only few examples of the broad literature dedicated to DARPs. A good summary of the different variants and methods was nevertheless proposed by Cordeau and Laporte [1]. As a first observation, we can see that most of the approaches are based either on Mixed Integer Programming, Local Search or Dynamic Programming. Conversely, solutions based on Constraint Programming (CP) seem to have been less studied even if some recent works exist [14–18]. However, thanks to its flexibility, we believe that CP can play an important role for solving practical DARPs.

The contribution of this paper is a flexible and efficient approach based on CP for modeling and solving the static Patient Transportation Problem, which is a specific case of the DARP. A general model is first proposed. Several extensions that can be easily integrated to the model are then detailed. A generic search strategy for maximizing the number of selected requests is also proposed. It avoids branching on variables related to a request whenever it is not selected. From a practical point of view, we provide a solution to a problem issued by a non-profit organization, handling the transportation logistic of people requiring health care. The solution we propose is usable in practice, thanks to its efficiency and flexibility to accommodate new situations. Performances of the approach are corroborated by both synthetic and real instances.

This paper is organized as follows. Next section describes the nature of the problem we are considering. Section 3 presents some recent developments related to the problem studied. A core model with its different components is firstly detailed in Sect. 4. Additional features that can be easily integrated in the model are then presented in Sect. 5. Finally, experiments on synthetic and real instances are carried out in Sect. 6.

2 Problem Description

The Patient Transportation Problem (PTP) is a static optimization problem aiming to bring patients to health centers and to take them back home once the care has been delivered. To do so, a fleet of vehicles is available. The fleet is heterogeneous and is mainly composed of ambulances and private drivers operating as volunteers. Each patient has a set of characteristics and is represented by a request. The objective is to satisfy as many requests as possible within a fixed horizon, which is typically bounded by the working hours. Three aspects of decision are considered in the PTP: (1) selecting which requests to service, (2) assigning vehicles to requests and (3) routing and scheduling appropriately the vehicles. An illustration of the PTP on a toy example with two patients (A and B) and a single vehicle is shown in Fig. 1. A possible solution consists in the following sequence: taking A (S_1), bringing A to the hospital (S_2), taking B (S_3), taking back A (S_4), dropping A to its home (S_5), bringing B to the hospital (S_6), waiting for B (S_7) and dropping B to its home (S_8). Some specific characteristics must also be considered in the PTP. Here are some of them:

- Patients can have several constraints such as a maximum travel time or a maximum waiting time at the hospital. The time to embark and disembark a patient must sometimes be considered.
- The set of requests is heterogeneous. Some patients only require to go from their home to a health center, while some of them also need a return trip once the care has been delivered. In the latter case, they must be taken back home, or to another place if requested. It is also possible to have patients requiring only a return trip. Besides, requests can involve more than one passenger at once. For instance, a child can be accompanied by his parents.
- The vehicle fleet is heterogeneous. Vehicles can differ by their capacity, their initial/final location (typically a depot) and their availability. Some patients can only be taken by particular vehicles. For instance, patients in wheelchairs can only be transported by specific vehicles.
- Availability of vehicles can be non continuous. For instance, they can be available from 9am to 1pm and from 3pm to 6pm.

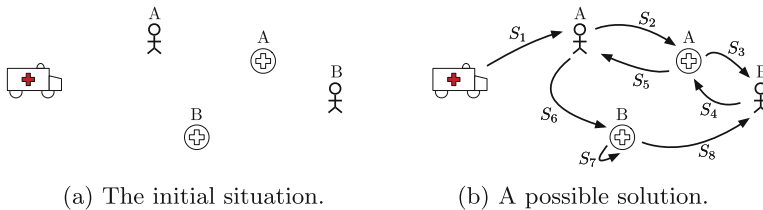


Fig. 1. Illustration of the PTP with one vehicle and two patients.

Let us finally notice that this version of PTP is static: the whole set of requests is known beforehand and no new request is added in real time. It is used by the organization for designing the first daily schedule given the pool of requests received the previous days.

3 Related Work

To the best of our knowledge, the approach of Liu et al. [18] is the closest and most recent work related to our problem. The authors model and solve the Senior Transportation Problem (STP) using different approaches: CP, MIP and Logic Based Benders Decomposition. The objective is also to maximize the sum of the (weighted) served requests. Their results show that the CP model has the best performances. The STP shares many similarities with our problem but has nevertheless some differences:

- Requests are one-way only and there is no return trip.
- The problem is a transportation problem where the selection of each request is constrained only by the vehicles availability and a maximum travel time, there are no constraints related to the appointment for care.
- There are no constraints linking patients to specific vehicles.

While some constraints are straightforward to add in the STP model, the integration of others would require more modifications. For instance, by properly defining the time windows to make sure the patients arrive on time for their care, appointment constraints for the care can be handled by the STP. However, additional constraints would be necessary to link forward with backward trips and preserve the consistency of the tour. Ensuring that vehicles are the same or can be different for both trips also requires some modifications.

Besides, the modeling and solving parts are also different. In the approach of Liu et al. [18], each decision variable is linked to a location and auxiliary variables are introduced to express that a location is visited by a particular vehicle. In our model, the decision variables are linked to trips instead of visited locations. We express capacity constraints with the standard `cumulative` constraint [19] and can take advantage of efficient propagators [20–24]. Conversely, Liu et al. [18] enforce the capacity constraints of vehicles through renewable resources and `cumul` functions using the `StepAtStart` functions from CP Optimizer. Those abstractions are less standard in CP solvers and modeling languages such as Minizinc or XCSP3 (renewable resources can be modeled with cumulative constraints [25]). Finally we use a custom search strategy combined with a Large Neighborhood Search while Liu et al. rather uses the CP Optimizer default search.

4 Modeling

This section presents a CP model for the PTP, flexible to easily handle different variants of the problem, and efficient enough to solve real instances. The PTP is modeled as a constrained based scheduling problem.

Parameters. Let R be the set of requests and V the set of vehicles. Each request is linked to a patient. The related parameters are depicted in Table 1. Some of them correspond to a location ($start_i$, $dest_i$ and ret_i) and are used for computing a travel time matrix ($T_{i,j}$) from location i to j . A request consists in two trips: a forward trip from a start location to a destination ($start_i$ to $dest_i$) and a backward trip from the previous destination to a return location ($dest_i$ to ret_i).

Table 1. Parameters used in the model.

Entity	Parameter	Meaning
Request	$start_i$	Starting place of the patient linked to request i
	$dest_i$	Place where the care is delivered for the patient of request i
	ret_i	Return place of the patient linked to request i
	l_i	Number of places taken by the patient of request i
	u_i	Time at which the health care service begins for request i
	d_i	Time needed to deliver the care for the patient of request i
	p_i	Maximum travel time of the patient linked to request i
	c_i	Category of patient of request i (wheelchair, without, etc.)
Vehicle	k_j	Capacity of vehicle j (i.e. the number of places available)
	C_j	Set of patient categories that vehicle j can take

Decision Variables. The problem is to choose which requests will be selected, the vehicles assigned to the requests, the route of the vehicles and their timetable. We model it as a scheduling problem with conditional activities using the formalism proposed by Laborie et al. [26–28]. In the standard form, each conditional activity A_i is modeled with four variables, a start date $s(A_i)$, a duration $d(A_i)$, an end date $e(A_i)$ and a binary execution status $x(A_i)$. If the activity is executed, it behaves as a classical activity that is executed on its time interval, otherwise it is not considered by any constraint. In our case, we also define $v(A_i)$ as the vehicle that has been assigned to an activity A_i . Each request (i) is attached to a forward activity (A_i^F) defining the time slot when the patient is brought from its home to the health center (from $start_i$ to $dest_i$) and to a backward activity (A_i^B) for the time interval of the return trip (from $dest_i$ to ret_i). Furthermore, A_i denotes any activity, either forward or backward, A^F the set of forward activities and A^B the set of backward activities. Equation 1 defines A_i^o and A_i^d as the origin and the destination locations of the activities linked to a request i .

$$\forall i \in R : \begin{cases} A_i^o = \begin{cases} start_i & \text{if } A_i \in A^F \\ dest_i & \text{if } A_i \in A^B \end{cases} \\ A_i^d = \begin{cases} dest_i & \text{if } A_i \in A^F \\ ret_i & \text{if } A_i \in A^B \end{cases} \end{cases} \tag{1}$$

Temporal relations between activities are illustrated in Fig. 2a for an arbitrary example. The focus is on activity A_i^F . There are four specific transition times ($T_{x,y}$) with any other activity (A_j^F on this example), they correspond to the time to go from A_i^o to A_j^o , from A_i^o to A_j^d , from A_i^d to A_j^o and from A_i^d to A_j^d . Activity A_i^F must also be completed before the appointment of the request (u_i), and the related backward activity cannot begin before the end of the appointment ($u_i + d_i$). Finally, each activity is executed on a resource, representing the vehicle assigned to the activity. At any moment, the load of the vehicle cannot exceed its capacity. It is illustrated in Fig. 2b by a load profile for an arbitrary set of 4 activities executed on the same vehicle.

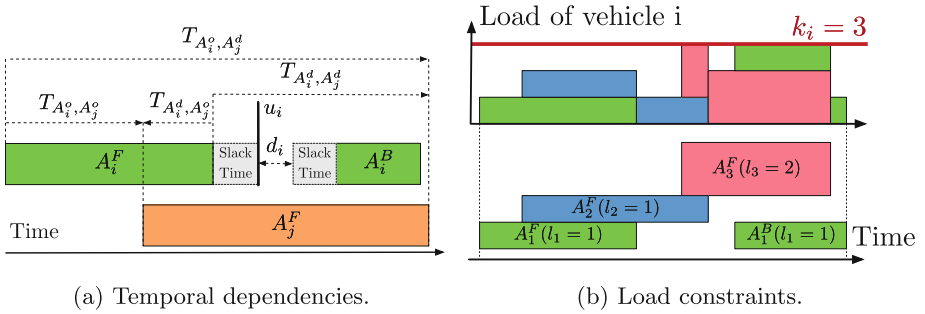


Fig. 2. Illustration of the parameters of Table 1.

Decision variables related to the selection of requests are depicted in Eq. 2. They are boolean variables defining whether the request is selected or not.

$$\forall i \in R : S_i \in \{0, 1\} \quad (2)$$

Variables related to the conditional activities are shown in Eq. 3. Patients cannot arrive at the health center after the time at which the appointment begins (forward activity) and cannot leave it before the end of the care (backward activity). Symbol H denotes the time horizon considered. The domain of the vehicle selection variables (v) contains only the vehicles that are compatible with the patient category of the request. Note that the duration variable (d) is not a decision variable as its value depends on the start (s) and the end (e) of the activity. Domains for forward activities implicitly handle the deadline satisfaction for the care for each request. It ensures that the patients arrive to the health center ahead of schedule for their care ($e(A_i^F) \leq u_i$). Similarly, domains for backward activities ensure that patients cannot leave the center before the time at which the care has been delivered ($s(A_i^B) \geq u_i + d_i$).

$$\forall i \in R : \begin{cases} s(A_i^F) \in [0, u_i] \\ e(A_i^F) \in [0, u_i] \\ d(A_i^F) = e(A_i^F) - s(A_i^F) \\ x(A_i^F) \in \{0, 1\} \\ v(A_i^F) \in \{j \mid j \in V \wedge c_i \in C_j\} \end{cases} \quad \begin{cases} s(A_i^B) \in [u_i + d_i, H] \\ e(A_i^B) \in [u_i + d_i, H] \\ d(A_i^B) = e(A_i^B) - s(A_i^B) \\ x(A_i^B) \in \{0, 1\} \\ v(A_i^B) \in \{j \mid j \in V \wedge c_i \in C_j\} \end{cases} \quad (3)$$

Constraints

Binding Requests to Activities. A request is selected if and only if the forward and backward activities are both completed (Eq. 4).

$$\forall i \in R : (S_i = 1) \equiv (x(A_i^F) = 1 \wedge x(A_i^B) = 1) \tag{4}$$

Forward and Backward Selection. A forward and backward activity linked to the same request must have the same execution status (Eq. 5). This constraint is redundant with Eq. 4 but can nevertheless be used for a better pruning.

$$\forall i \in R : x(A_i^F) = x(A_i^B) \tag{5}$$

Inter-Activity Time Travel Consistency. The start/end of an activity cannot overlap with the start/end of other activities when they are processed by the same vehicle. The time interval between any two locations visited by a same vehicle is at least the time required to travel between these two locations (Eq. 6). It is also referred as *setup time*. It is illustrated in Fig. 2a. The \vee relation is used to consider situations where activity A_i occurs before or after A_j .

$$\forall i, j \in R \mid i \neq j : \begin{cases} (v(A_i) = v(A_j)) \rightarrow ((s(A_j) - s(A_i) \geq T_{A_i^o, A_j^o}) \vee (s(A_i) - s(A_j) \geq T_{A_j^o, A_i^o})) \\ (v(A_i) = v(A_j)) \rightarrow ((s(A_j) - e(A_i) \geq T_{A_i^o, A_j^d}) \vee (s(A_i) - e(A_j) \geq T_{A_j^o, A_i^d})) \\ (v(A_i) = v(A_j)) \rightarrow ((e(A_j) - s(A_i) \geq T_{A_i^d, A_j^o}) \vee (e(A_i) - s(A_j) \geq T_{A_j^d, A_i^o})) \\ (v(A_i) = v(A_j)) \rightarrow ((e(A_j) - e(A_i) \geq T_{A_i^d, A_j^d}) \vee (e(A_i) - e(A_j) \geq T_{A_j^d, A_i^d})) \end{cases} \tag{6}$$

An alternative way to enforce the travel times is to use a `NoOverlap` with transition time constraint imposed on activities created at each location [18]. In particular, the propagator proposed by Dejemeppe et al. [29] could possibly be extended to handle optional activities. But the decomposition approach relying on reification and binary constraints is arguably the most portable formulation for other solvers and modeling languages.

Intra-Activity Time Travel Consistency. The duration of each activity cannot be lesser than the time required to go from the origin to the destination (Eq. 7).

$$\forall i \in R : d(A_i) \geq T_{A_i^o, A_i^d} \tag{7}$$

Maximum Travel Time. It is also suitable to constraint the maximal travel time of patients. It prevents situations where a patient stays too long in a vehicle. To do so, the duration of each activity is constrained (Eq. 8).

$$\forall i \in R : d(A_i) \leq p_i \tag{8}$$

Cumulative Resource. At any moment, the number of places occupied by patients in a same vehicle j cannot exceed its capacity k_j (Eq. 9). This behaviour is illustrated in the arbitrary example of Fig. 2b. This constraint is referred in the literature as the cumulative resource global constraint [19]. In our case, each activity A_i consumes l_i resources. We use the filtering algorithm of Gay et al. [20]. The vehicle of a non-executed activity is not considered by the constraint.

$$\forall j \in V : \text{cumulative} \left(\left\{ (A_i, l_i) \mid i \in R \wedge v(A_i) = j \right\}, k_j \right) \quad (9)$$

Objective Function. The first criterion considered for the objective function is the satisfaction of requests. We want to maximize the number of served requests (Eq. 10). Other objective functions can be considered. For instance, we could be interested in minimizing the accumulated travel time for all the patients (Eq. 11). The travel time of a request corresponds to the duration of its activities. It is also possible to minimize the maximum travel time (Eq. 12). To do so, the maximal duration of the whole set of activities has to be minimized. Other objective functions are also proposed by Cordeau and Laporte [1]. They can be used together inside the same model using either a lexicographic ordering or a Pareto multi objective criterion [30].

$$\max \left(\sum_{i \in R} S_i \right) \quad (10)$$

$$\min \left(\sum_{i \in R} d(A_i) \right) \quad (11)$$

$$\min \left(\max_{i \in R} d(A_i) \right) \quad (12)$$

Search Phase. The search tree is explored using a standard branch and bound depth first search. The decision variables are divided into two categories: the *request variables* (Eq. 2) and the *activity variables* (Eq. 3). Given the main objective of the problem (maximizing the number of served patients), our primal heuristic is to select patients on the left branches ($S_i = 1$) and discard them on the right branches ($S_i = 0$). Whenever a patient has been selected in a search node, all its related activity variables are subsequently assigned (start time, duration and end time and vehicle) before considering again the next patient selection variable. On the contrary, whenever a patient is not selected ($S_i = 0$ on the right branch), there is no need to consider the other decision variables related to this patient. The idea is to branch on the activity variables only if the related request variable has been selected ($S_i = 1$). Otherwise, no search is performed on the activity variables. We denote this search strategy as the *Maximum Selection Search*. The main asset of this search is that activity variables are branched on only when they are relevant to a solution. It drastically reduces the size of the search tree. An example of search tree is illustrated in Fig. 3.

This meta-search strategy for optional activities can be combined with any existing variable-value heuristic or used for similar applications such as packing as most rectangles as possible. As a variable heuristic on the request variables we use a Conflict Ordering Search heuristic (COS) [31]. A conflict is recorded on a request only when it is impossible to assign in the sub-tree all its other activity variables. The fallback heuristic combined with COS is to select the next requests with the highest *minimum slack*, defined as the sum of the minimum duration multiplied by the patient load for its forward and backward activities. The subsearch on the other activity variables follows a *min-domain first fail* strategy for the variable selection and a custom greedy value heuristic based on the type of the corresponding variable which can be a time-related decision or a vehicle choice. In the former case, the heuristic selects the closest time to the corresponding appointment. In the latter case, the vehicle that has the most remaining places is selected.

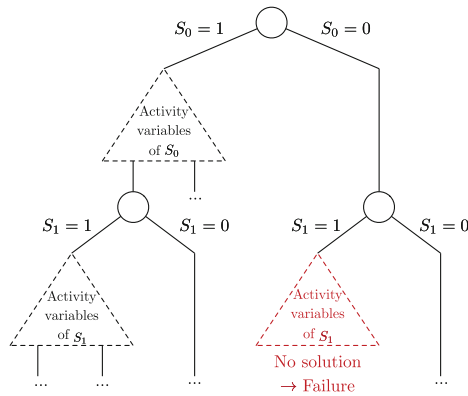


Fig. 3. Canonical shape of the search tree for two request variables (S_0 and S_1).

Large Neighborhood Search. In order to boost the performances on large instances, a Large Neighborhood Search (LNS) [32] is also used. At each iteration, a set of request variables is chosen randomly and then relaxed. The other variables are fixed to their value in the last solution. For the request variables that are selected ($S_i = 1$), the corresponding activity variables are also fixed based on the current solution. The remaining unbound variables form a smaller search space which is explored using the search defined earlier. A new iteration is started when the reduced search space is completely explored or once a fixed number of backtracks is reached.

5 Extensions of the Model

One of the main asset of this model is its flexibility to easily accommodate new constraints depending on the situation. This section presents some variants of the problem and how they can be integrated in the core model.

Mandatory Requests. It is possible to enforce the selection of some requests (Eq. 13). Parameter m_j is a boolean value indicating if a request j is mandatory.

$$\forall i \in \{j \mid j \in R \wedge m_j = 1\} : S_i = 1 \tag{13}$$

Maximum Waiting Time. The time that a patient has to wait at the health center, either before or after his care, is often constrained. Parameter w_i indicates the maximum amount of time that a patient can wait. It is handled by adapting the definition of domains in Eq. 3. It avoids situations where patients are dropped to the health center too early or taken back too late (Eq. 14).

$$\forall i \in R : \begin{cases} s(A_i^F) \in [0, u_i - w_i] \\ e(A_i^F) \in [0, u_i - w_i] \\ (...) \end{cases} \quad \begin{cases} s(A_i^B) \in [u_i + d_i + w_i, H] \\ e(A_i^B) \in [u_i + d_i + w_i, H] \\ (...) \end{cases} \tag{14}$$

Integrating Service Time. Most often, the time required to embark or disembark a patient is negligible. However in some cases, it could be more representative to consider it. For instance, embarking a patient with a wheelchair can take a significant amount of time. Such a dependency can be integrated in the inter-activity time travel consistency constraints defined in Eq. 6.

Vehicles Availability. Vehicles can also have constraints on their availability. They are available during a period and cannot leave their initial position (i.e. a depot) before the period. Similarly, they have to go back to the depot before the end of the period. Let us introduce $start_j$ the starting location of a vehicle j , $dest_j$ its return destination and $[b_j, r_j]$ its availability window. The travel time matrix (T) defined previously is extended in order to take into account these new locations. We define $D_i^o = start_{v(A_i)}$ and $D_i^d = dest_{v(A_i)}$ as the origin/destination location of the vehicle linked to activity A_i as defined in Eq. 3. Constraints on vehicle availability are expressed in Eq. 15. It states that an activity cannot begin before the availability of its vehicle plus the time required to go from the initial depot to the patient place. Similarly, the vehicles must have enough time to return to their depot in order to stay in the availability window.

$$\forall i \in R : \begin{cases} s(A_i) \geq b_{v(A_i)} + T_{D_i^o, A_i^o} \\ e(A_i) \leq r_{v(A_i)} - T_{A_i^d, D_i^d} \end{cases} \tag{15}$$

Finally, some vehicles can have non continuous availability. For instance, they can be available from 9am to 1pm and from 3pm to 6pm. We handle this specificity by duplicating the vehicles for each continuous interval. The availability of each vehicle is then composed by a unique interval. In practice, vehicles are duplicated at most once (morning and afternoon shift).

Same Vehicle Forward/Backward. The forward and the backward trips can be constrained in order to be handled by the same vehicle (Eq. 16). Parameter q_j is a boolean value indicating if the forward and the backward trip of request j must be handled by the same vehicle.

$$\forall i \in \{j \mid j \in R \wedge q_j = 1\} : v(A_i^F) = v(A_i^B) \quad (16)$$

Empty Locations. Some patients only require to go from their home to a health center without return trip. It is also possible to have patients needing only a trip from the health center to their home. A location can then be empty. When the start location is empty the request has no forward trip. Similarly, there is no backward trip when the return location is empty. This variant is handled by extending the notion of requests. A request has no forward activity when the start location is empty and no backward activity when the return location is empty. In such cases, some constraints of the previous are simplified or removed in order to consider only situations involving a forward or a backward activity. More specifically, Eq. 4 is adapted as follows (Eq. 17, \vee instead of \wedge) and constraint in Eq. 5 does not hold anymore.

$$\forall i \in R : (S_i = 1) \equiv (x(A_i^F) = 1 \vee x(A_i^B) = 1) \quad (17)$$

6 Experimental Results

This section evaluates the performance of the model on synthetic and real instances. The model tested is referred as the *Scheduling with Maximal Selection Search* (SCHED+MSS) approach. It corresponds to the core model described in Sect. 4 with the following extensions: *maximum waiting time*, *integrating service time*, *vehicles availability* and *empty locations*. No constraints on the *maximum travel time* were asked by the partner organization. Finally, the objective considered is to maximize the number of requests satisfied (Eq. 10).

Approaches Considered. Our model is compared with four other approaches: a greedy search, the same CP model without the maximal selection search, a similar scheduling model implemented in CP Optimizer and a successor model, more standard for solving routing problems with CP.

Greedy Search (GREEDY). It mimics the manual decision process used by the non-profit organization. It consists in selecting first the requests having the smallest starting time and choosing for them the closest compatible vehicle. The idea is to minimize the time between the trips of each vehicle across the requests. Each trip is inserted at the earliest possible time such that later trips can be inserted with more flexibility. If a trip cannot be inserted, the request is discarded.

Successor Model (SUCC). As an alternative to our approach, a successor model was considered. Similar models were used for solving DARPs using CP [15, 17]. Each trip is represented by two stops which correspond to the place where the patient is loaded and the place where they are unloaded. Each request has then two or four stops depending on whether it is a single trip or a double trip. The successor and the predecessor of each stop are both modeled by a variable indicating the next and the previous stop. As in [17], ride time and vehicle capacity constraints are modeled via auxiliary variables representing the load, serving vehicle, and serving time for each requests. A *circuit* constraint [33] ensures that the successor and predecessor variables form a circuit without sub-tours for each vehicle. The requests that are not serviced are assigned to a same dummy vehicle with infinite capacity. Finally, a maximum selection heuristic wrapped under LNS and a COS variable heuristic are also used for the search.

CP Optimizer implementation (CPO). The scheduling model has been implemented in IBM CP Optimizer in order to compare our search with the default search proposed by this solver. This search combines LNS with a failure directed search (FDS) strategy [34]. In order to accommodate the solver, the capacity constraints of vehicles are modeled using *cumul* functions in the same way as in the model of Liu et al. [18].

Scheduling Model with Simple Search (SCHED). It corresponds to the model presented in the previous section without the *maximal selection search* heuristic. Additional reified constraints assign the activity variables to a default value when a request is not served. It is used to avoid wasting time searching on activity variables when the corresponding request is not selected.

Datasets Used. The experiments are based on two datasets, a synthetic and a real one. The synthetic dataset has been randomly generated based on the characteristics of the problem. Synthetic instances are classified according to their size (number of patients, vehicles and health centers) and their difficulty which is related to the amount of constraints and the availability of vehicles. The real dataset has been provided by the non-profit organization. It corresponds to one month of exploitation with one instance per day. Each of them contains the requests received for the day, the vehicles available.

Experimental Protocol. Experiments have been carried out on an AMD Opteron 6176 processor (2300 MHz). Execution time for a run is limited to 1800 s and memory consumption to 6 GB. The greedy search has been implemented in Scala and the OsaR solver [35] is used for the other models except for the CPO model that has been modeled and solved with the academic version of IBM ILOG CPLEX CP Optimizer V12.8. For the reproducibility of results, the models, the synthetic dataset and the random generator are available online on CSPLib [36].

The backtrack limit and relaxation size of the LNS are adaptive parameters initially fixed to respectively 1000 failures and 10 requests. The backtrack limit is increased by 20% when 100 consecutive iterations have failed to find a new solution and to completely explore the search. The relaxation size is increased by 20% when the relaxed search space is completely explored for 50 consecutive iterations. Search parameters are set to their defaults for CPO. The greedy solution is considered as the first solution of the LNS for each method.

Given the random nature of approaches based on LNS (SUCC, CPO, SCHED and SCHED+MSS), 5 runs for each instance with a different seed have been performed and the best solution obtained is recorded. The greedy search (GREEDY) is ran only once due to its deterministic nature. The models are also compared using the improvement ratio (ρ_m) of a method (m) defined as the relative improvement of the solution obtained with the method (x_m) compared to the solution found using the greedy search (x_{GREEDY}): $\rho_m = \frac{x_m - x_{\text{GREEDY}}}{x_{\text{GREEDY}}}$.

Results. Results for both synthetic and real instances are reported in Table 2. Instances are ordered by their difficulty and the number of patients ($|R|$). The best solution obtained for each instance is also reported. The number of patients serviced is considered as the objective value. As the relaxation size is adaptive, it can eventually grow to 100%. In this case, if the search space is completely explored, the solution is proven optimal. Besides, if all the patients are serviced, the upper bound is reached and the solution is also proven optimal. The dominating model is highlighted for each instance.

Let us first focus on synthetic instances. As we can see, the scheduling model with the maximal selection search (SCHED+MSS) obtains the best solution for almost all the tests, even when the optimum is not reached. The improvement ratio is up to 130% compared to the greedy solution. Interestingly, performance of scheduling models is correlated with the difficulty of instances: the improvement gap increases when the instances are getting harder. The greedy search (GREEDY) gives poor solutions when the problem is strongly constrained. Results regarding the scheduling model with the simple search (SCHED) shows the interest of the custom search.

The successor model (SUCC) is outperformed by the scheduling approaches. This is expected as the successor model has a larger search space due to the additional decisions variables compared to the scheduling model. Furthermore, the successor approach makes the insertion of new stops in routes more difficult as it requires to change the value of the successor variables forming the routes in addition to the vehicle variable. This limits the effectiveness of the LNS.

Concerning the CP Optimizer model (CPO), it is also outperformed by the two other scheduling approaches. Such results are mainly due because of the default search used in CPO model: it is generic and not designed for this specific problem. However, it is important to point out that on harder instances, it tends to perform better than the successor model. This could indicate that the model used contributes more to the effectiveness of the approach than the search method. Note that as the CPO approach is based on another solver, other factors could also influence the performances.

Table 2. Experimental results ($|R|$, $|V|$ and $|H|$ are the number of requests, vehicles and hospitals ; ρ is the improvement ratio in percent, * indicates that the solution has been proven optimal).

Instances					GREEDY	SUCC		CPO		SCHED		SCHED+MSS		
Difficulty	Name	$ H $	$ V $	$ R $	BestSol	Sol	Sol	ρ	Sol	ρ	Sol	ρ	Sol	ρ
Easy	RAND-E-1	4	2	16	*15	14	15	7.1	*15	7.1	*15	7.1	15	7.1
	RAND-E-2	8	4	32	*32	32	32	0.0	*32	0.0	*32	0.0	*32	0.0
	RAND-E-3	12	5	48	*28	26	26	0.0	*28	7.7	28	7.7	*28	7.7
	RAND-E-4	16	6	64	62	58	61	5.2	59	1.7	62	6.9	62	6.9
	RAND-E-5	20	8	80	74	72	73	1.4	72	0.0	73	1.4	74	2.8
	RAND-E-6	24	9	96	95	91	93	2.2	92	1.1	92	1.1	95	4.4
	RAND-E-7	28	10	112	106	100	101	1.0	100	0.0	103	3.0	106	6.0
	RAND-E-8	32	12	128	*128	127	*128	0.8	127	0.0	*128	0.8	*128	0.8
	RAND-E-9	36	14	144	142	141	142	0.7	141	0.0	142	0.7	142	0.7
	RAND-E-10	40	16	160	157	154	154	0.0	157	1.9	157	1.9	157	1.9
Medium	RAND-M-1	8	2	16	*12	8	9	12.5	11	37.5	*12	50.0	11	37.5
	RAND-M-2	16	3	32	19	16	18	12.5	17	6.3	19	18.8	19	18.8
	RAND-M-3	24	4	48	32	25	25	0.0	26	4.0	30	20.0	32	28.0
	RAND-M-4	32	4	64	37	25	25	0.0	33	32.0	35	40.0	37	48.0
	RAND-M-5	40	5	80	55	45	45	0.0	48	6.7	51	13.3	55	22.2
	RAND-M-6	48	5	96	52	36	40	11.1	40	11.1	50	38.9	52	44.4
	RAND-M-7	56	6	112	63	46	47	2.2	48	4.3	63	37.0	63	37.0
	RAND-M-8	64	8	128	83	65	70	7.7	65	0.0	81	24.6	83	27.7
	RAND-M-9	72	8	144	81	62	62	0.0	64	3.2	72	16.1	81	30.6
	RAND-M-10	80	9	160	99	73	75	2.7	75	2.7	88	20.5	99	35.6
Hard	RAND-H-1	16	2	16	*8	7	7	0.0	*8	14.3	*8	14.3	*8	14.3
	RAND-H-2	32	3	32	19	15	15	0.0	18	20.0	19	26.7	17	13.3
	RAND-H-3	48	4	48	32	18	19	5.6	23	27.8	32	77.8	29	61.1
	RAND-H-4	64	4	64	23	10	12	20.0	22	120.0	20	100.0	23	130.0
	RAND-H-5	80	5	80	42	29	31	6.9	29	0.0	38	31.0	42	44.8
	RAND-H-6	96	5	96	38	22	22	0.0	27	22.7	38	72.7	38	72.7
	RAND-H-7	112	6	112	39	25	27	8.0	32	28.0	37	48.0	39	56.0
	RAND-H-8	128	8	128	75	57	63	10.5	61	7.0	71	24.6	75	31.6
	RAND-H-9	144	8	144	72	50	54	8.0	53	6.0	67	34.0	72	44.0
	RAND-H-10	160	8	160	72	46	48	4.3	50	8.7	63	37.0	72	56.5
Real	REAL-1	1	9	2	*2	2	*2	0.0	*2	0.0	*2	0.0	*2	0.0
	REAL-2	1	9	2	*2	2	*2	0.0	*2	0.0	*2	0.0	*2	0.0
	REAL-3	3	9	3	*1	1	*1	0.0	*1	0.0	*1	0.0	*1	0.0
	REAL-4	2	9	4	*4	4	*4	0.0	*4	0.0	*4	0.0	*4	0.0
	REAL-5	5	9	21	*21	21	*21	0.0	*21	0.0	*21	0.0	*21	0.0
	REAL-6	5	9	22	*22	22	*22	0.0	*22	0.0	*22	0.0	*22	0.0
	REAL-7	5	9	23	*23	23	*23	0.0	*23	0.0	*23	0.0	*23	0.0
	REAL-8	7	9	24	*24	24	*24	0.0	*24	0.0	*24	0.0	*24	0.0
	REAL-9	15	9	45	*44	44	44	0.0	*44	0.0	*44	0.0	*44	0.0
	REAL-10	26	9	99	*98	98	98	0.0	*98	0.0	*98	0.0	*98	0.0
	REAL-11	22	9	100	91	87	89	2.3	87	0.0	90	3.4	91	4.6
	REAL-12	32	9	101	*100	97	98	1.0	97	0.0	*100	3.1	99	2.1
	REAL-13	37	9	110	103	97	98	1.0	97	0.0	100	3.1	103	6.2
	REAL-14	28	9	111	*102	99	99	0.0	100	1.0	100	1.0	*102	3.0
	REAL-15	35	9	122	110	94	97	3.2	94	0.0	102	8.5	110	17.0
	REAL-16	36	9	123	108	107	107	0.0	108	0.9	108	0.9	108	0.9
	REAL-17	42	9	128	114	103	103	0.0	105	1.9	105	1.9	114	10.7
	REAL-18	31	9	130	121	112	115	2.7	113	0.9	115	2.7	121	8.0
	REAL-19	34	9	131	114	103	107	3.9	103	0.0	108	4.9	114	10.7
	REAL-20	34	9	134	118	106	107	0.9	106	0.0	108	1.9	118	11.3
	REAL-21	39	9	136	119	108	112	3.7	108	0.0	114	5.6	119	10.2
	REAL-22	31	9	138	121	113	117	3.5	113	0.0	117	3.5	121	7.1
	REAL-23	31	9	139	121	113	113	0.0	113	0.0	115	1.8	121	7.1
	REAL-24	37	9	139	110	103	103	0.0	104	1.0	106	2.9	110	6.8
	REAL-25	39	9	139	125	118	118	0.0	121	2.5	121	2.5	125	5.9
	REAL-26	38	9	140	119	107	107	0.0	109	1.9	115	7.5	119	11.2
	REAL-27	35	9	147	129	120	121	0.8	120	0.0	126	5.0	129	7.5
	REAL-28	34	9	151	131	115	116	0.9	115	0.0	121	5.2	131	13.9
	REAL-29	39	9	155	127	117	119	1.7	117	0.0	123	5.1	127	8.5
	REAL-30	41	9	159	131	115	115	0.0	119	3.5	121	5.2	131	13.9

Similar results are observed for the real instances. The scheduling model with the maximal selection search is dominating again. However, the improvement ratio is now up to 17% only. It happens because such real instances are easier to solve compared to the medium and difficult synthetic instances. It shows both the pertinence of the scheduling model and the search framework we introduced.

Finally, we also considered the waiting time minimization (Eq. 11) as a secondary objective using a lexicographical search. However, it yielded only minor improvements regarding the solution obtained using the main objective. It mainly occurs because the value heuristic used already ensures that solutions minimizing the waiting time are tried first.

7 Conclusion and Perspective

In many countries, there is an increasing demand for disabled people requiring health care. Providing a door-to-door transportation to patients minimizing the operational costs while maintaining a sufficient quality of service is still a challenge nowadays. In this context, we introduced the Patient Transportation Problem, which is a specific case of the well-known Dial-a-Ride Problem. This paper proposes a CP approach based on scheduling for solving Patient Transportation Problems. The focus was to design a flexible approach that can easily handle different variants of the problem while being efficient enough to solve real instances. Experimental results have shown that the scheduling models outperforms greedy strategies and successor models often used in classical Vehicles Routing Problems. A generic search strategy maximizing the number of selected requests is also proposed and improves the results.

In practice, Patient Transportation Problems also have a dynamic aspect: new requests, or modification/cancellation of old ones can occur online and a new solution must be found in real time. As future work, we plan to extend our approach in order to deal with such aspects. To do so, we plan to use the CP solution as an initial solution and local search for quickly adapting the solution as modifications are received.

Having discovered recently the approach of Liu et al. [18] developed in parallel with our work, we also wish to investigate experimentally the differences of performances with both models. We also plan to design more advanced LNS relaxations, for instance based on partial order schedules [37]. Lazy clause generation approaches relying on explaining the cumulative constraint [24] may also be worth trying on this problem.

Acknowledgments. This research is financed by the Walloon Region (Belgium) as part of PRESupply Project. The problem has been proposed by the CSD, a Belgian non-profit organization operating at Liège.

References

1. Cordeau, J.F., Laporte, G.: The dial-a-ride problem: models and algorithms. *Ann. Oper. Res.* **153**, 29–46 (2007)
2. Melachrinoudis, E., Min, H.: A tabu search heuristic for solving the multi-depot, multi-vehicle, double request dial-a-ride problem faced by a healthcare organisation. *Int. J. Oper. Res.* **10**, 214–239 (2011)
3. Liu, R., Xie, X., Augusto, V., Rodriguez, C.: Heuristic algorithms for a vehicle routing problem with simultaneous delivery and pickup and time windows in home health care. *Eur. J. Oper. Res.* **230**, 475–486 (2013)
4. Detti, P., Papalini, F., de Lara, G.Z.M.: A multi-depot dial-a-ride problem with heterogeneous vehicles and compatibility constraints in healthcare. *Omega* **70**, 1–14 (2017)
5. Cordeau, J.F., Laporte, G.: A tabu search heuristic for the static multi-vehicle dial-a-ride problem. *Transp. Res. Part B Methodol.* **37**, 579–594 (2003)
6. Parragh, S.N.: Introducing heterogeneous users and vehicles into models and algorithms for the dial-a-ride problem. *Transp. Res. Part C Emerg. Technol.* **19**, 912–930 (2011)
7. Parragh, S.N., Cordeau, J.F., Doerner, K.F., Hartl, R.F.: Models and algorithms for the heterogeneous dial-a-ride problem with driver-related constraints. *OR Spectr.* **34**, 593–633 (2012)
8. Psaraftis, H.N.: An exact algorithm for the single vehicle many-to-many dial-a-ride problem with time windows. *Transp. Sci.* **17**, 351–357 (1983)
9. Melachrinoudis, E., Ilhan, A.B., Min, H.: A dial-a-ride problem for client transportation in a health-care organization. *Comput. Oper. Res.* **34**, 742–759 (2007)
10. Cordeau, J.F., Gendreau, M., Laporte, G.: A tabu search heuristic for periodic and multi-depot vehicle routing problems. *Networks* **30**, 105–119 (1997)
11. Parragh, S.N., Doerner, K.F., Hartl, R.F., Gandibleux, X.: A heuristic two-phase solution approach for the multi-objective dial-a-ride problem. *Networks* **54**, 227–242 (2009)
12. Berbeglia, G., Cordeau, J.F., Gribkovskaia, I., Laporte, G.: Static pickup and delivery problems: a classification scheme and survey. *Top* **15**, 1–31 (2007)
13. Attanasio, A., Cordeau, J.F., Ghiani, G., Laporte, G.: Parallel tabu search heuristics for the dynamic multi-vehicle dial-a-ride problem. *Parallel Comput.* **30**, 377–387 (2004)
14. Berbeglia, G., Pesant, G., Rousseau, L.M.: Checking the feasibility of dial-a-ride instances using constraint programming. *Transp. Sci.* **45**, 399–412 (2011)
15. Berbeglia, G., Cordeau, J.F., Laporte, G.: A hybrid tabu search and constraint programming algorithm for the dynamic dial-a-ride problem. *INFORMS J. Comput.* **24**, 343–355 (2012)
16. Parragh, S.N., Schmid, V.: Hybrid column generation and large neighborhood search for the dial-a-ride problem. *Comput. Oper. Res.* **40**, 490–497 (2013)
17. Jain, S., Van Hentenryck, P.: Large neighborhood search for dial-a-ride problems. In: Lee, J. (ed.) *CP 2011. LNCS*, vol. 6876, pp. 400–413. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23786-7_31
18. Liu, C., Aleman, D.M., Beck, J.C.: Modelling and solving the senior transportation problem. In: van Hoesve, W.-J. (ed.) *CPAIOR 2018. LNCS*, vol. 10848, pp. 412–428. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93031-2_30
19. Beldiceanu, N., Carlsson, M., Rampon, J.X.: Global constraint catalog, (revision a) (2012)

20. Gay, S., Hartert, R., Schaus, P.: Simple and scalable time-table filtering for the cumulative constraint. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 149–157. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_11
21. Vilím, P.: Timetable edge finding filtering algorithm for discrete cumulative resources. In: Achterberg, T., Beck, J.C. (eds.) CPAIOR 2011. LNCS, vol. 6697, pp. 230–245. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21311-3_22
22. Gay, S., Hartert, R., Schaus, P.: Time-table disjunctive reasoning for the cumulative constraint. In: Michel, L. (ed.) CPAIOR 2015. LNCS, vol. 9075, pp. 157–172. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18008-3_11
23. Ouellet, P., Quimper, C.-G.: Time-table extended-edge-finding for the cumulative constraint. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 562–577. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_42
24. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Explaining the cumulative propagator. *Constraints* **16**, 250–282 (2011)
25. Simonis, H., Cornelissens, T.: Modelling producer/consumer constraints. In: Montanari, U., Rossi, F. (eds.) CP 1995. LNCS, vol. 976, pp. 449–462. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60299-2_27
26. Laborie, P., Rogerie, J.: Reasoning with conditional time-intervals. In: FLAIRS Conference, pp. 555–560 (2008)
27. Laborie, P., Rogerie, J., Shaw, P., Vilím, P.: Reasoning with conditional time-intervals. Part II: an algebraical model for resources. In: FLAIRS Conference, pp. 201–206 (2009)
28. Laborie, P., Rogerie, J., Shaw, P., Vilím, P.: IBM ILOG CP optimizer for scheduling. *Constraints*, 1–41 (2018)
29. Dejemeppe, C., Van Cauwelaert, S., Schaus, P.: The unary resource with transition times. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 89–104. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_7
30. Ngatchou, P., Zarei, A., El-Sharkawi, A.: Pareto multi objective optimization. In: 2005 Proceedings of the 13th International Conference on Intelligent Systems Application to Power Systems, pp. 84–91. IEEE (2005)
31. Gay, S., Hartert, R., Lecoutre, C., Schaus, P.: Conflict ordering search for scheduling problems. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 140–148. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_10
32. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M., Puget, J.-F. (eds.) CP 1998. LNCS, vol. 1520, pp. 417–431. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49481-2_30
33. Lauriere, J.L.: A language and a program for stating and solving combinatorial problems. *Artif. Intell.* **10**, 29–127 (1978)
34. Vilím, P., Laborie, P., Shaw, P.: Failure-directed search for constraint-based scheduling. In: Michel, L. (ed.) CPAIOR 2015. LNCS, vol. 9075, pp. 437–453. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18008-3_30
35. Oscar Team: Oscar: Scala in OR (2012). <https://bitbucket.org/oscarlib/oscar>
36. Thomas, C., Cappart, Q., Schaus, P., Rousseau, L.M.: CSPLib problem 082: Patient transportation problem. <http://www.csplib.org/Problems/prob082>
37. Godard, D., Laborie, P., Nuijten, W.: Randomized large neighborhood search for cumulative scheduling. *ICAPS* **5**, 81–89 (2005)



Unifying Reserve Design Strategies with Graph Theory and Constraint Programming

Dimitri Justeau-Allaire^{1,2,3(✉)}, Philippe Birnbaum^{1,2,3}, and Xavier Lorca⁴

¹ CIRAD, UMR AMAP, 34398 Montpellier, France

{`dimitri.justeau-allaire, philippe.birnbaum`}@cirad.fr

² Institut Agronomique néo-Calédonien (IAC), 98800 Noumea, New Caledonia

³ AMAP, Univ Montpellier, CIRAD, CNRS, INRA, IRD, Montpellier, France

⁴ ORKID, Centre de Génie Industriel, IMT Mines Albi,

Campus Jarlard, 81013 Albi cedex 09, France

`xavier.lorca@mines-albi.fr`

Abstract. The delineation of areas of high ecological or biodiversity value is a priority of any conservation program. However, the selection of optimal areas to be preserved necessarily results from a compromise between the complexity of ecological processes and managers' constraints. Current reserve design models usually focus on few criteria, which often leads to an oversimplification of the underlying conservation issues. This paper shows that Constraint Programming (CP) can be the basis of a more unified, flexible and extensible framework. First, the reserve design problem is formalized. Secondly, the problem is modeled from two different angles by using two graph-based models. Then CP is used to aggregate those models through a unique Constraint Satisfaction Problem. Our model is finally evaluated on a real use case addressing the problem of rainforest fragmentation in New Caledonia, a biodiversity hotspot. Results are promising and highlight challenging perspectives to overtake in future work.

1 Introduction

Human activities are exerting pressure on natural habitats, which generally results in a loss of surface and an increase of fragmentation. As a consequence, many species depending on those habitats are threatened, sometimes with extinction. In this context, it is essential to devote an important part of conservation efforts in the protection of natural habitats through the establishment of nature reserves [1–4]. Designing a reserve system is a difficult process involving a trade-off between the conservation targets and the socioeconomic constraints. This problem is known as the *reserve design problem*. The associated questions are at the crossroad between conservation biology, geography, mathematics, computer science, decision theory and environmental philosophy [5]. In this paper, we focus on the mathematical modeling and the computational solving of the reserve design

problem. From this point of view, it is a decision and/or optimization problem. In almost all cases, the combinatorial complexity justifies the need of a systematic approach based on mathematical modeling and computational tools.

In the literature, two major aspects of the reserve design problem usually stand out: the *feature covering* and the *spatial configuration*. The first is often referred as the reserve (or site) selection problem [6–9]. In extension, we refer to the reserve design problem when spatial attributes are considered [10–14]. Current models usually focus on a few aspects of the problem because: (1) they provide an ad-hoc solution to a specific instance of the problem, or (2) they are limited by the modeling paradigm. However, there is a need for a more unified and flexible framework [15] which, in our opinion and based on our experience in New Caledonia, could help to reduce the gap between computer scientists, conservation scientists, and practitioners.

In this paper, we show how the combination of graph-based models with CP can be the basis of such a framework. After a detailed description of the reserve design problem (Sect. 2), we present two graph-based models (Sect. 3). One model is dedicated to the constraint representation of the features covering issues (Sect. 3.1) and the other one is dedicated to the constraint representation of the spatial issues (Sect. 3.2). We then unify the models throughout a single CP model based on the Choco constraint solver [16] (Sect. 4). Finally, a realistic operational use case on the problem of rainforest fragmentation in New Caledonia is depicted and first results are discussed (Sect. 5).

2 Description of the Problem

The reserve design is a decision and/or optimization problem in the discretized geographical space. Given a set of geographical features (e.g., Fig. 2), we are looking for a reserve system satisfying several criteria, in accordance to conservation targets. In this section, we describe and formalize the problem precisely. We start by defining the characteristics of the problem and then define a set of criteria that can be required for a reserve system.

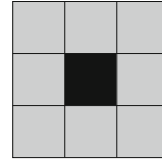
2.1 Characteristics of the Problem – Input Data

The Discretized Geographical Space. The geographical space is tessellated into n granular parcels, which are the decision variables of the problem. Several tessellation methods are possible [17, 18]. The most commonly used is the regular square grid (illustrated in Fig. 1). We choose to this method in this paper.

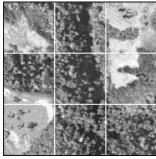
We denote the number of rows by r , the number of columns by c and the set of parcels by \mathcal{P} . We identify a single parcel with the letter i , and index the parcels with integers from 0 to $n - 1$: $\mathcal{P} = \{i \mid i \in \llbracket 0, n \llbracket \}$. While this indexing is not the most convenient for a grid, it has the advantage to be independent of the tessellation method and thus offers extensibility for future work. Finally, we use the 8-connected (cf. Fig. 1) neighborhood to define the adjacency between the parcels, in opposition to the 4-connected neighborhood.

0	1	2
3	4	5
6	7	8

(a) Square grid tessellation



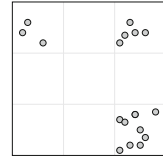
(b) 8-connected neighborhood (gray) of the parcel 4 (black)

Fig. 1. Square grid tessellation and 8-connected neighborhood illustrations.


(a) Feature 0: Dominance of dense rainforest habitat (presence-absence)

0	0	0.99
0.70	0	0.49
0.35	0	0

(b) Feature 1: SDM of an endemic ant species (probability of presence)



(c) Feature 2: Exhaustive inventory of a rare fern species (abundance)

Fig. 2. Three feature examples.

The Environmental Features. The geographical space is characterized by a set of m environmental features. A feature can be anything that can be spatially represented (e.g. the presence of a species, a certain type of habitat, human constructions). We denote by \mathcal{F} the set of environmental features and use the letter j to identify the features: $\mathcal{F} = \{j \mid j \in \llbracket 0, m \rrbracket\}$.

The Values of the Features. To each feature j is associated a set \mathcal{V}_j , representing the available data about j among the parcels: $\mathcal{V}_j = \{v_{ji} \in \mathbb{R}^+ \mid i \in \mathcal{P}\}$. Each $v_{ji} \in \mathcal{V}_j$ corresponds to a value describing the feature j in the parcel i . Three types of data are possible: the presence-absence data, the abundance data and the probability of presence data. An example for each data type is given in Fig. 2, and below is a short description for each of them:

- *Presence-absence*: if j is present in the parcel i , $v_{ji} = 1$, else $v_{ji} = 0$. For each $(j, i) \in \mathcal{F} \times \mathcal{P}$ we then have $v_{ji} \in \{0, 1\}$. The presence-absence data is often used to describe the occurrence distribution of a species or a particular characteristic of the landscape (e.g. forest, savanna, fields, roads, cities).
- *Abundance*: in this case, v_{ji} represents a quantitative value about the feature j in the parcel i (e.g. density of trees per parcel, average annual rainfall). For each $(j, i) \in \mathcal{F} \times \mathcal{P}$ we then have $v_{ji} \in [0, +\infty[$.
- *Probability of presence*: it can be possible to evaluate the probability of presence of a feature j for every parcel in \mathcal{P} . The most common situation is the use of Species Distribution Models (SDMs), that are able to combine observations of a species with environmental data to predict its spatial distribution [19, 20]. With probability of presence data, for each $(j, i) \in \mathcal{F} \times \mathcal{P}$ we have $v_{ji} \in [0, 1]$.

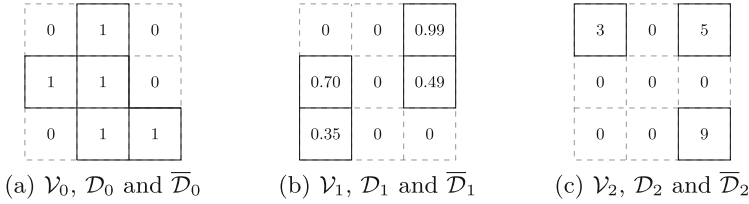


Fig. 3. The values, domains and anti-domains associated with the features in Fig. 2. The domains are represented with solid lines and the anti-domains with dashed lines.

The Domains/Anti-Domains of the Features. As illustrated in Fig. 3, to each feature, is associated a set \mathcal{D}_j and its complement $\overline{\mathcal{D}_j}$. \mathcal{D}_j represents the domain of j , that is, the parcels where j is present or where the probability of presence of j is not null: $\mathcal{D}_j = \{i \in \mathcal{P} \mid v_{ji} > 0\}$. Conversely, $\overline{\mathcal{D}_j}$ represents the anti-domain of j , that is, the parcels where j is not present or where the probability of presence of j is null: $\overline{\mathcal{D}_j} = \{i \in \mathcal{P} \mid v_{ji} = 0\}$.

2.2 The Reserve System – Solution of the Problem

In the first place, we define the terms “parcel” (sometimes called “site” in the literature), “reserve” and “reserve system”. As defined in the previous subsection, a parcel is a granular selection unit of the discretized geographical space. On top of that, a reserve is a set of spatially continuous selected parcels (note that a single selected isolated parcel is a reserve). Finally a reserve system is a set of spatially disjoint reserves (note that a reserve system can be composed of a single reserve). We illustrated the previous definitions in Fig. 4.

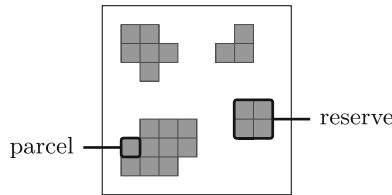


Fig. 4. Illustration of a reserve system, composed of four reserves, themselves made of several adjacent parcels.

Given that, a solution to our problem is a reserve system whose attributes are satisfying a set of criteria, themselves depending on the conservation question. We denote such a reserve system by \mathcal{S} , its number of reserves by n_r and its k^{th} reserve by \mathcal{X}_k : $\mathcal{S} = \{\mathcal{X}_k \subseteq \mathcal{P} \mid k \in \llbracket 0, n_r \rrbracket\}$ (where \mathcal{P} is the set of parcels).

2.3 Required Criteria for a Reserve System

According to the underlying conservation questions, several criteria can be required for a reserve system. We distinguish between the feature covering criteria and the spatial criteria.

Feature Covering Criteria. By providing one of the first formalization of the reserve selection problems, ReVelle et al. [7] introduced three fundamental feature covering criteria.

- *Covered Features.* Among the features that are covered (with certainty) by the reserve system \mathcal{S} , we want a set of mandatory features \mathcal{F}' to be represented (e.g. rare or endangered species).
- *α -Covered Features.* Assuming that the v_{ji} 's are pairwise independent, we want a set of features \mathcal{F}' to be covered by \mathcal{S} with a probability of at least α . This criterion is helpful when probability of presence data is available.
- *k-Redundant Features.* A feature j is k -redundant in the reserve system \mathcal{S} if and only if it is covered (with certainty) by at least k distinct parcels. We want to enforce this property for a set of features \mathcal{F}' (e.g. for increasing the chances of persistence of vulnerable species).

Spatial Criteria. A list of six geometric principles had been defined by Diamond [10] and Williams et al. [11] summarized them into six spatial attributes to take into account when designing a reserve system: the *number of reserves*, the *reserve areas* (by extension we define the *reserve system area*), the *reserve proximity*, the *reserve connectivity*, the *reserve shape* and *core areas and buffer zones*. Here we consider three of those spatial attributes (expressed as criteria) and keep the remaining ones for future work.

- *Number of reserves.* Determining if the best suited is a “single large or several small reserves” (SLOSS), or a “few large or many small reserves” (FLOMS) is a well known debate in ecology [10,21]. The conclusion is that the answer strongly depends on the context, and that flexibility is needed. We therefore want to set a minimum value N_{min} and/or a maximum value N_{max} for the number of reserves.
- *Reserve Areas.* Following the previous criterion, it is also essential to provide control on the reserve areas by setting a minimum area A_{min} and a maximum area A_{max} .
- *Reserve System Area.* It should also be possible to express this criterion on the whole reserve system area, by setting a minimum total area $A_{T_{min}}$ and a maximum total area $A_{T_{max}}$.

3 The Graph-Based Models

In this section, we present two graph-based models. The first one is a resource allocation model that will enable us to express the feature covering criteria in the form of constraints. In the same way, the second one is a spatial model

that will enable us to control the spatial criteria. In both models, each parcel is represented by a vertex. This common characteristic is essential since it is the one that makes the aggregation of the models possible, through a set of appropriate channeling constraints.

3.1 The Resource Allocation Graph

We consider parcels as resources that can be allocated to the conservation of features, then considered as tasks, and thus define the directed graph $G_r = (V_r, A_r)$, also called the *resource allocation graph*. The vertices of G_r are partitioned into three disjoint sets F_r , P_r and $\{s, t\}$. F_r represents the *feature* (or *task*) vertices, P_r represents the *parcel* (or *resource*) vertices, s is the source vertex and t the sink vertex.

$$\begin{aligned} V_r &= F_r \cup P_r \cup \{s, t\}; \\ F_r &= \{f_j \mid j \in \mathcal{F}\}; \\ P_r &= \{p_i \mid i \in \mathcal{P}\}. \end{aligned} \tag{1}$$

Furthermore, using $A_r(X, Y)$ as the notation for the set of all X - Y arcs, we define the arcs of G_r in the following way:

$$A_r = A_r(s, F_r) \cup A_r(F_r, P_r) \cup A_r(P_r, t). \tag{2}$$

$A_r(s, F_r)$ and $A_r(P_r, t)$ are defined such that there is an arc from s to each feature vertex and an arc from each parcel vertex to t :

$$\begin{aligned} A_r(s, F_r) &= \{(s, f_j) \mid f_j \in F_r\}; \\ A_r(P_r, t) &= \{(p_i, t) \mid p_i \in P_r\}. \end{aligned} \tag{3}$$

Moreover, $A_r(F_r, P_r)$ represent the possible allocations between F_r and P_r . More precisely, there is an arc from a feature vertex f_j to a parcel vertex p_i if and only if the feature j is represented in the parcel i , that is $i \in \mathcal{D}_j$. We then have:

$$A_r(F_r, P_r) = \bigcup_{j \in \mathcal{F}} \{(f_j, p_i) \mid i \in \mathcal{D}_j\}. \tag{4}$$

On the arcs of G_r , we define a lower bound (or demand) function $l : A_r \mapsto \mathbb{R}_{\infty}^+$ and an upper bound (or capacity) function $u : A_r \mapsto \mathbb{R}_{\infty}^+$ such that if f is a flow in G_r :

$$\forall a \in A_r, l(a) \leq f(a) \leq u(a). \tag{5}$$

Finally, we define $H_r : \mathcal{P}(P_r) \mapsto \mathcal{P}(V_r) \times \mathcal{P}(A_r)$ that associates to a set $X \subseteq P_r$ the subgraph of G_r induced by $\{s, t\} \cup F_r \cup X$, that is, the resource allocation graph obtained when only considering a subset of parcels. We denote by $V_r[H_r(X)]$ the vertices of $H_r(X)$ and by $A_r[H_r(X)]$ the arcs of $H_r(X)$. An example is provided in Fig. 5.

$$H_r(X) = G_r[\{s, t\} \cup F_r \cup X]. \tag{6}$$

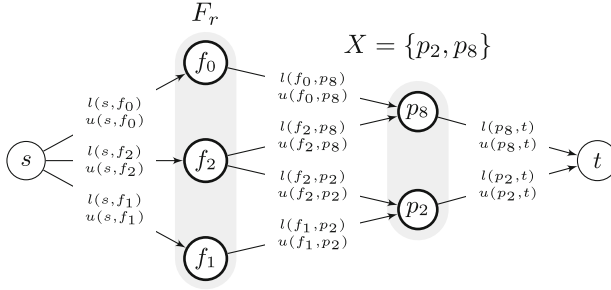


Fig. 5. $H_r(\{p_2, p_8\})$ associated with the example in Fig. 2. Lower and upper bounds are represented on the arcs.

Expressing Feature Covering Criteria as Constraints. From this point, to each feature covering criterion (as defined in the previous section) we associate a constraint that can be applied on the resource allocation model. More precisely, if \mathcal{S} is a reserve system and X_s its associated set of parcel vertices, a criterion is satisfied by \mathcal{S} if and only if its associated constraint is satisfied by $H_r(X_s)$. We express these constraints as flow constraints by defining the value of l and u on certain arcs. When the value of l is not explicitly defined, it is unconstrained and then set to 0. Similarly, u is set to $+\infty$ when its value is not explicitly defined.

Covered Features. In our resource allocation model, we can easily express this criterion as a flow constraint on $H_r(X_s)$.

Constraint 1: Covered Features.

Input parameter(s): A set of features $\mathcal{F}' \subseteq \mathcal{F}$.

The set of features \mathcal{F}' is covered by \mathcal{S} if and only if $H_r(X_s)$ admits a feasible flow f verifying (5) when:

$$\begin{cases} l(s, f_j) = 1, \quad \forall f_j \in \mathcal{F}'_r; \\ l(f_j, p_i) = 1, \quad \forall (f_j, p_i) \in A_r(F_r, P_r); \\ u(f_j, p_i) = 1, \quad \forall (f_j, p_i) \in A_r(F_r, P_r) \text{ such that } v_{ji} \geq 1; \\ u(f_j, p_i) = 0, \quad \forall (f_j, p_i) \in A_r(F_r, P_r) \text{ such that } v_{ji} < 1. \end{cases} \quad (7)$$

α -Covered Features. To express this criterion, we assume that the probabilities of presence v_{ji} are pairwise independent. We then rely on the probability of absence $q_{ji} = (1 - v_{ji})$ and express the constraint as:

$$\forall j \in \mathcal{F}', \quad \prod_{i \in \mathcal{S}} q_{ji} \leq 1 - \alpha. \quad (8)$$

We then express the α -presence constraint in the following way:

Constraint 2: α -Covered Features.

Input parameter(s): A set of features \mathcal{F}' and a real $\alpha \in [0, 1]$.

The set of features \mathcal{F}' is covered by \mathcal{S} with a probability of at least α if and only if $H_r(X_s)$ admits a feasible flow f verifying (5) when:

$$\begin{cases} l(s, f_j) = -\log(1 - \alpha), \forall f_j \in F'_r; \\ u(f_j, p_i) = -\log(q_{ji}), \forall (f_j, p_i) \in A_r(F_r, P_r) \text{ such that } v_{ji} < 1. \end{cases} \quad (9)$$

k-Redundant Features. Since the k-redundancy is actually a generalization of the covering features criterion, we can also express it as a flow constraint on $H_r(X_s)$.

Constraint 3: k-Redundant Features.

Input parameter(s): A set of features \mathcal{F}' and a positive integer k .

The k-redundancy of the set of features \mathcal{F}' in the reserve \mathcal{S} is satisfied if and only if $H_r(X_s)$ admits a feasible flow f verifying (5) when:

$$\begin{cases} l(s, f_j) = k, \forall f_j \in F'_r; \\ l(f_j, p_i) = 1, \forall (f_j, p_i) \in A_r(F_r, P_r); \\ u(f_j, p_i) = 1, \forall (f_j, p_i) \in A_r(F_r, P_r) \text{ such that } v_{ji} \geq 1; \\ u(f_j, p_i) = 0, \forall (f_j, p_i) \in A_r(F_r, P_r) \text{ such that } v_{ji} < 1. \end{cases} \quad (10)$$

3.2 The Spatial Graph

We now define the undirected graph $G_s = (V_s, E_s)$, the *spatial graph*, which is a representation of the discretized geographical space \mathcal{P} (a $r \times c$ regular square grid in our case). Once again, to each parcel i of \mathcal{P} , we associate a vertex p_i , we then have:

$$V_s = \{ p_i \mid i \in \mathcal{P} \}. \quad (11)$$

Moreover, the edges of G_s are defined such that if p_u and p_v are two vertices, there is an edge between p_u to p_v if and only if the parcels u and v are spatially adjacent. The edges of G_s can be partitioned into four disjoint sets: the horizontal

edges (E_H), the vertical edges (E_V), the north-west to south-east diagonal edges ($E_{N_{WSE}}$) and the north-east to south-west diagonal edges ($E_{N_{ESW}}$).

$$\begin{aligned}
 E_s &= E_H \cup E_V \cup E_{N_{WSE}} \cup E_{N_{ESW}}; \\
 E_H &= \{ (p_i, p_{i+1}) \mid i \in \mathcal{P} \wedge \neg(i+1) \equiv 0(c) \}; \\
 E_V &= \{ (p_i, p_{i+c}) \mid i \in \mathcal{P} \wedge i < c(r-1) \}; \\
 E_{N_{WSE}} &= \{ (p_i, p_{i+c+1}) \mid i \in \mathcal{P} \wedge i < c(r-1) \wedge \neg(i+1) \equiv 0(c) \}; \\
 E_{N_{ESW}} &= \{ (p_i, p_{i+c-1}) \mid i \in \mathcal{P} \wedge i < c(r-1) \wedge \neg i \equiv 0(c) \}.
 \end{aligned}
 \tag{12}$$

See Fig. 6 for an illustration of the above equation. Also note that it takes into account the extremal positions of the grid. In fact, the parcels located in the first column are the one whose index is a multiple of c , that is $i \equiv 0(c)$. Moreover, the parcels located in the last column are the ones preceding those that are located in the first column, that is $(i+1) \equiv 0(c)$. Finally, the parcels located in the last line are the ones satisfying $i < c(r-1)$.

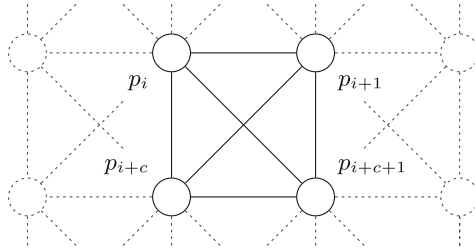


Fig. 6. Illustration of a portion of a spatial graph G_s associated with a $r \times 4$ square grid, using the 8-connectivity neighborhood definition.

Expressing Spatial Criteria as Constraints. Similarly to what had been defined for the resource allocation graph, to a solution \mathcal{S} of the problem we associate $X_s \subseteq V_s$. Moreover, to each reserve $\mathcal{X}_k \in \mathcal{S}$ we associate $X_s(k) \in X_s$, the vertices associated to the parcels of \mathcal{X}_k . We now express each spatial criterion as a constraint that can be applied on $G_s[X_s]$.

Number of Reserves. We easily express this criterion by bounding the number of connected components (NCC, [22–24]) in $G_s[X_s]$.

Constraint 4: Number of Reserves.

Input parameter(s): Two positive integer N_{min} and N_{max} .

Ensuring that the number of reserves in \mathcal{S} is bounded by N_{min} and N_{max} is equivalent to bounding the NCC of $G_s[X_s]$ with N_{min} and N_{max} .

$$N_{min} \leq \text{NCC}(G_s[X_s]) \leq N_{max}.
 \tag{13}$$

Reserve Areas. We express this criterion as a constraint on the number of vertices of the smallest connected component of $G_s[X_s]$ (MIN_NCC, [23–25]) and on the number of vertices of the largest connected component of $G_s[X_s]$ (MAX_NCC, [23–25]).

Constraint 5: Reserve Areas.

Input parameter(s): Two positive integer A_{min} and A_{max} .

Ensuring that the area of every reserve $\mathcal{X}_k \in \mathcal{S}$ is bounded by A_{min} and A_{max} is equivalent to constraining the lower bound of $\text{MIN_NCC}(G_s[X_s])$ to A_{min} and the upper bound of $\text{MAX_NCC}(G_s[X_s])$ to A_{max} .

$$\forall k \in \llbracket 0, n_r \rrbracket, \quad \begin{aligned} \text{MIN_NCC}(G_s[X_s]) &\geq A_{min}; \\ \text{MAX_NCC}(G_s[X_s]) &\leq A_{max}. \end{aligned} \tag{14}$$

Reserve System Area. In the current case of a regular tessellation method, we can control the whole reserve system’s area by bounding the norm of X_s .

Constraint 6: Reserve System Area.

Input parameter(s): Two positive integer $A_{T_{min}}$ and $A_{T_{max}}$.

Ensuring that the total area of the reserve system is bounded by $A_{T_{min}}$ and $A_{T_{max}}$ is equivalent to bounding $|X_s|$.

$$A_{T_{min}} \leq |X_s| \leq A_{T_{max}}. \tag{15}$$

4 The CP Model

In this section we present our CP model for the reserve design problem. For its implementation, we rely on the solver Choco [16] and its extension Choco-graph [26], which provides graph variables and constraints.

The Decision Variables. We naturally model the parcels with a boolean variable array, named `parcels`. If the parcel i is selected in the reserve system, `parcels[i] = 1`, else `parcels[i] = 0`.

```
BoolVar[] parcels = model.boolVarArray("parcels", n);
```

These decision variables are the cornerstone of our CP model because they allow us to aggregate the two models we introduced in the previous section.

The Feature Covering Constraints. Given the particular configuration of the resource allocation graph, we are able to express each feature covering constraint with several local flow conservation inequalities, one for each feature involved in the constraint. Note that we would certainly benefit from the filtering of a global flow constraint [27]. However, there is no such constraint implemented in Choco at the time we are writing this paper. We thus keep this idea for future work.

Constraint 1, Covered Features (7): with local flow conservation inequalities, (7) becomes:

$$\forall j \in \mathcal{F}', \quad \sum_{i=0}^{n-1} b_i \times (v_{ji} \geq 1) \geq 1.$$

Below is the implementation with Choco 4, using the `scalar` constraint.

```
for (int j : featuresToCover) {
    int[] coeffs = Arrays.stream(V[j])
        .mapToInt(v -> (v >= 1) ? 1 : 0)
        .toArray();
    model.scalar(parcel, coeffs, ">=", 1).post();
}
```

Constraint 2, α -Covered Features (9): the coefficients in the `scalar` constraint must be integers. We then retain only two digits of precision for the probabilities of presence. If $\alpha \in [0, 0.99]$ then $-\log(1-\alpha) \in [0, 2]$, moreover, with this precision the order of the smallest variation between two values ($\alpha = 0$ and $\alpha = 0.01$) is 10^{-3} , we thus multiply our local flow inequality by 10^3 in order to stay in the integer domain. If $v_{ji} \geq 1$, we set the flow upper bound to $-10^3 \log(1 - 0.999) = 3000$ as a replacement for $+\infty$. Consequently, we reduce (9) to:

$$\forall j \in \mathcal{F}', \quad \sum_{i=0}^{n-1} b_i \times \min(-10^3 \log(1 - v_{ji}), 3000) \geq -10^3 \log(1 - \alpha).$$

Below is the implementation with Choco 4.

```
for (int j : featuresToCover) {
    int[] coeffs = Arrays.stream(V[j])
        .mapToInt(
            v -> (v >= 1) ? 3000 : (int) (-1000 * Math.log10(1 - v)))
        .toArray();
    int scaled = (int) (-1000 * Math.log10(1 - alpha));
    model.scalar(parcel, coeffs, ">=", scaled).post();
}
```

Constraint 3, k -Redundant Features (10): similarly, we reduce (10) to:

$$\forall j \in \mathcal{F}', \quad \sum_{i=0}^{n-1} b_i \times (v_{ji} \geq 1) \geq k.$$

And implement it the following way with Choco 4:

```
for (int j : featuresToCover) {
    int[] coeffs = Arrays.stream(V[j])
        .mapToInt(v -> (v >= 1) ? 1 : 0)
        .toArray();
    model.scalar(parcel, coeffs, ">=", k).post();
}
```

The Spatial Constraints. We rely on Choco-graph to express the spatial constraints in our CP Model. First, we use a graph variable g to model the reserve system. Its kernel is the empty graph (GLB in the code), and its envelope is G_s (GUB in the code).

```
UndirectedGraph GLB = new UndirectedGraph(model, n, BIPARTITESSET, false);
UndirectedGraph GUB = new UndirectedGraph(model, n, BIPARTITESSET, false);
for (int i = 0; i < n; i++) {
    GUB.addNode(i);
    for (int ii : getNeighbors(i)) {
        GUB.addEdge(i, ii);
    }
}
UndirectedGraphVar g = model.graphVar("g", GLB, GUB);
```

Then, we link the graph variable g with the boolean variables `parcels` using the `nodesChanneling` constraint.

```
model.nodesChanneling(g, parcels).post();
```

We also force the existence of an edge between two selected adjacent parcels through an `edgeChanneling` constraint with a reified `and` constraint between each pair $(i1, i2)$ of adjacent parcels. Doing so, we ensure that every existing edges between two selected vertices are also present in our graph variable.

```
BoolVar forceEdge = model.and(parcel[i1], parcel[i2]).reify();
model.edgeChanneling(g, forceEdge, i1, i2).post();
```

Constraint 4, Number of Reserves (13): we use the `nbConnectedComponents` and the `arithm` constraints.

```
IntVar nbCC = model.intVar("nbCC", Nmin, Nmax);
model.nbConnectedComponents(g, nbCC).post();
```

Constraint 5, Reserve Areas (14): at the time we are writing this paper, there is no constraint in Choco-graph for controlling the `MIN_NCC` and `MAX_NCC` graph properties. We thus implemented the `sizeConnectedComponents`¹ constraint, which allows us to bound `MIN_NCC` and `MAX_NCC`.

¹ <https://gist.github.com/dimitri-justeau/8098af35824bbf8d52ef21282291e621>.

```
model.sizeConnectedComponents(g, Amin, Amax).post();
```

Constraint 6, Reserve System Area (15): we can control the number of vertices of G_s (that is, the number of parcels) through the `nbNodes` graph constraint, or through the `sum` constraint over the decision variables.

```
IntVar nbParcels = model.intVar(Atmin, Amax);
model.nbNodes(g, nbParcels).post(); // Option 1
model.sum(parcel, "=", nbParcels).post(); // Option 2
```

5 Use Case: Rainforest Fragmentation in New Caledonia

New Caledonia is biodiversity hotspot located in the South Pacific, slightly north of the tropic of the Capricorn. The flora of this large archipelago is distinguished by an exceptionally high rate of endemism. Like most of the world’s remaining natural forests, New Caledonian rainforests are endangered with surface loss and fragmentation. A case study had been conducted in the south of New Caledonia in order to highlight “how does forest fragmentation affect tree communities” [28]. We relied on this case study and its associated dataset (up to date) for our use case, and considered the following fictive but realistic operational scenario:

“We want to establish a reserve system in which a pool of endangered species must be present. In addition, most of the other species known in the area must have a high probability to occur, or a high habitat suitability. The reserve system must be mostly covering rainforest areas. Its area and its number of reserves must be limited because of budget limitation. Moreover, each reserve must be large enough to ensure the persistence of the species.”

Note. In this scenario, the objective is to protect both existing and potential rainforest areas. To do so, we relied on SDM layers that were generated with presence-only data and thus produce a score of habitat suitability rather than a standardized probability of presence. A high habitat suitability in a non-rainforest zone can then be interpreted as an adequate zone for recolonization.

5.1 Input Data, Constraints and Parameters

The original dataset consists of the mapping of a 60 km² landscape where 97 tree communities had been sampled in 88 digitized rainforest fragments (forest/non-forest). The dataset gathers 5431 identified trees belonging to 223 species. Moreover, an SDM raster layer was available for 173 of the species [29,30]. Arbitrarily, we considered the 50 species without SDM as the endangered ones. We then prepared this dataset by tessellating the study area into a 46 × 75 regular square grid and by rasterizing the dataset according to this grid. Each parcel then has

an area of about 1.7 ha. Note that we also defined a set of forbidden parcels corresponding to lakes and mining sites.

From this point, we defined a feature for each observed species in the area. When available, we relied on the SDM layer for the feature data (probability of presence data). We forced the values to 1 for the parcels where an observation is available. When no SDM was available, we only relied on the occurrence dataset (presence-absence). We represented the rainforest coverage as a presence-absence feature.

We then applied the *Covered Features* constraint for the set of endangered species, and the α -*Covered Features* constraint for the other species with $\alpha = 0.8$. In order to ensure a minimum rainforest area of 340 ha in the reserve system, we applied the *k-Redundant Features* constraint for the rainforest coverage, with $k = 200$ parcels. Moreover, we enforced the forbidden parcels on the envelope of the graph variable \mathbf{g} . We then set the minimum area of the reserves to $A_{min} = 176$ parcels (about 300 ha) with the *Reserve Areas* constraint. In addition, we limited the reserve system area using the *Reserve System Area* constraint, with $A_{t_{max}} = 589$ parcels (about 1000 ha). According to those restrictive parameters, we allowed the number of reserve to be at most two, using the *Number of Reserves* constraint, with $N_{min} = 1$ and $N_{max} = 2$.

5.2 Questioning and Results

In the first place, the number of reserves and the number of parcels are critical parameters of our use case: the less the better. This is why we started by implementing a search strategy that starts by branching on the lower bound of the `nbCC` variable and continues by selecting the lower bound of the `parcels` variables, sorted in descending order by a score corresponding to the number of features with a value greater than 0.6 (cf. 7). The solver quickly found a solution

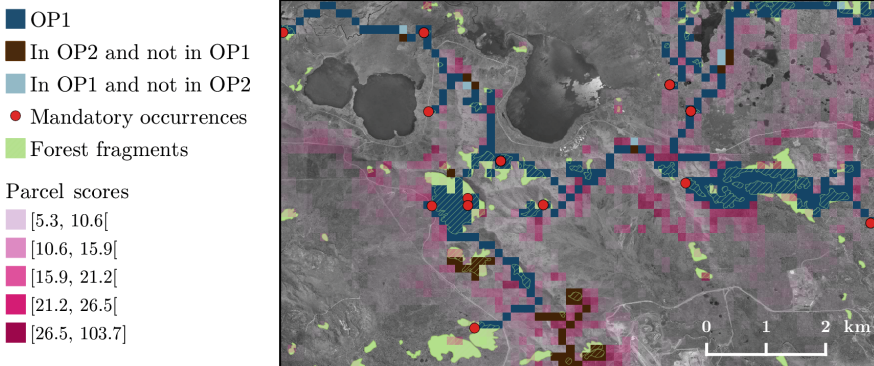


Fig. 7. Mapping of the use case best solutions. The parcel scores correspond to the heuristic score and the mandatory occurrences to rare species observed only once in the study area (they must then be covered by any solution).

Table 1. Use case results: resolution times and solution characteristics. All experiments were run on an Intel Core i5-5200U CPU (2.20 GHz \times 4), with 7.7 GB of RAM.

	DP	OP1	OP2
Resolution time	28 s	3 h 24 m	1 h 5 m
Number of solutions found	1	8	25
Number of reserves	1	1	1
Number of parcels	318	292	328
Number of rainforest parcels	200	200	224

to the decision problem (DP), as summarized in Table 1. Given that, we ran a first optimization problem (OP1) where we tried to minimize the total area of the reserve system, that is the `nbParcels` variable. We limited the computation time to 4 h and retrieved the best solution found, which reduced the total area by 8% in comparison to DP (cf. Table 1). In order to cover more rainforest parcels, we ran a second optimization problem (OP2) in which we forced the `nbParcels` variable to be within 15% of the best value found in OP1, and tried to maximize k (the number of forest parcels), thus defined as an integer variable. After a limited run of 4 h, we could increase the area of rainforest by 12% (cf. Table 1). A mapping of our results is provided in Fig. 7.

6 Conclusion and Challenges

To the best of our knowledge this paper tackles, for the very first time, the reserve design problem from a constraint programming point of view. It is also the first time that a reserve design model integrates such a diversity of constraints, simultaneously involving decisions based on occurrences, SDMs and spatial attributes with an exact approach. Although performance enhancements are needed, the combination of graph-based modeling and constraint programming reveals as a powerful and promising framework for dealing with the reserve design problem.

Based on a challenging use case, our model highlighted a solution compatible with the conservation strategy, namely a trend to link isolated forest patches in order to enhance the functioning of tree communities. However, in this use case we restrained to a binary landscape only composed of forest/non-forest while it is often assumed that a reserve system must include an assemblage of several landscape types. In such a mosaic, an important challenge lays in weighting and balancing the reserve system characteristics and shape in order to maintain (or restore) the functional connectivity inside and between the reserves. In fact, the functional isolation of an habitat leads to a reduction of biological flows, which tends to amplify its spatial isolation. Moreover, since the underlying processes are dynamic, robust solutions must rely both on the current state and future scenarios. It also remains to model the impacts of a reserve system on the off-reserve area, such as the creation of boundaries or enclosed areas.

These elements lead us to identify several lacks and challenges. First of all, the main lack is that Choco solver does not offer an implementation of the flow constraint [31,32]. We will focus on its implementation in future work. Next, a bottleneck in the constraint propagation is the interaction between the constraint on the number and the size of the connected components [25]. We actually treat each one independently but we think that there is a possible enhancement of the filtering by dealing with their interaction. A first challenge for future work concerns our capacity to model constraints on the shape of the reserves by using graph properties, such as graph diameter, in order to design reserves that are compatible with the long-term persistence of species. A second challenge is more oriented to decision making aspects such as identifying key areas that have to be present in any solution. Finally, a last challenge is related to our capacity to take a dual point of view: is it possible to take into account managers' needs on the off-reserve area by adding constraints on the same graph representation?

References

1. Beier, P., Spencer, W., Baldwin, R.F., McRAE, B.H.: Toward best practices for developing regional connectivity maps. *Conserv. Biol.* **25**(5), 879–892 (2011)
2. Baguette, M., Blanchet, S., Legend, D., Stevens, V.M., Turlure, C.: Individual dispersal, landscape connectivity and ecological networks. *Biol. Rev.* **88**(2), 310–326 (2013)
3. Haddad, N.M., et al.: Habitat fragmentation and its lasting impact on Earths ecosystems. *Sci. Adv.* **1**(2), e1500052 (2015)
4. Prendergast, J.R., Quinn, R.M., Lawton, J.H., Eversham, B.C., Gibbons, D.W.: Rare species, the coincidence of diversity hotspots and conservation strategies. *Nature* **365**(6444), 335–337 (1993)
5. Sarkar, S.: Environmental philosophy: from theory to practice. *Stud. History Philos. Sci. Part C Stud. History Philos. Biol. Biomed. Sci.* **45**, 89–91 (2013)
6. Pressey, R.L., Humphries, C.J., Margules, C.R., Vane-Wright, R.I., Williams, P.H.: Beyond opportunism: key principles for systematic reserve selection. *Trends Ecol. Evol.* **8**(4), 124–128 (1993)
7. ReVelle, C.S., Williams, J.C., Boland, J.J.: Counterpart models in facility location science and reserve selection science. *Environ. Model. Assess.* **7**(2), 71–80 (2002)
8. Billionnet, A.: Solving the probabilistic reserve selection problem. *Ecol. Model.* **222**, 546–554 (2011)
9. Watts, M.E., et al.: Marxan with Zones: software for optimal conservation based land- and sea-use zoning. *Environ. Model. Softw.* **24**(12), 1513–1521 (2009)
10. Diamond, J.M.: The island dilemma: lessons of modern biogeographic studies for the design of natural reserves. *Biol. Conserv.* **7**(2), 129–146 (1975)
11. Williams, J.C., ReVelle, C.S., Levin, S.A.: Spatial attributes and reserve design models: a review. *Environ. Model. Assess.* **10**(3), 163–181 (2005)
12. Billionnet, A.: Designing connected and compact nature reserves. *Environ. Model. Assess.* **21**(2), 211–219 (2016)
13. Dilkina, B., et al.: Trade-offs and efficiencies in optimal budget-constrained multi-species corridor networks. *Conserv. Biol.* **31**(1), 192–202 (2017)
14. Jafari, N., Nuse, B.L., Moore, C.T., Dilkina, B., Hepinstall-Cymerman, J.: Achieving full connectivity of sites in the multiperiod reserve network design problem. *Comput. Oper. Res.* **81**, 119–127 (2017)

15. Rodrigues, A.S., Cerdeira, J.O., Gaston, K.J.: Flexibility, efficiency, and accountability: adapting reserve selection algorithms to more complex conservation problems. *Ecography* **23**(5), 565–574 (2000)
16. Prud'homme, C., Fages, J.G., Lorca, X.: Choco Documentation (2017)
17. Sahr, K., White, D., Kimerling, A.J.: Geodesic discrete global grid systems. *Cartography Geogr. Inf. Sci.* **30**(2), 121–134 (2003)
18. Birch, C.P.D., Oom, S.P., Beecham, J.A.: Rectangular and hexagonal grids used for observation, experiment and simulation in ecology. *Ecol. Model.* **206**(3), 347–359 (2007)
19. Guisan, A., Zimmermann, N.E.: Predictive habitat distribution models in ecology. *Ecol. Model.* **135**(2), 147–186 (2000)
20. Elith, J., Leathwick, J.R.: Species distribution models: ecological explanation and prediction across space and time. *Ann. Rev. Ecol. Evol. Syst.* **40**(1), 677–697 (2009)
21. Etienne, R.S., Heesterbeek, J.A.: On optimal size and number of reserves for metapopulation persistence. *J. Theor. Biol.* **203**(1), 33–50 (2000)
22. Doms, G.: The CP(Graph) Computation Domain in Constraint Programming. Ph.D. thesis, UCL - Université Catholique de Louvain (2006)
23. Beldiceanu, N., Carlsson, M., Rampon, J.X., Truchet, C.: Graph Invariants as Necessary Conditions for Global Constraints. Swedish Institute of Computer Science (2005)
24. Beldiceanu, N., Carlsson, M., Demasse, S., Petit, T.: Graph properties based filtering. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 59–74. Springer, Heidelberg (2006). https://doi.org/10.1007/11889205_7
25. Beldiceanu, N., Carlsson, M., Rampon, J.X.: Global Constraint Catalog, 2nd edn., (Revision A). Swedish Institute of Computer Science (2012)
26. Fages, J.G., Prud'homme, C., Lorca, X.: Choco Graph Documentation, February 2018
27. Bockmayr, A., Pizaruk, N., Aggoun, A.: Network flow problems in constraint programming. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 196–210. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45578-7_14
28. Ibanez, T., Hequet, V., Chambrey, C., Jaffré, T., Birnbaum, P.: How does forest fragmentation affect tree communities? a critical case study in the biodiversity hotspot of New Caledonia. *Landscape Ecol.* **32**(8), 1671–1687 (2017)
29. Pouteau, R., et al.: Accounting for the indirect area effect in stacked species distribution models to map species richness in a montane biodiversity hotspot. *Divers. Distrib.* **21**(11), 1329–1338 (2015)
30. Schmitt, S., Pouteau, R., Justeau, D., Boissieu, F., Birnbaum, P.: SSDM: an R package to predict distribution of species richness and composition based on stacked species distribution models. *Methods Ecol. Evol.* **8**(12), 1795–1803 (2017)
31. Steiger, R., van Hoeve, W.J., Szymanek, R.: An efficient generic network flow constraint. In: Proceedings of the 2011 ACM Symposium on Applied Computing, SAC 2011, pp. 893–900. ACM, New York (2011)
32. Downing, N., Feydy, T., Stuckey, P.J.: Explaining flow-based propagation. In: Beldiceanu, N., Jussien, N., Pinson, É. (eds.) CPAIOR 2012. LNCS, vol. 7298, pp. 146–162. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29828-8_10



Self-configuring Cost-Sensitive Hierarchical Clustering with Recourse

Carlos Ansotegui¹, Meinolf Sellmann², and Kevin Tierney³(✉)

¹ University of Lleida, Lleida, Spain
carlos@diei.udl.cat

² General Electric, Global Research Center, Niskayuna, USA
meinolf@ge.com

³ Bielefeld University, Bielefeld, Germany
kevin.tierney@uni-bielefeld.de

Abstract. We revisit algorithm selection for declarative programming solvers. We introduce two main ideas to improve cost-sensitive hierarchical clustering: First, to augment the portfolio builder with a self-configuration component. And second, we propose that the algorithm selector assesses the confidence level of its own prediction, so that a more defensive recourse action can be used to overturn the original recommendation.

1 Introduction

Constraint programming is the quintessential outcome of a consequent pursuit of the declarative programming paradigm. The movement started with logic programming half a century ago. The idea of declarative programming is, in essence, to free the user from the task of directing the process flow of a computer. Instead, the user can declare which properties a solution ought to have, and leaves the task of providing such solutions entirely to the machine. The desired properties can be understood as constraints on the solution, thus leading directly to constraint programming.

If we take the word programming literally, the task of the compiler in constraint programming is to translate the constraint program into a process flow that can be executed on a computer. One of the key aspects of constraint programming research has therefore been the provisioning of ever more efficient *solvers*, i.e., programs that find solutions for constraint programs.

A key observation has been that different algorithms for solving constraint programs¹ work with incomparable efficiency on different problem instances. That is, one solver may work really well on instance A, but takes a very long time to solve instance B, while another solver may solve that same instance B very quickly and yet cannot solve instance A in any practically affordable time.

This work was financially supported in part by TIN2016-76573-C2-2-P.

¹ We use the term constraint program to describe all related declarative programming problems, such as mathematical programming, satisfiability, sat modulo theories, quantified Boolean formulae, and of course actual constraint programming.

Consequently, in a seminal paper, [5] propose the use of an *algorithm portfolio* and show that significant performance gains can be achieved even when combining solvers, which each provide outstanding performance only on rather specific types of instances. Combined in one portfolio, however, robust performance across a wide range of instances can be provided, and by combining these complementary islands of excellence, superior performance is achieved.

In this paper, we revisit algorithm selection by means of cost-sensitive hierarchical clustering (CSHC). We introduce the idea of assessing the self-confidence of the classification model and propose two recourse actions that can be taken when the confidence in the solver recommendation is low. We then show how this technology, which we call Recourse-CSHC, can be customized and tuned automatically for building highly efficient, specific solver portfolios for a variety of declarative programming problems. Experiments on the standard algorithm selection library [2] show that Recourse-CSHC defines a new state of the art in solver portfolio generation.

2 Related Work on Algorithm Selection

The key ingredient for a solver portfolio is an algorithm selector and scheduler (AS2). While not a constraint solver in or by itself, the AS2 handles the important managerial front end to a solver portfolio and decides which solver or solvers are best suited to tackle a given constraint programming instance. It thus turns a set of solvers, each with specific capabilities that do not generalize holistically, into one solver that works robustly across a wide range of instances. In this way it catapults performance to an unknown level.

Since the inception of solver portfolios, our technological ability to provide highly effective AS2s has drastically improved. In the beginning, so-called *empirical hardness models* of solvers were used to predict how well the respective solver would do on a given problem instance [6]. This led to a SAT portfolio called SATzilla that dominated the international SAT competitions for many years [14]. It was then discovered that the forecasting of the actual solver performance on a given instance is an unnecessary and, in fact, detrimental detour. Instead, a classification of which solver to run based on a cost-sensitive nearest neighbor approach turned out to work better [4]. A follow-up paper to this work was [12] where a sequence of cost-sensitive classifiers was used to provide more robust classifications. A revised SATzilla-2012 [15] then proposed to use binary cost-sensitive classification for each pair of solvers. The solver with the most pairwise “wins” is then chosen for the instance at hand.

[4] also introduced a highly efficient method for building solver schedules based on column generation. The fact that schedules can be highly effective had already shown earlier by the CP-Hydra approach [13]. The reason why solver schedules can be effective is precisely because an instance that takes one solver hours or days to solve could be extremely easy to tackle for another solver that provides a solution within seconds.

In [11], a static schedule of solvers is used for 10% of the total available time. Only when no solver in the schedule is able to solve the instance, instance features are computed and a direct, cost-based multi-classification approach called cost-sensitive hierarchical clustering (CSHC) is used to determine which solver will get run for the remaining allowed time.

The current most efficient AS2 for selecting the quickest solvers, according to the Open Algorithm Selection Challenge from 2017 [7, 8], is called SUNNY-fkvar [10]. This approach combines the idea of cost-sensitive nearest-neighbor classification with scheduling. Namely, for the given instance, the k nearest neighbors are computed, and then a scheduler is built at runtime which would solve the most of these neighbors in the shortest time.

3 Cost-Sensitive Hierarchical Clustering

CSHC [11] performs cost-sensitive multi-classification directly, without building individual models for all pairs of solvers as in SATzilla-2012, which creates a significant computational burden when training a new portfolio. The construction works very much like a random forest. For each new tree, only a subset of features are allowed to be used, and a sub-sample (with replacement) is built from the total training set. The training set for the solver selector consists of two tables. The first provides features for each problem instance. The second contains the performance of each solver in the portfolio for each instance.

To build a tree, we first put all sub-sampled instances into the root node. Then, we consider instance features and respective feature values and assess how well the corresponding bi-partition of instances in the node would affect some surrogate objective function. Usually, this is done by iterating over all allowed features for the current tree, and then considering splitting values as induced by the set of instances in the current node.

In regular decision trees, we greedily aim to minimize entropy, which becomes the surrogate when splitting each node recursively. In the case of cost-sensitive hierarchical clustering, we aim to split the set of instances in the current node in such a way that each of the two new subsets can agree on one compromise solver that would solve all instances in the respective subset with best performance. Note that this compromise solver may not be the best solver for any instance in the respective subset of instances.

The process of splitting nodes in the tree is repeated until a stopping criterion is reached. In CSHC there are three: A maximum depth limit, a limit on the minimum number of instances that remain in each node, and a comparison of the performance that is achieved when solving all instances in the current node with the same solver and the performance that is achieved when solving each partition with its respective best solver. Once the performance gained in this way is no longer above a certain threshold, the recursive bi-partition stops.

At runtime, we compute the features of the given instance and run it through each tree until we reach a leaf node. In [11], various methods are introduced for selecting which solver to run. The method we use here is called rank-based selection: For each leaf node, we order all solvers according to their performance on the instances associated with that leaf, thus giving solvers a rank for each leaf node. The solver that has the best sum of all ranks (over all leaves the test instance falls into) is chosen to tackle the given instance.

4 Classification Confidence and Recourse

Whenever a machine learning model – such as the cost-sensitive classification model we reviewed above – makes a recommendation, one might ask how confident we should be in the recommendation. Questions we might ask are: Was the class that is recommended a narrow winner, or did it win by a large margin? In a cost-sensitive setting, we might ask if the performance advantage of the winning class was significant or was it very narrow?

When our confidence in the recommendation is low, the obvious next question to ask is, what can we do about it? Is there more reasoning that we could do to gain more confidence in our recommendation? Is there another recommendation we could make that may not offer much probability for stellar performance but that has a lower probability of causing a high penalty?

4.1 Confidence Assessment

To assess confidence in a recommendation, it is important to understand the nature of cost-sensitive classification first. Just because a second solver might also promise good performance does not mean that we need to be concerned about our choice. There is not just one correct class we have to choose, and even the second-best option may still give very good performance.

To compute a proxy for confidence, we consider the multi-set of training instances that fall into the same leaves as the test instance. For this multi-set, we compute the average solver performance, the standard deviation σ over the various solver performances, and the performance of the best solver on these instances. Then, we compute how many standard deviations above average the best solver actually performs. We use this value as a proxy for confidence.

To give a concrete example, assume there are four solvers, with performances 50, 100, 200 and 250. The average performance is 150, the standard deviation σ is slightly larger than 91. This means, the best solver (with performance 50) is a bit more than one σ better than the average solver.

When the best solver is multiple σ better than the average, we may feel rather confident in our choice. Once the advantage over an average solver becomes slimmer, though, this may indicate that some solvers are really good on some instances closely related to the test instance, while other solvers are good on others. In this case, we should rightfully fear that the test instance is one of those for which the solver recommended by CSHC does not do too well.

4.2 Recourse Actions

If confidence is low, we may want to consider hedging our risks. Choosing any other solver is not going to make us feel any better about our choice, as obviously their advantage over average performance is even less pronounced than the initially recommended solver. Instead, we can consider to invoke a schedule of solvers. We suggest two concrete options for recourse actions.

Static Recourse Schedule: The first recourse action we can take, if we have reason to believe that betting everything on just one solver is too risky, is to invoke a static schedule of solvers that is computed up-front at training time. We run this schedule for a certain percentage of the total time available (e.g., 40%), and for the rest of the time we then invoke the solver recommended by CSHC. The reasoning here is to use a robust schedule that works well across all training instances in case we are not confident with choosing just one solver.

Dynamic Recourse Schedule: The second recourse alternative is to devise a dynamic schedule that would perform well on the multi-set of instances most closely related to the given test instance (defined exactly in the same way as we did to assess CSHC confidence).

5 Configuring the Algorithm Selector

5.1 Importance of Calibration

The methodology outlined above has a number of parameters and, depending on the portfolio at hand, we need to adjust them accordingly to obtain an effective AS2. For example, what is the confidence threshold we should use to switch from running a single recommended solver to a schedule of solvers? Or, consider the time given to a static scheduler that runs for some duration of time at the beginning. Two extreme cases illustrate the problem.

In the first case, assume we have a very long timeout and very few algorithms in the portfolio, say, three. Assume further that each training instance can be solved by only one solver, and that in a fraction of the timeout, say, less than one tenth of the total time allowed. In this case, running each solver for 11% of the time upfront seems like a really good idea, because we can expect that this scheduler will effectively solve most test instances, which we expect to have similar characteristics as our training instances. For the remaining two thirds of the time, we may then run one selected solver.

Contrast this with the case where we again have three solvers in the portfolio, and we find in the training data that each instance can be solved by at least two solvers, but even the fastest needs at least 95% of the timeout. In that case, running a scheduler for one third of the time upfront is obviously a terrible idea.

There are various ways to cope with the need for calibration. One is to let the user set the parameters. Another is to put hard-coded rules into the training process that will set the parameters accordingly. Our solution is to be completely data-driven and exploit an automatic algorithm configurator for this purpose. That is to say, we use the training data, split it into various base and validation sets, then build an AS2 using the information in the base set, for a specific setting of the parameters, and then evaluate the AS2 on the validation set, to assess how well the AS2 can be expected to perform.

This self-configuration is in the same spirit as the AutoFolio approach from [9]. In the latter, the authors train portfolio parameters, for example to determine parameters for pre-schedulers, to select which selection mechanism is used, for example cost-sensitive k-NN as in 3S, or an ensemble of pairwise cost-sensitive random forests as in Satzilla'11, and to set the parameters of the respective selection mechanism. In essence, AutoFolio is thus a portfolio of AS2s. In contrast, we only tune one AS2 approach, namely CSHC, but the latter with the ability to overrule its own recommendation and with an integrated scheduling mechanism.

5.2 Gender-Based Genetic Algorithm Configuration

We use gender-based genetic programming (GGA) for tuning [1]. GGA splits the population of parameterizations in two parts. One part is competitive and fiercely fights for the right of mating, exerting great selection pressure and a massive drive for improving parameterizations very quickly. The other gender is non-competitive and is essential in providing diversity to balance the diversity threatening optimization pressure that the competitive gender exerts on the population. The use of an evolutionary approach offers the additional advantage of being inherently parallel. For more details, we refer to [1].

5.3 Parameters of CSHC-Recourse

There are eleven core parameters, seven that regard the original CSHC, and four new parameters that will guide the recourse actions. Below is a list of the original CSHC parameters, followed by the four parameters for the recourse mechanism:

- A1** Static Schedule Timelimit: Time limit for a short schedule up-front before a long-running solver is chosen to get instances taken care of for which one solver is extremely efficient. This parameter decides what percentage of the total available time will be used upfront. Type: Continuous. Range: 0%–100%.
- A2** Number of Trees: The forest built by CSHC consists of trees. This parameter determines how many. Type: Ordinal. Range 20–500.
- A3** Sub-sampling Size: To build a new tree, we sub-sample (with replacement) the training instances first, to build a new randomized base set for training the tree. This parameter determines the size of the sub-sample as a percentage of the total number of training instances. Type: Continuous. Range: 65%–75%.
- A4** Feature Set Size: To build a new tree, we also reduce the number of features by taking a random subset of features that may be branched on by the tree. This parameter determines the size of this subset as a factor of the square root of the total number of instance features. Type: Continuous. Range: 0.25–4.0.
- A5** Minimum Cluster Size: When building a new tree, we stop the recursive branching when the number of instances in the current node falls below a certain threshold. This parameter determines this threshold. Type: Ordinal. Range: 1–20.
Tree Depth Limit: We also stop the recursion when the tree would exceed a certain depth, which this parameter specifies. Type: Ordinal. Range: 1–100.
- A7** Minimum Improvement Threshold: This parameter determines when to stop the recursion in the tree building procedure, defined as when splitting the current node no longer gives any significant advantage. We consider the cost of solving all instances in the current node with the best solver for this set, and divide it by the sum of the costs to solve both suggested partitions if each were solved by its respective best solver. If this ratio is above the threshold that this parameter represents, we stop the recursion. Type: Continuous. Range: 0.5–1.0.
- B1** Recourse Confidence Threshold: When the confidence falls below this threshold, we trigger a recourse action. The confidence is computed as the number of standard deviations that the performance of the recommended solver lies above the average performance of all solvers. Type: Continuous. Range: 0.0–5.0.
- B2** Dynamic Recourse Threshold: The default recourse is to fall back on a static schedule. However, when the confidence falls even below the threshold given by this parameter, we compute a dynamic schedule optimized for the instances most related to the current test instance. Type: Continuous. Range: 0.0–5.0.
- B3** Static Recourse Schedule Timelimit: When the confidence is below the Recourse Confidence Threshold, but above the Dynamic Recourse Threshold, we run a static schedule that is optimized for all training instances. This parameter specifies the time limit for this schedule as a percentage of the total available time. Only after this schedule is executed, the CSHC recommended solver is run for the remaining available time. Type: Continuous. Range: 20%–100%.
- B4** Dynamic Recourse Schedule Timelimit: When the confidence is below both the Recourse Confidence and Dynamic Recourse Thresholds, we run a schedule of solvers specifically optimized at runtime. This parameter determines the allowed time for this schedule, as a percentage of the total time available. Only after this schedule is executed, the CSHC recommended solver run for the remaining available time. Type: Continuous. Range: 20%–100%.

Table 1. OASC 2017 declarative programming scenarios

Name	#Instances	#Solvers	#Features	VBS	Best single solver
MiniZinc-Time-2016	100	8	95	14	1274
SAT12-All	1614	31	115	376	11691
SAT03-16_Indu	2000	10	483	294	3544
QBF-2016	825	24	46	13	2328
MaxSAT-PMS-2016	601	19	37	42	1220
MaxSAT-WPMS-2016	630	18	37	93	1435
MIP-2016	218	5	143	246	2381

6 Experimental Analysis

In Table 1 we list the declarative programming scenarios from the 2017 Open Algorithm Selection Challenge [7, 8], which covered a wide array of declarative programming domains: Three regard constraint programming and satisfiability, one is on quantified Boolean formulae, two tackle maximum satisfiability, and one is on mixed integer programming. The table shows the characteristics of each benchmark. We use the same train/test split as was given in the algorithm selection challenge, which consists of a specific two-third/one-third split of all instances in the respective benchmark. Self-configuring CSHC-Recourse took between 8 and 12 wall clock hours on a cluster with five 8-core CPUs, depending on the respective OASC benchmark.

All solvers and portfolios are measured by their average PAR10 runtime over all test instances. That is, if a solver or portfolio cannot solve an instance within the benchmark specific time limit, then the penalty for this instance becomes ten times the limit. For all other instances, the cost is exactly the time that was needed to solve the instance.

Table 2. Parameters of CSHC-Recourse on OASC 2017 benchmarks

Name	A1	A2	A3	A4	A5	A6	A7	B1	B2	B3	B4
CSHC default	0.1	500	0.7	0.7	1	100	0.95	N/A	N/A	N/A	N/A
MiniZinc-Time-2016	0.2	100	0.75	0.9	10	5	0.6	4.6	0.8	0.2	0.6
SAT12-All	1.0	200	0.71	0.25	9	4	0.5	3.0	1.0	0.22	0.6
SAT03-16_Indu	0.01	500	0.75	0.22	5	20	0.93	0.24	0.3	2.95	0.5
QBF-2016	0.15	200	0.68	1.2	3	100	0.95	3.66	0.9	0.8	0.8
MaxSAT-PMS-2016	0.1	500	0.74	1.1	2	15	0.9	3.8	0.6	1.5	0.6
MaxSAT-WPMS-2016	0.0	500	0.75	4.0	1	100	0.7	1.9	0.6	0.12	0.8
MIP-2016	0.1	500	0.7	0.7	1	100	1.0	4.98	0.8	3.4	0.4

Table 3. Comparison of various portfolio builders on OASC 2017

Name	Sunny		zilla		CSHC default		CSHC no recourse		CSHC recourse	
	MiniZinc-Time-2016	128	91.0%	473	63.6%	885	30.9%	112	92.2%	126
SAT12-All	4248	65.8%	5072	58.5%	7848	34.0%	7848	34.0%	2734	79.2%
SAT03-16_Indu	3517	0.8%	2974	17.6%	3464	2.5%	2905	19.7%	2836	21.8%
QBF-2016	1011	56.9%	476	80.0%	382	84.1%	382	84.1%	286	88.2%
MaxSAT-PMS-2016	545	57.2%	639	49.3%	646	48.7%	778	34.5%	635	49.6%
MaxSAT-WPMS-2016	213	91.0%	872	41.9%	629	60.1%	495	70.0%	408	76.5%
MIP-2016	1459	43.2%	1365	47.6%	2344	1.8%	2246	6.3%	1299	50.7%
Average		57.99%		51.2%		37.4%		48.7%		65.3%

Table 1 shows the average PAR10 score on the test sets for the virtual best solver (VBS), a perfect oracle that chooses the best solver for each instance. As a further reference point, we show the average PAR10 performance of the single best solver on each respective test set, regardless of whether this solver would also perform best on the respective training set or not. In the competition, each portfolio’s performance is normalized by these two values, with single best solver at 0, and the VBS at 1.

In Table 3, we show the PAR10 and normed performance for Sunny-fkvar [10] (the winner of the OASC 2017 for time tuning tasks as we are considering here), the latest incarnation of zilla [3] (which most closely resembles the cost-sensitive classification in the original CSHC), and three variants of CSHC: The non-tuned default version, the tuned original version (where we split the training set 50 times into 67% base and 33% validation to assess the effect of different parameter settings that are tuned by GGA), and the tuned version of the new CSHC with recourse.

We first observe the benefits of using solver portfolios. Even without consideration of the concrete benchmark, the un-tuned CSHC closes more than one third of the gap between single best solver and VBS, leading to an average solver speedup of a factor slightly larger than 2.

We further see, by comparing CSHC-Default performance with CSHC-No-Recourse, that configuring CSHC is an important step for achieving near state-of-the-art performance. In Table 2 we give an overview of the different parameterizations which shows that the different strategies that the automatic tuner prescribed differ significantly for different benchmarks.

The table also shows the significant gains that can be obtained by falling back to a more conservative recourse action when we do not have a significant front-runner solver. A more detailed look into QBF-2016, e.g., reveals that we override CSHC 194 times. Out of these, 191 overrides make no significant difference, because CSHC would have already solved the instance within the timeout, and so does the recourse scheduler. For 3 instances, however, CSHC would not have solved the instance, while the recourse scheduler managed to do so. On the other hand, on this benchmark we never observe that the recourse scheduler cannot solve the instance in time, while the plain CSHC AS2 can. As the Minizinc-Time-2016 benchmark shows, this is not necessarily the case in general, though.

Overall, we see that self-configuration and recourse scheduling lead to very significant performance improvements. On average, the new methodology closes about two thirds of the gap between single best solver (for the test set!), and a perfect oracle that does not even compute instance features and yet miraculously picks the best solver for each problem instance.

7 Conclusion

We introduced two ideas to improve cost-sensitive hierarchical clustering (CSHC) portfolios: To augment CSHC with a recourse scheduler that kicks in whenever our confidence in CSHC's recommendation is low, and to self-tune CSHC, including its recourse behavior, with respect to the concrete training data at hand. Experiments on seven different benchmarks show that these ideas lead to a new state of the art when it comes to providing cutting edge solver selection and scheduling for a wide range of declarative programming applications.

Acknowledgements. We thank the Paderborn Center for Parallel Computation (PC²) for the use of their high throughput cluster and Marius Lindauer for his kind help with the OASC benchmarks.

References

1. Ansotegui, C., Malitsky, Y., Samulowitz, H., Sellmann, M., Tierney, K.: Model-based genetic algorithms for algorithm configuration. In: IJCAI, pp. 733–739 (2015)
2. Bischl, B., et al.: ASlib: a benchmark library for algorithm selection. *Artif. Intell.* **237**, 41–58 (2016)
3. Cameron, C., Hoos, H.H., Leyton-Brown, K., Hutter, F.: OASC-2017: *zilla submission. In: Open Algorithm Selection Challenge 2017, pp. 15–18 (2017)
4. Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Algorithm selection and scheduling. In: CP, pp. 454–469 (2011)
5. Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., Shoham, Y.: A portfolio approach to algorithm selection. In: IJCAI, pp. 1542–1543 (2003)
6. Leyton-Brown, K., Nudelman, E., Shoham, Y.: Empirical hardness models: methodology and a case study on combinatorial auctions. *J. ACM (JACM)* **56**(4), 22 (2009)
7. Lindauer, M., van Rijn, J., Kotthoff, L.: Open algorithm selection challenge 2017: setup and scenarios. In: Open Algorithm Selection Challenge 2017, pp. 1–7 (2017)
8. Lindauer, M., van Rijn, J.N., Kotthoff, L.: The Algorithm Selection Competition Series 2015–17. ArXiv e-prints, May 2018
9. Lindauer, M., Hutter, F., Hoos, H.H., Schaub, T.: AutoFolio: an automatically configured algorithm selector (extended abstract). In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, 19–25 August 2017, pp. 5025–5029 (2017)
10. Liu, T., Amadini, R., Mauro, J.: Sunny with algorithm configuration. In: Open Algorithm Selection Challenge 2017, pp. 12–14 (2017)
11. Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Algorithm portfolios based on cost-sensitive hierarchical clustering. In: IJCAI, pp. 608–614 (2013)

12. Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Boosting sequential solver portfolios: knowledge sharing and accuracy prediction. In: 7th International Conference on Learning and Intelligent Optimization, LION 7, Catania, Italy, pp. 153–167 (2013)
13. O’Mahony, E., Hebrard, E., Holland, A., Nugent, C., O’Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Irish Conference on Artificial Intelligence and Cognitive Science (2008)
14. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for sat. *JAIR* **32**(1), 565–606 (2008)
15. Xu, L., Hutter, F., Shen, J., Hoos, H., Leyton-Brown, K.: SATzilla2012: improved algorithm selection based on cost-sensitive classification models. In: SAT Competition (2012)

CP and Data Science Track



User's Constraints in Itemset Mining

Christian Bessiere¹, Nadjib Lazaar¹, and Mehdi Maamar²(✉)

¹ LIRMM, University of Montpellier, CNRS, Montpellier, France
{bessiere,lazaar}@lirmm.fr

² CRIL-CNRS, University of Artois, Lens, France
maamar@cril.fr

Abstract. Discovering significant itemsets is one of the fundamental tasks in data mining. It has recently been shown that constraint programming is a flexible way to tackle data mining tasks. With a constraint programming approach, we can easily express and efficiently answer queries with user's constraints on itemsets. However, in many practical cases queries also involve user's constraints on the dataset itself. For instance, in a dataset of purchases, the user may want to know which itemset is frequent and the day at which it is frequent. This paper presents a general constraint programming model able to handle any kind of query on the dataset for itemset mining.

1 Introduction

People have always been interested in analyzing phenomena from data by looking for significant itemsets. This task became easier and accessible for big datasets thanks to computers, and thanks to the development of specialized algorithms for finding frequent/closed/... itemsets. Nevertheless, looking for itemsets with additional user's constraints remains a bottleneck nowadays. According to [11], there are three ways to handle user's constraints in an itemset mining problem. We can use a pre-processing step that restricts the dataset to transactions that satisfy the constraints. Such a technique quickly becomes infeasible when there is a large number of sub-datasets satisfying the user's constraints. We can integrate the filtering of the user's constraints into the specialized data mining process in order to extract only the itemsets satisfying the constraints. Such a technique requires the development of a new algorithm for each new itemset mining problem with user's constraints. We can sometimes use a post-processing step to filter out the itemsets violating the user's constraints. Such a brute-force technique does not apply to all kinds of constraints and is computationally infeasible when the problem without the user's constraints has too many solutions.

In a recent line of work [4–6, 8, 9], constraint programming (CP) has been used as a declarative way to solve data mining problems. Such an approach has not competed yet with state of the art data mining algorithms [10, 12] for simple queries. Nevertheless, the advantage of the CP approach is to be able to add extra (user's) constraints in the model so as to generate only *interesting* itemsets at no other implementation cost.

The weakness of the CP approach is that the kind of user’s constraints that can be expressed has never been clarified. It is easy to post constraints on the kind of itemsets we are interested in but the user may be interested in mining only in some particular transactions of the dataset. For instance, the user may be interested in itemsets that are frequent in transactions corresponding to purchases of less than 100€. None of the current CP approaches is able to catch such kind of constraints. Hence, as with specialized approaches, we need to preprocess the dataset with an ad-hoc algorithm to generate a sub-dataset containing only transaction of less than 100€. It becomes more complex if the user is interested in itemsets that are frequent in transactions corresponding to purchases of a particular sequence of days (such as ‘the week of Christmas’, ‘every Saturday’, etc.). Preprocessing the dataset can lead to the generation of a huge number of sub-datasets, each corresponding to a potential sequence of days.

Another consequence of the lack of clarification of what the CP approach can or cannot do is that some CP models can be flawed by the closedness property. As shown in [1], non-monotone constraints interfere with closedness.¹ For instance, if we post the global constraint for frequent closed itemsets (FCIs) proposed in [6] in a CP model and if in addition we post the constraint specifying that the user is only interested in itemsets of size k , then frequent itemsets of size k having a superset of same frequency will be lost.

In this paper we present a classification of user’s constraints with respect to which itemsets are extracted and from where in the dataset they are extracted. We then propose a generic CP model in which we can capture all these types of user’s constraints. The interaction between constraints and closedness is discussed.

The paper is organized as follows. Section 2 presents the background in data mining and constraint programming. In Sect. 3 we propose a taxonomy of the types of user’s constraints that can be useful in itemset mining. In Sect. 4, we present a CP model able to capture all these user’s constraints. Section 5 gives some case studies that can be expressed using our CP model. Section 6 reports experiments.

2 Background

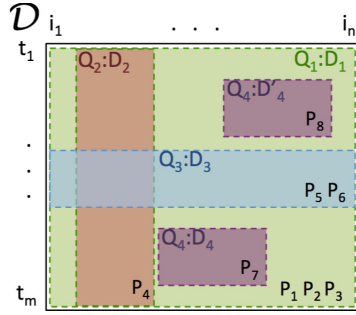
2.1 Itemset Mining

Let $\mathcal{I} = \{1, \dots, n\}$ be a set of n *item* indices and $\mathcal{T} = \{1, \dots, m\}$ a set of m *transaction* indices. An itemset P is a subset of \mathcal{I} . The set of itemsets is $\mathcal{L}_{\mathcal{I}} = 2^{\mathcal{I}} \setminus \{\emptyset\}$. A transactional dataset is a set $\mathcal{D} \subseteq \mathcal{I} \times \mathcal{T}$. A sub-dataset is a subset of \mathcal{D} obtained by removing columns (items) and/or rows (transactions). The set of possible sub-datasets is denoted by $\mathcal{L}_{\mathcal{D}}$. The cover of an itemset P in a sub-dataset D , denoted by $cover(D, P)$, is the set of transactions in D containing P . The frequency of an itemset P in D is the ratio $\frac{|cover(D, P)|}{|cover(D, \emptyset)|}$.

¹ A constraint c is monotone if any superset of an itemset P satisfying c also satisfies c .

Trans.	Items		
	\mathcal{I}_1	\mathcal{I}_2	\mathcal{I}_3
\mathcal{T}_1	t_1	B C	E F G H
	t_2	A	D G
\mathcal{T}_2	t_3	A	C D H
	t_4	A	E F
\mathcal{T}_3	t_5	B	E F
	t_6	B	E F G

(a)



(b)

Fig. 1. (a) A transaction dataset \mathcal{D}_1 . (b) Queries on dataset \mathcal{D} .

An itemset P is closed in a sub-dataset D if and only if the set of items common to all transactions of $cover(D, P)$ is P itself (that is, $\bigcap_{t \in cover(D, P)} t = P$).

Example 1. Let us consider the dataset \mathcal{D}_1 involving 8 items and 6 transactions and displayed in Fig. 1a. The cover $cover(\mathcal{D}_1, BEF)$ of the itemset BEF in \mathcal{D}_1 is equal to $\{t_1, t_5, t_6\}$. The frequency of BEF in \mathcal{D}_1 is thus 50%. The itemset BEF is closed in \mathcal{D}_1 . The itemset BE is not closed in \mathcal{D}_1 because F belongs to all transactions in $cover(\mathcal{D}_1, BE)$.

2.2 Constraint Programming (CP)

A constraint program is defined by a set of variables $X = \{X_1, \dots, X_n\}$, where D_i is the set of values that can be assigned to X_i , and a finite set of constraints \mathcal{C} . Each constraint $C(Y) \in \mathcal{C}$ expresses a relation over a subset Y of variables X . The task is to find assignments $(X_i = d_i)$ with $d_i \in D_i$ for $i = 1, \dots, n$, such that all constraints are satisfied.

2.3 CP Models for Itemset Mining

In [3,8], De Raedt *et al.* have proposed CP4IM, a first CP model for itemset mining. They showed how some constraints (e.g., frequency and closedness) can be formulated using CP. This model uses two sets of Boolean variables: (1) item variables $\{X_1, X_2, \dots, X_n\}$, such that $(X_i = 1)$ if and only if the extracted itemset P contains i ; (2) transaction variables $\{T_1, T_2, \dots, T_m\}$, such that $(T_t = 1)$ if and only if the extracted itemset P is in the transaction t . The relationship between P and T is modeled by m reified n -ary constraints. The minimal frequency constraint and the closedness constraint are also encoded by n -ary and m -ary reified constraints.

Recently, global constraints have been proposed to model and solve efficiently data mining problems. The CLOSEDPATTERN global constraint in [6] compactly encodes both the minimal frequency and the closedness constraints. This global constraint does not use reified constraints. It is defined only on item variables. The filtering algorithm ensures domain consistency in a polynomial time and space complexity. The COVERSIZE global constraint in [9] uses a reversible sparse bitset data structure to compute the subset of transactions that cover an itemset. The filtering algorithm computes a lower and an upper-bound on the frequency.

3 User’s Constraints Taxonomy

For an itemset mining task we aim at extracting all itemsets P of $\mathcal{L}_{\mathcal{I}}$ satisfying a query $Q(P)$ that is a conjunction of (user’s) constraints. The set $Th(Q) = \{P \in \mathcal{L}_{\mathcal{I}} \mid Q(P)\}$ is called *a theory* [7]. Common examples of user’s constraints on extracted itemsets are frequency, closedness, maximality, etc. However, it may be desirable for a user to ask for itemsets extracted from particular parts of the dataset. In the general case, a query predicate, denoted by $Q(D, P)$, is expressed both on the itemsets P it returns and on the sub-datasets $D \in \mathcal{L}_{\mathcal{D}}$ on which it mines. The extracted elements forming a theory are now pairs:

$$Th(Q) = \{(D, P) \mid D \in \mathcal{L}_{\mathcal{D}} \wedge P \in \mathcal{L}_{\mathcal{I}} \wedge Q(D, P)\}.$$

To make the description of our user’s constraints taxonomy less abstract, we suppose a categorization of items and transactions. Items are products belonging to k categories (e.g., food, electronics, cleaning, etc), denoted by $\mathcal{I} = \{\mathcal{I}_1, \dots, \mathcal{I}_k\}$. Transactions are categorized into v categories of customers (e.g., categories based on age/gender criteria), denoted by $\mathcal{T} = \{\mathcal{T}_1, \dots, \mathcal{T}_v\}$. It is important to bear in mind that these categories are just examples provided for illustration purposes. Example 2 presents the running example (with categories) that will be used to illustrate each of the types of user’s constraints we present in this section.

Example 2. Let us consider again the dataset \mathcal{D}_1 displayed in Fig. 1a. For our running example, items belong to three categories $\{A, B\}$, $\{C, D, E\}$ and $\{F, G, H\}$, and transactions belong to three categories $\{t_1, t_2\}$, $\{t_3, t_4\}$ and $\{t_5, t_6\}$.

3.1 User’s Constraints on Itemsets

When the user comes with constraints only on the nature of the itemsets to extract, the query, Q_1 , is equivalent to a standard itemset mining task. We mine on the whole dataset. Figure 1b graphically illustrates this. The itemsets that are solution for Q_1 (i.e., P_1 , P_2 and P_3) are extracted from $D_1 = \mathcal{D}$.

An example of such a query where user’s constraints are expressed only on itemsets is the query Q_1 asking for FCIs:

$$Q_1(D, P) \equiv frequent(D, P, \theta) \wedge closed(D, P)$$

where $frequent(D, P, \theta)$ and $closed(D, P)$ are predicates expressing user's constraints on the frequency (with a minimum frequency θ) and the closedness of an itemset P in D , where D is \mathcal{D} in this case. The query Q_1 on the dataset \mathcal{D}_1 of Fig. 1a with a minimum frequency $\theta \geq 50\%$ returns A , BEF , EF and G as FCIs.

As a second example of such a query on itemsets, the user can ask a query Q'_1 where the extracted itemsets are FCIs and the items are taken from at least lb and at most ub categories:

$$Q'_1(D, P) \equiv Q_1(D, P) \wedge atLeast(P, lb) \wedge atMost(P, ub)$$

where $atLeast(P, lb)$ and $atMost(P, ub)$ are user's constraints ensuring that the itemset P overlaps between lb and ub categories of items. The query Q'_1 on the dataset \mathcal{D}_1 of Fig. 1a with $lb = ub = 2$ and minimum frequency $\theta = 50\%$ only returns EF . It does not return A and G because each of these itemsets belongs to a single category. It does not return BEF because it belongs to three categories.

3.2 User's Constraints on Items

In addition to constraints on itemsets, the user may want to put constraints on the items themselves. Such constraints are constraints on the dataset. They specify on which items/columns the mining will occur. In Fig. 1b, constraints on items lead the query, Q_2 , to mine on the sub-dataset D_2 satisfying constraints on items, from which we extract the itemset P_4 satisfying the constraints on itemsets.

As an example, the user can ask a query Q_2 , where the extracted itemsets are FCIs of sub-datasets containing at least lb_I categories of items and at most ub_I categories:

$$Q_2(D, P) \equiv Q_1(D, P) \wedge atLeastI(D, lb_I) \wedge atMostI(D, ub_I)$$

where $atLeastI(D, lb_I)$ and $atMostI(D, ub_I)$ are user's constraints ensuring that the dataset D contains between lb_I and ub_I categories of items. As opposed to Q_1 and Q'_1 , Q_2 seeks itemsets in sub-datasets satisfying a property on their items. The query Q_2 on the dataset \mathcal{D}_1 of Fig. 1a with $lb_I = ub_I = 2$ and minimum frequency $\theta = 50\%$ returns:

- A , BE and E on $\mathcal{I}_1 + \mathcal{I}_2$,
- A , BF , F and G on $\mathcal{I}_1 + \mathcal{I}_3$,
- EF and G on $\mathcal{I}_2 + \mathcal{I}_3$.

3.3 User's Constraints on Transactions

The user may also want to put constraints on transactions. Such constraints determine on which transactions/rows the mining will occur. In Fig. 1b, constraints on transactions lead the query, Q_3 , to mine on the subset D_3 of transactions from which we extract the itemsets P_5 and P_6 .

As an example, the user can ask a query Q_3 , where the extracted itemsets are FCIs on at least lb_T and at most ub_T categories:

$$Q_3(D, P) \equiv Q_1(D, P) \wedge atLeastT(D, lb_T) \wedge atMostT(D, ub_T)$$

where $atLeastT(D, lb_T)$ and $atMostT(D, ub_T)$ are user's constraints ensuring that the dataset D contains between lb_T and ub_T categories of transactions. The query Q_3 on the dataset \mathcal{D}_1 of Fig. 1a with $lb_T = ub_T = 2$ and minimum frequency $\theta = 50\%$ returns:

- A, AD, CH, EF and G on $\mathcal{T}_1 + \mathcal{T}_2$,
- $BEF, BEFG$ and G on $\mathcal{T}_1 + \mathcal{T}_3$,
- A, BEF and EF on $\mathcal{T}_2 + \mathcal{T}_3$.

3.4 User's Constraints on Items and Transactions

Finally, the user may want to put constraints on both items and transactions. In Fig. 1b, such constraints lead the query, Q_4 , to mine on D_4 and D'_4 from which we extract the itemsets P_7 and P_8 .

The user can ask a query Q_4 , where the extracted itemsets are FCIs of sub-datasets containing at least lb_I and at most ub_I categories of items and at least lb_T and at most ub_T categories of transactions:

$$Q_4(D, P) \equiv Q_2(D, P) \wedge Q_3(D, P)$$

The query Q_4 on the dataset \mathcal{D}_1 of Fig. 1a with $lb_I = ub_I = lb_T = ub_T = 2$ and minimum frequency $\theta = 50\%$ will have to explore nine possible sub-datasets in which to look for frequent closed itemsets:

	$\mathcal{I}_1 + \mathcal{I}_2$	$\mathcal{I}_1 + \mathcal{I}_3$	$\mathcal{I}_2 + \mathcal{I}_3$
$\mathcal{T}_1 + \mathcal{T}_2$	A, AD, C, E	A, F, G, H	CH, D, EF, G
$\mathcal{T}_1 + \mathcal{T}_3$	BE	BF, BFG, G	EF, EFG, G
$\mathcal{T}_2 + \mathcal{T}_3$	A, BE, E	A, BF, F	EF

Q_4 is merely a combination of Q_1 (user's constraints on itemsets), Q_2 (user's constraints on items), and Q_3 (user's constraints on transactions). We presented it to show that our model allows any kind of combinations of user's constraints.

3.5 A Simple Illustration: Where Ferrari Cars Are Frequently Bought?

Consider a dataset of cars purchases in France, where each transaction/purchase also contains items representing the city, the department, and the region where the purchase was performed. (City/department/region is the way France is

administratively organized.) The user may be interested in finding where (city, department or region) more than 10% of the purchases are Ferrari cars. This can be done by the query:

$$RQ(D, P) \equiv frequent(D, P, 10\%) \wedge (Ferrari \in P) \wedge \\ (Reg(D) \vee Dep(D) \vee City(D))$$

where $Reg(D)$, $Dep(D)$ and $City(D)$ are user's constraints ensuring that the dataset D corresponds to one of the administrative entities of France.

4 A General CP Model for Itemset Mining

We present ITEMSET, a CP model for itemset mining taking into account any type of user's constraints presented in Sect. 3.

4.1 Variables

P , T , H and V are Boolean vectors to encode:

- $P = \langle P_1, \dots, P_n \rangle$: the itemset we are looking for. For each item i , the Boolean variable P_i represents whether i is in the extracted itemset.
- $T = \langle T_1, \dots, T_m \rangle$: the transactions that are covered by the extracted itemset.
- $H = \langle H_1, \dots, H_n \rangle$: The items in the sub-dataset where the mining will occur. $H_i = 0$ means that the item/column i is ignored.
- $V = \langle V_1, \dots, V_m \rangle$: The transactions in the sub-dataset where the mining will occur. $V_j = 0$ means that the transaction/row j is ignored.

$\langle H, V \rangle$ circumscribes the sub-dataset used to extract the itemset. The CP solver searches in different sub-datasets, backtracking from a sub-dataset and branching on another. $\langle P, T \rangle$ represents the itemset we are looking for, and its coverage in terms of transactions.

4.2 Constraints

Our generic CP model consists of three sets of constraints:

$$\text{ITEMSET}(P, H, T, V) = \begin{cases} \text{DATASET}(H, V) \\ \text{CHANNELING}(P, H, T, V) \\ \text{MINING}(P, H, T, V) \end{cases}$$

$\text{DATASET}(H, V)$ is the set of constraints that express user's constraints on items (i.e., H) and/or transactions (i.e., V). This set of constraints circumscribes the sub-datasets.

$\text{CHANNELING}(P, H, T, V)$ is the set of channeling constraints that express the relationship between the two sets of variables $\langle P, T \rangle$ and $\langle H, V \rangle$:

$$\begin{aligned} H_i = 0 &\Rightarrow P_i = 0 \\ V_j = 0 &\Rightarrow T_j = 0 \end{aligned}$$

These constraints guarantee that if an item (resp. a transaction) is not part of the mining process, it will not be part of the extracted itemset (resp. the cover set).

$\text{MINING}(P, H, T, V)$ is the set of constraints that express the (user's) constraints on itemsets such as frequency, closedness, size, and more sophisticated user's constraints.

5 ItemSet Model: Cases Studies

In this section, we illustrate our CP model ITEMSET on the queries detailed in Sect. 3. For each query, user's constraints can be written in the DATASET and/or MINING parts of the ITEMSET model. CHANNELING remains unchanged.

Query Q_1

For query Q_1 , we have user's constraints only on itemsets. That is, the mining process will occur on the whole set of transactions. For such a case, we have:

$$\text{DATASET}(H, V) = \begin{cases} \forall i \in \mathcal{I} : H_i = 1 \\ \forall j \in \mathcal{T} : V_j = 1 \end{cases}$$

The user asks for FCIs:

$$\text{MINING}(P, H, T, V) = \begin{cases} \forall j \in \mathcal{T} : T_j = 1 \Leftrightarrow \sum_{i \in \mathcal{I}} P_i (1 - \mathcal{D}_{ij}) = 0 \\ \forall i \in \mathcal{I} : P_i = 1 \Rightarrow \frac{1}{|\mathcal{T}|} \sum_{j \in \mathcal{T}} T_j \mathcal{D}_{ij} \geq \theta \\ \forall i \in \mathcal{I} : P_i = 1 \Leftrightarrow \sum_{j \in \mathcal{T}} T_j (1 - \mathcal{D}_{ij}) = 0 \end{cases}$$

This corresponds to the model presented in [3] and how it can be written in the MINING part of our ITEMSET model. The first constraint represents the coverage constraint, the second is the minimum frequency with respect to a given minimum frequency θ , and the third one expresses the closedness constraint. Note that to obtain an optimal propagation, this part can be replaced by the global constraint CLOSEDPATTERN [6]:

$$\text{MINING}(P, H, T, V) = \text{CLOSEDPATTERN}_\theta(P, T)$$

Query Q'_1

For Q'_1 , we have k item categories. The user asks for FCIs extracted from the whole dataset but the items composing the extracted FCI must belong to at least lb categories and at most ub categories where $lb \leq ub \leq k$. The DATASET part is the same as in the case of Q_1 . The MINING part takes into account the new user's constraint on itemsets:

$$\text{MINING}(P, H, T, V) = \begin{cases} \text{CLOSEDPATTERN}_\theta(P, T) \\ lb \leq \sum_{j=1}^k \max_{i \in \mathcal{I}_j} P_i \leq ub \end{cases}$$

The first constraint is used to extract FCIs. The second constraint holds if and only if the items of the extracted itemset belong to lb to ub categories.

Query Q_2

For Q_2 , the user asks for FCIs not from the whole dataset as in Q_1 and Q'_1 , but from a part of the dataset with lb_I to ub_I categories of items. Such user's constraints on items are expressed in the DATASET part of our model as:

$$\text{DATASET}(H, V) = \begin{cases} lb_I \leq \sum_{j=1}^k \min_{i \in \mathcal{I}_j} H_i = \sum_{j=1}^k \max_{i \in \mathcal{I}_j} H_i \leq ub_I \\ \forall j \in \mathcal{T} : V_j = 1 \end{cases}$$

For each category, the first constraint activates all items or none. The number of categories with their items activated is between lb_I to ub_I . The second constraint activates the whole set of transactions. The MINING part is the almost the same as in the case of Q_1 . The only difference is that we need an adapted version of the $\text{CLOSEDPATTERN}_\theta$ where frequent closed itemsets are mined in the sub-dataset circumscribed by the H and V vectors:

$$\text{MINING}(P, H, T, V) = \text{CLOSEDPATTERN}_\theta(P, H, T, V)$$

Query Q_3

For Q_3 , we have v transaction categories. With Q_3 , the user asks for FCIs not from the whole set of transactions but from at least lb_T and at most ub_T transaction categories. These user's constraints on transactions are written in our model as:

$$\text{DATASET}(H, V) = \begin{cases} \forall i \in \mathcal{I} : H_i = 1 \\ lb_T \leq \sum_{j=1}^v \min_{i \in \mathcal{T}_j} V_i = \sum_{j=1}^v \max_{i \in \mathcal{T}_j} V_i \leq ub_T \end{cases}$$

The first constraint activates the whole set of items. For each category, the second constraint activates all transactions or none. The number of categories with their transactions activated is between lb_T and ub_T . The user asks for CFIs. That is, the MINING part is the same as in the case of Q_2 .

Query Q_4

Q_4 involves the different types of user’s constraints presented in this paper. We have k item categories and v transaction categories. The user asks for FCIs on at least lb_I and at most ub_I categories of products and at least lb_T and at most ub_T categories of customers.

$$\text{DATASET}(H, V) = \begin{cases} lb_I \leq \sum_{j=1}^k \min_{i \in \mathcal{I}_j} H_i = \sum_{j=1}^k \max_{i \in \mathcal{I}_j} H_i \leq ub_I \\ lb_T \leq \sum_{j=1}^v \min_{i \in \mathcal{T}_j} V_i = \sum_{j=1}^v \max_{i \in \mathcal{T}_j} V_i \leq ub_T \end{cases}$$

The first constraint ensures the sub-dataset satisfies the constraints on items (categories activated as a whole and between lb_I and ub_I of them activated). The second constraint ensures the sub-dataset satisfies the constraints on transactions (categories activated as a whole and between lb_T and ub_T of them activated). As we look for FCIs, the MINING part remains the same as in the case of Q_2 and Q_3 .

Query RQ

We illustrate our model on the query presented in Sect. 3.5: *Where Ferrari cars are frequently bought?* To make it simple, suppose that transactions are categorized into r regions $\mathcal{T} = \{\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_r\}$, each region is composed of d departments $\mathcal{T}_i = \{\mathcal{T}_{i:1}, \mathcal{T}_{i:2}, \dots, \mathcal{T}_{i:d}\}$, and each department is composed of c cities $\mathcal{T}_{i:j} = \{\mathcal{T}_{i:j:1}, \mathcal{T}_{i:j:2}, \dots, \mathcal{T}_{i:j:c}\}$. (In the real case, the number of cities per department and departments per region can vary).

The CHANNELING part of the model is the same as in our generic CP model presented in Sect. 4. We need to define the DATASET and the MINING parts for the RQ query. In the following, f refers to the item representing the fact that the brand of the car is Ferrari.

$$\text{DATASET}(H, V) = \begin{cases} \forall i \in \mathcal{I} \setminus \{f\} : H_i = 0 \\ P_f = 1 \\ (1) \vee (2) \vee (3) \end{cases}$$

where (1), (2), and (3) are the constraints specifying that itemsets are extracted from a region, a department, or a city. That is, (1), (2), and (3) are constraints

that we can express as in the second line of $\text{DATASET}(H, V)$ of query Q_3 with $lb_T = ub_T = 1$.²

$$\text{MINING}(P, H, T, V) = \text{frequent}(P, H, T, V, 10\%)$$

where frequent itemsets are mined in the sub-dataset circumscribed by the H and V vectors. The V vector characterizes the places where Ferrari cars are frequently bought. We thus observe that the interesting part of the solutions of this data mining task is more in the value of the V variables than in the P variables.

An Observation on Closedness

As pointed out in the introduction and in [1], closedness can interfere with user's constraints when they are not monotone. Existing CP approaches can lead to the loss of solutions because CP approaches extract closed itemsets that in addition satisfy the user's constraints. (Some itemsets may satisfy the user's constraints whereas not being closed, and closed itemsets may violate a user's constraint.) What we usually want is to extract itemsets that are closed with respect to the user's constraints. Our model allows us to specify on which sub-datasets frequency and closedness will be computed. As a consequence, when these sub-datasets satisfy some non-intersecting properties, we are able to safely combine closedness with non-monotone constraints.

Take for instance Example 1 with a minimum frequency of 50%. If we want itemsets not containing A , F , or G , and closed with respect to these constraints, no system is able to return the only solution BE because it is not closed (BEF has the same frequency). In our model, if we set $H_A = H_F = H_G = 0$, BE is returned as a closed itemset of the sub-dataset $\mathcal{D}_1[BCDE]$. Similarly, if we want itemsets of size 2 and closed with respect to this constraint, all systems will return EF and will miss BE and BF , which are not closed because BEF has the same frequency. In our model, if we set $\sum_1^n H_i = 2$ (in addition to $\sum_1^n P_i = 2$), BE and BF are returned as closed itemsets of the sub-datasets $\mathcal{D}_1[BE]$ and $\mathcal{D}_1[BF]$ respectively. Observe that none of the constraints above are monotone. Unfortunately, not all user's constraints can be combined with closedness in our model. If we want itemsets of size *at most* 2 and closed with respect to this constraint, and if we set $\sum_1^n H_i \leq 2$, A , BE , BF , EF , and G are returned, but also B , E , and F because they are closed for the sub-datasets $\mathcal{D}_1[B]$, $\mathcal{D}_1[E]$ and $\mathcal{D}_1[F]$ respectively, whereas they are not closed for the constraint "itemset of size at most 2"

² A CP expert may object that disjunctions of predicates are not the most efficient way to express constraints. This operational concern can be addressed by capturing (1) \vee (2) \vee (3) into a single global constraint, or by simply adding redundant constraints $V_p = 1 \rightarrow V_r = 1$ for every pair (p, r) of transactions in the same city, and $(V_p = 1 \wedge V_q = 1) \rightarrow V_r = 1$ for every triplet (p, q, r) of transactions in the same region (resp. department) such that p and q are not in the same department (resp. city).

Table 1. Properties of the used datasets

Dataset	$ \mathcal{T} $	$ \mathcal{I} $	$ \overline{\mathcal{T}} $	ρ	Domain
Zoo	101	36	16	44%	Zoo database
Primary	336	31	15	48%	Tumor descriptions
Vote	435	48	16	33%	U.S voting Records
Chess	3196	75	37	49%	Game steps
Mushroom	8124	119	23	19%	Species mushrooms

Primary: Primary-tumor

6 Experimental Evaluation

We made experiments to evaluate the queries Q_1 , Q_2 , Q_3 and Q_4 on our generic CP model ITEMSET for itemset mining.

6.1 Benchmark Datasets

We selected several real-sized datasets from the FIMI repository³ and the CP4IM repository⁴. These datasets have various characteristics representing different application domains. For each dataset, Table 1 reports the number of transactions $|\mathcal{T}|$, the number of items $|\mathcal{I}|$, the average size of transactions $|\overline{\mathcal{T}}|$, its density ρ (i.e., $|\overline{\mathcal{T}}|/|\mathcal{I}|$), and its application domain. The datasets are presented by increasing size.

6.2 Experimental Protocol

We implemented the ITEMSET model presented in Sect. 4. This implementation, named CP-ITEMSET, is in C++, on top of the Gecode solver (www.gecode.org/). The frequency and closedness constraints are performed by a new implementation of the CLOSEDPATTERN global constraint taking into account the variables H , V .⁵ For LCM, the state-of-the-art specialized algorithm for CFIs, we used the publicly available version (<http://research.nii.ac.jp/uno/codes.htm>). All experiments were conducted on an Intel Xeon E5-2665 @2.40 Ghz and a 48 GB RAM with a timeout of 900 s.

In all our experiments we selected a minimum support θ and a minimum size of itemsets k in order to have constrained instances with less than 10 solutions. The reason of this protocol is that a human cannot process millions of solutions. The purpose of user's constraints is to allow the user to focus on interesting solutions only. But, whatever the desired number of solutions, LCM always

³ <http://fimi.ua.ac.be/data/>.

⁴ <https://dtai.cs.kuleuven.be/CP4IM/datasets/>.

⁵ <http://www.lirmm.fr/~lazaar/cpminer.html>.

Table 2. LCM and CP-ITEMSET on Q_1 queries. (Times are in seconds.)

Instances	#FCIs	LCM	CP-ITEMSET
Zoo_50_5	4	0.01	0.01
Primary_60_6	1	0.01	0.01
Vote_50_2	1	0.01	0.01
Mushroom_50_5	8	0.02	0.10
Chess_80_10	4	0.03	0.29

needs to go through a LCM +(preprocessing and/or postprocessing) that generates millions of patterns and then filters out the ‘non-interesting’ ones. Even if LCM is very fast to enumerate a huge number of itemsets, it cannot avoid the combinatorial explosion of all possible sub-datasets.

An instance is defined by the pair frequency/minsize (θ, k) . For example, Zoo_50_5 denotes the instance of the Zoo dataset with a minimum support of 50% and solutions of at least 5 items. Note that the constraint on the size of the itemset is simply added to the MINING part of our ITEMSET model as follows: $minSize_k(P) \equiv \sum_{i \in \mathcal{I}} P_i \geq k$. Note also that such a constraint is integrated in LCM without the need to a post-processing to filter out the undesirable itemsets.

6.3 Query Q_1

Our first experiment compares LCM and CP-ITEMSET on queries of type Q_1 , where we have only user’s constraints on itemsets. We take the Q_1 of the example in Sect. 3.1, where the user asks for FCIs. We added the *minSize* constraint in the MINING part of the ITEMSET model. Table 2 reports the CPU time, in seconds, for each approach on each instance. We also report the total number of FCIs ($\#FCIs \leq 10$) for each instance.

The main observation that we can draw from Table 2 is that, as expected, the specialized algorithm LCM wins on all the instances. However, CP-ITEMSET is quite competitive. LCM is only from 1 to 9 times faster.

6.4 Query Q_2

In addition to user’s constraints on itemsets, in Q_2 the user is able to express constraints on items. We take the Q_2 of the example in Sect. 3.2, where items are in categories and the user asks for FCIs extracted from at least lb_I and at most ub_I categories. We again added the *minSize* constraint.

Table 3 reports the results of the comparison between PP-LCM (LCM with a preprocessing) and CP-ITEMSET on a set of instances. For each instance, we report the number of item categories $\#\mathcal{I}_i$, the used lb_I and ub_I , the total number $\#D$ of sub-datasets satisfying the constraints on items, the number of solutions $\#FCIs$, and the time in seconds. Note that the categories have the same size and for a given $\#\mathcal{I}_i = n'$, and an (lb_I, ub_I) , we have $\#D = \sum_{i=lb_I}^{ub_I} \binom{n'}{i}$.

Table 3. PP-LCM and CP-ITEMSET on Q_2 queries. (Times are in seconds.)

Instance	$\#\mathcal{I}_i$	(lb_I, ub_I)	$\#D$	$\#FCIs$	PP-LCM	CP-ITEMSET
Zoo_80_2	6	(2,3)	35	5	0.58	0.02
	6	(3,4)	35	10	0.62	0.03
Primary_70_5	3	(2,3)	4	2	0.17	0.02
Vote_50_2	6	(2,3)	35	5	0.53	0.02
Mushroom_50_4	17	(2,2)	136	9	5.32	6.14
Mushroom_50_4	17	(2,3)	816	1	41.31	51.04
Chess_70_10	5	(2,3)	20	1	1.24	2.12
Chess_80_10	5	(2,5)	26	5	1.93	3.32
Chess_70_5	15	(2,2)	105	6	2.78	0.97
Chess_80_6	15	(2,3)	560	2	14.57	7.15

It is important to bear in mind for such a query, PP-LCM acts in two steps: (i) pre-processing generating all possible sub-datasets with respect to the user's constraints on items; (ii) run LCM on each sub-dataset. The first step can be very expensive in terms of memory consumption because the space complexity of generating all sub-datasets is in $O(n' \times n \times m)$, where n' is the number of item categories, and n and m the number of items and transaction.

In Table 3 we observe that CP-ITEMSET outperforms PP-LCM on 6 instances out of 10.

6.5 Query Q_3

Let us now present our experiments on queries of type Q_3 where we have user's constraints on itemsets and transactions. We take the Q_3 of the example in Sect. 3.3 where transactions are in categories. We added the *minSize* constraint.

Table 4 reports the results of the comparison between PP-LCM and CP-ITEMSET. For each instance, we report the number of transaction categories $\#\mathcal{T}_i$, the lower and upper bounds (lb_T, ub_T) on transaction categories, the number of sub-datasets $\#D$, the number of extracted solutions $\#FCIs$ and the time in seconds. Note that for a number of categories $\#\mathcal{T}_i = m'$ and a given (lb_T, ub_T) , we have $\#D = \sum_{i=lb_T}^{ub_T} \binom{m'}{i}$.

For Q_3 , PP-LCM acts again in two steps. The space complexity of the preprocessing step is in $O(m' \times n \times m)$, with m' transaction categories, n items and m transactions.

In Table 4 we observe that CP-ITEMSET is faster than PP-LCM on 6 instances out of 10. CP-ITEMSET wins on instances where $\#D$ is large. On *Vote_80_3* with $\#\mathcal{T}_i = 29$ and $(lb_T, ub_T) = (2, 5)$, PP-LCM reports a timeout whereas CP-ITEMSET solves it in 12 min.

6.6 Query Q_4

Our last experiment is on queries of type Q_4 where the user can put constraints on both items and transactions in addition to the ones on the itemsets themselves. We take the Q_4 of the example in Sect. 3.4 where items and transactions are in categories. We added the *minSize* constraint.

Table 4. PP-LCM and CP-ITEMSET on Q_3 queries. (Times are in seconds.)

Instance	$\#T_i$	(lb_T, ub_T)	$\#D$	$\#FCIs$	PP-LCM	CP-ITEMSET
Zoo_70.10	10	(1,10)	1,023	2	7.95	1.12
Zoo_80.5	10	(2,10)	1,013	8	9.05	1.37
Primary_85.4	7	(2,7)	120	1	1.45	0.25
Vote_70.6	29	(2,3)	4,060	3	37.93	17.95
Vote_80.3	29	(2,4)	27,811	4	324.53	135.53
Vote_80.3	29	(2,5)	146,566	4	TO	739.31
Mushroom_70.12	12	(2,2)	66	3	3.13	24.45
	12	(3,3)	220	2	12.63	87.65
Chess_90.22	34	(2,2)	561	1	8.43	15.10
Chess_90.26	94	(2,2)	4,371	3	49.73	68.82

TO: timeout

Table 5 reports results of the comparison between PP-LCM and CP-ITEMSET acting on different instances. We report the number of uniform categories of items/transactions, the used (lb_I, ub_I) and (lb_T, ub_T) , the number of sub-datasets $\#D$, the number of solutions $\#FCIs$ and the time in seconds. PP-LCM needs

Table 5. LCM and CP-ITEMSET on Q_4 queries. (Times are in seconds.)

Instances	$\#I_i$	$\#T_i$	(lb_I, ub_I)	(lb_T, ub_T)	$\#D$	$\#FCIs$	PP-LCM	CP-ITEMSET
Zoo_70.6	6	10	(2,3)	(2,3)	5,775	8	39.69	1.75
Zoo_50.11	6	10	(3,4)	(3,4)	11,550	9	88.66	3.36
Zoo_85.5	6	10	(2,6)	(2,10)	57,741	8	521.89	31.86
Primary_82.5	3	12	(2,3)	(2,10)	16,280	8	199.58	36.13
Vote_70.6	6	29	(2,3)	(2,3)	142,100	2	TO	118.67
Vote_72.5	8	29	(2,3)	(2,3)	341,040	2	TO	201.79
Mushroom_80.5	17	12	(2,2)	(2,2)	8,976	10	446.42	102.68
Mushroom_82.5	17	12	(2,2)	(3,3)	29,920	7	TO	455.19
Chess_90.16	5	34	(2,3)	(2,2)	11,220	3	286.42	87.22

TO: timeout

to generate all possible sub-datasets $\#D = \sum_{i=lb_T}^{ub_T} \binom{m'}{i} \times \sum_{i=lb_I}^{ub_I} \binom{n'}{i}$, where n', m', n and m are respectively the number of item categories, transaction categories, items and transactions. CP-ITEMSET is able to deal with the different queries Q_4 just by changing the parameters $k, lb_T, ub_T, lb_I, ub_I$, whereas PP-LCM needs a time/memory consuming preprocessing before each query.

We see in Table 5 that CP-ITEMSET significantly outperforms PP-LCM. On the instances where PP-LCM does not report a timeout, CP-ITEMSET is from 4 to more than 26 times faster than PP-LCM. The pre-processing step of PP-LCM can reach 90% of the total time. As $\#D$ grows exponentially, it quickly leads to an infeasible preprocessing step (see the 3 timeout cases of PP-LCM).

7 Related Work

In [3, 8], De Raedt *et al.* proposed CP4IM, a CP model to express constraints in itemset mining. CP4IM is able to express user's constraints on the itemset P that is returned. Hence, CP4IM is able to deal with queries of type Q_1 , in which user's constraints are on itemsets only. However, in CP4IM, the variables T representing transactions are internal variables only used to get the cover of the itemset P that is returned, that is, $T_i = 1$ if and only if the itemset P is covered by transaction i . These T variables are not decision variables that would allow constraining the transactions. Adding user's constraints directly on these variables would generate incorrect models.

MiningZinc is a programming language on top of Minizinc. Several examples of complex data mining queries using MiningZinc are discussed in [2]. However, in these examples, when closedness is required, the user's constraints are monotone, and when the mining is performed on sub-datasets, these sub-datasets are statically defined. If we need the mining process to dynamically specify on which sub-datasets the frequency, closeness, and other properties are computed, we believe that MiningZinc requires to implement a model similar to the one we propose in this paper.

8 Conclusion

We have presented a taxonomy of the different types of user's constraints for itemset mining. Constraints can express properties on the itemsets as well as on the items and transactions that compose the datasets on which to look. We have introduced a generic constraint programming model for itemset mining. We showed how our generic CP model can easily take into account any type of user's constraints. We empirically evaluated our CP model. We have shown that it can handle the different types of constraints on different datasets. The CP approach can find the itemsets satisfying all user's constraints in an efficient way compared to the specialized algorithm LCM, which requires a memory/time consuming preprocessing step.

Acknowledgment. Christian Bessiere was partially supported by the ANR project DEMOGRAPH (ANR-16-CE40-0028). Nadjib Lazaar is supported by the project I3A TRACT (CNRS INSMI INS2I - AMIES - 2018). Mehdi Maamar is supported by the project CPER Data from the region “Hauts-de-France” We thank Yahia Lebbah for the discussions we shared during this work.

References

1. Bonchi, F., Lucchese, C.: On closed constrained frequent pattern mining. In: Proceedings of the 4th IEEE International Conference on Data Mining (ICDM 2004), 1–4 November 2004, Brighton, UK, pp. 35–42 (2004)
2. Guns, T., Dries, A., Nijssen, S., Tack, G., Raedt, L.D.: MiningZinc: a declarative framework for constraint-based mining. *Artif. Intell.* **244**, 6–29 (2017)
3. Guns, T., Nijssen, S., Raedt, L.D.: Itemset mining: a constraint programming perspective. *Artif. Intell.* **175**(12–13), 1951–1983 (2011)
4. Kemmar, A., Lebbah, Y., Loudni, S., Boizumault, P., Charnois, T.: Prefix-projection global constraint and top-k approach for sequential pattern mining. *Constraints* **22**(2), 265–306 (2017)
5. Khiari, M., Boizumault, P., Crémilleux, B.: Constraint programming for mining n-ary patterns. In: Cohen, D. (ed.) CP 2010. LNCS, vol. 6308, pp. 552–567. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15396-9_44
6. Lazaar, N., et al.: A global constraint for closed frequent pattern mining. In: Rueher, M. (ed.) CP 2016. LNCS, vol. 9892, pp. 333–349. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44953-1_22
7. Mannila, H., Toivonen, H.: Levelwise search and borders of theories in knowledge discovery. *Data Min. Knowl. Discov.* **1**(3), 241–258 (1997)
8. Raedt, L.D., Guns, T., Nijssen, S.: Constraint programming for itemset mining. In: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, 24–27 August 2008, pp. 204–212 (2008)
9. Schaus, P., Aoga, J.O.R., Guns, T.: CoverSize: a global constraint for frequency-based itemset mining. In: Beck, J.C. (ed.) CP 2017. LNCS, vol. 10416, pp. 529–546. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66158-2_34
10. Uno, T., Asai, T., Uchida, Y., Arimura, H.: An efficient algorithm for enumerating closed patterns in transaction databases. In: Suzuki, E., Arikawa, S. (eds.) DS 2004. LNCS (LNAI), vol. 3245, pp. 16–31. Springer, Heidelberg (2004)
11. Wojciechowski, M., Zakrzewicz, M.: Dataset filtering techniques in constraint-based frequent pattern mining. In: Hand, D.J., Adams, N.M., Bolton, R.J. (eds.) Pattern Detection and Discovery. LNCS (LNAI), vol. 2447, pp. 77–91. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45728-3_7
12. Zaki, M.J., Hsiao, C.: CHARM: an efficient algorithm for closed itemset mining. In: Proceedings of the Second SIAM International Conference on Data Mining, Arlington, VA, USA, 11–13 April 2002, pp. 457–473 (2002)



On Maximal Frequent Itemsets Mining with Constraints

Said Jabbour¹, Fatima Ezzahra Mana^{1,3}, Imen Ouled Dlala^{1,4},
Badran Raddaoui², and Lakhdar Sais¹✉

¹ CRIL-CNRS, Université d'Artois, 62307 Lens Cedex, France
{dlala,jabbour,sais}@cril.fr

² SAMOVAR, Télécom SudParis, CNRS, Univ. Paris-Saclay, Evry, France
badran.raddaoui@telecom-sudparis.eu

³ INPT, Institut National des Postes et Telecommunications, Rabat, Morocco

⁴ LARODEC, University of Tunis, Tunis, Tunisia

Abstract. Recently, a new declarative mining framework based on constraint programming (CP) and propositional satisfiability (SAT) has been designed to deal with several pattern mining tasks. The itemset mining problem has been modeled using constraints whose models correspond to the patterns to be mined. In this paper, we propose a new propositional satisfiability based approach for mining maximal frequent itemsets that extends the one proposed in [20]. We show that instead of adding constraints to the initial SAT based itemset mining encoding, the maximal itemsets can be obtained by performing clause learning during search. A major strength of our approach rises in the compactness of the proposed encoding and the efficiency of the SAT-based maximal itemsets enumeration derived using blocked clauses. Experimental results on several datasets, show the feasibility and the efficiency of our approach.

1 Introduction

Frequent Itemsets Mining (abbreviated as FIM) is well-known and essential in data mining, knowledge discovery and data analysis. It plays an increasingly important role in a series of data mining applications, such as the discovery of associations rules, correlations, causality, sequential patterns, episodes, partial periodicity, emerging patterns, gradual patterns, and many other important discovery tasks. FIM has applications in various fields and becomes fundamental for data analysis as datasets and datastores are becoming very large. Since the first article of Agrawal [4] on association rules and itemset mining, the huge number of works, challenges, datasets and projects show the actual interest in this problem (see [3, 15, 25, 30] and [29] for a survey).

Unfortunately, mining only frequent itemsets generates an overwhelming number of patterns, from which it is difficult to retrieve useful informations. Consequently, for practical data mining, it is important to reduce the size of the output by exploiting the structure of the itemsets data. A well-known condensed representation is the *closed sets* [26, 32]; an itemset is closed if it has no

superset with the same frequency. Nevertheless, in many applications, especially in dense data, the set of all closed itemsets remains too large [13]. One of the most known recourse is then to mine the so-called *maximal* itemsets, where an itemset is maximal frequent if it has no superset that is frequent. So, maximal frequent itemsets is a subset of closed frequent itemsets.

In this paper we introduce SATMax, a new algorithm that makes an original use of SAT solvers for efficiently enumerating all maximal patterns embedded in a transaction database. Technically, the idea is to represent a maximal frequent itemset mining task as a propositional formula such that each of its models corresponds to a maximal pattern of interest. The main argument for this encoding is that it allows us to incorporate domain knowledge in the mining process in an easy and flexible manner, among which the maximal constraint, without presupposing deep insights into the mining mechanism. We address this issue by means of propositional satisfiability solving, a prime technology for knowledge representation and reasoning. This extends earlier results in the application of CP and SAT formalisms to data mining, by allowing to deal with optimization problems. SATMax uses a number of optimizations to efficiently prune away a large portion of the search space. It uses a novel progressive focusing technique to eliminate non-maximal itemsets and exploits blocking clauses for fast frequency checking. We conduct an extensive comparative experimental evaluation of SATMax against DMCP [24] a declarative state-of-the-art maximal itemsets mining approach and Eclat [5] a specialized algorithm.

2 Related Works

In the literature, various proposals have been introduced to mine maximal frequent itemsets from a database of transactions. Many of these existing algorithms are based on the enumeration of frequent itemsets. In [22], Bayardo proposed the MaxMiner algorithm which extends the Apriori algorithm. MaxMiner employs a breadth-first traversal of the search space to limit the database scanning. Furthermore, it uses a dynamic heuristic to increase the effectiveness of superset-frequency pruning. Later, several other enhancements have been suggested for mining maximal frequent itemsets. Pincer-Search algorithm combined the top-down and bottom-up techniques to discover the maximal frequent itemsets [23]. Agarwal et al. [2] implemented a depth-first search technique with bitmap representation (DepthProject), in which columns denote the items and rows denote the transactions. Like MaxMiner algorithm, the authors used dynamic reordering and look-ahead pruning. A projection mechanism is used to reduce the size of the database. The authors efficiently find the support counts and give a superset of the maximal frequent itemsets. Burdick et al. introduced MAFIA [6], an extension of DepthProject. They used vertical bit-vector data format. Compression and projection on bitmaps are applied to increase the performance of the proposed algorithm. Unlike DepthProject and MaxMiner pruning techniques, MAFIA used Parent Equivalence Pruning. Also, GenMax [13] is a back-track search based algorithm for identifying maximal frequent itemsets from a

transactional database. More specifically, this algorithm integrates numerous optimization techniques to prune the search space including progressive focusing that perform maximality checking and diffset propagation for fast support counting. To search maximal frequent itemsets, SmartMiner [33] records at each step tail information to guide the search for new maximal frequent itemsets. Moreover, Eclat algorithm [5] is proposed to find maximal frequent itemsets in transaction databases. This method carries out a depth first search on the subset lattice and determines the support of itemsets by intersecting transaction lists.

Several recent contributions to pattern mining exploit constraint programming and propositional satisfiability [7, 9, 12, 18, 20, 21, 27]. In this context, Guns et al. [14] studied the problem of mining maximal frequent itemsets using CP formalism. More precisely, the authors show how the typical constraint of maximality used in itemset mining can be formulated for use in CP. Besides, in [28], the authors formulate the problem of maximal frequent itemset mining as the enumeration of a set of models of a constraint network by adding a constraint to force the required models to be maximal.

3 Technical Background

This section introduces the preliminaries related to propositional satisfiability, maximal frequent itemset mining and its associated encoding in propositional logic.

3.1 Propositional Satisfiability (SAT)

We consider a standard propositional logical language \mathcal{L} built on a finite set of Boolean variables p, q, r, \dots and usual connectives $\neg, \vee, \wedge, \rightarrow$ and \leftrightarrow standing for negation, disjunction, conjunction, implication and equivalence connectives, respectively. A literal is either a Boolean variable p or its negation $\neg p$. The two literals p and $\neg p$ are called *complementary*. A clause is a formula that consists of a finite disjunction of literals. A conjunctive normal form formula (abbreviated as CNF) is defined over a set of Boolean variables as a conjunction (also represented as a set) of clauses. Let Φ be a CNF formula. We refer to the set of Boolean variables appearing in Φ as $Var(\Phi)$. Any formula in \mathcal{L} can be represented (while preserving satisfiability) in CNF using a set of clauses interpreted conjunctively.

A truth assignment, or boolean interpretation \mathcal{B} assigns truth values from $\{0, 1\}$ (0 corresponds to *false* and 1 to *true*) to every Boolean variable. An interpretation can be also seen as conjunctions or sets of literals. It is lifted to clauses and CNF formulas of \mathcal{L} following usual compositional rules. A CNF formula Φ is satisfiable when there exists at least one boolean interpretation \mathcal{B} satisfying it, i.e., $\mathcal{B}(\Phi) = 1$. Otherwise, it is unsatisfiable. If \mathcal{B} satisfies a formula Φ , \mathcal{B} is then called a *model* of Φ and is represented by the set of variables that it satisfies. We refer to the set of models of Φ as $\mathcal{M}(\Phi)$.

SAT is the decision problem of determining the satisfiability of a CNF formula, i.e., whether or not there exists a model of all clauses in the CNF.

This well known NP-Complete problem has seen spectacular progress these recent years. Interestingly, state-of-the-art SAT solvers have been shown of practical use, solving real-world instances encoding industrial problems up to millions of variables and clauses. As a consequence, providing SAT encoding for a given problem allows us to benefit from this continuous and spectacular progress.

3.2 Frequent Itemset Mining

We are given a set of distinct items (symbols) denoted as $\Omega = \{a, b, c, \dots\}$. A transaction database \mathcal{D} is a set of transactions $\{t_1, t_2, \dots, t_n\}$ such that each transaction $t_i \in \mathcal{D}$ ($i \in [1..n]$) is a subset of Ω , i.e., $t_i \subseteq \Omega$. Transactions can represent things such as the supermarket items purchased by a customer during a shopping visit, or the characteristics of a person as described by his or her replies in a census questionnaire. For instance, Table 1 gives a transaction dataset containing seven transactions $\{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$ described by five items, which will be used as a running example. Besides, each transaction $t_i \in \mathcal{D}$ ($i \in [1..n]$) has an associated unique identifier i called *TID*. A non-null finite subset of items I of Ω is more succinctly called an *itemset* (or pattern). An itemset with k items is called a k -itemset. The notation $I \subseteq t$ will be used to denote that the itemset I is a subset of the set of items that t contains. For convenience, we will often directly refer to a transaction as the set of items that it contains.

Classical data mining problems are usually concerned with itemsets that frequently occur in a database of transactions. The number of occurrences of an itemset in a database is commonly referred to as the support of this itemset. Informally, the support of an itemset measures how often an itemset X occurs in the database. In other words, the support of an itemset is the number of transactions in which that itemset occurs as a subset.

Definition 1. *Given a transaction database \mathcal{D} and an itemset X , the cover of X in \mathcal{D} , denoted $Cover(X)$, is defined as follows: $\{J \in \mathcal{D} \text{ and } X \subseteq J\}$. The support of X in \mathcal{D} , denoted $Supp(X)$, corresponds to the cardinality of $Cover(X)$, i.e., $Supp(X) = |Cover(X)|$.*

Example 1. Let us consider the transaction database \mathcal{D} stated in Table 1. There are five different items $\{a, b, c, d, e\}$ and seven transactions $\{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$. Then, the support of the itemset $X = abc$ is equal to 2, since X occurs in the transactions t_1 and t_2 .

Given a transaction database \mathcal{D} over Ω and a minimum support threshold set as λ according to users' preference, the problem of finding the complete set of frequent itemset is called the *frequent itemset mining problem* defined as:

$$FI(\mathcal{D}, \lambda) = \{X \subseteq \Omega \mid Supp(X) \geq \lambda\}.$$

Unfortunately, identifying the complete set of frequent itemsets may lead to a huge number of patterns. In order to overcome this problem, the concept of closed itemsets is afterward proposed.

Table 1. A transaction database \mathcal{D}

TID	Itemset
t ₁	a b c d
t ₂	a b c e
t ₃	a e
t ₄	a e d
t ₅	a b
t ₆	b d
t ₇	b e

Definition 2 (Closed Frequent Itemset). Let \mathcal{D} be a transaction database and X an itemset. Then, X is a closed itemset if there exists no itemset X' such that $X \subseteq X'$, and $\forall t \in \mathcal{D}, X \in t \rightarrow X' \in t$.

That is, enumerating all closed itemsets allows us to reduce the size of the output.

Extracting all the elements of $FI(\mathcal{D}, \lambda)$ can be obtained from the closed itemsets by computing their subsets. Then, we have $CFI(\mathcal{D}, \lambda) \subseteq FI(\mathcal{D}, \lambda)$.

Example 2. Let us consider again our example described in Table 1. The set of closed frequent itemsets with the minimal support threshold equal to 2 is: $CFI(\mathcal{D}, 2) = \{a, d, ab, ae, abc\}$.

In order to reduce the large number of extracted closed frequent itemsets, another condensed representation, called maximal frequent itemsets, has been introduced.

Definition 3 (Maximal Frequent Itemset). Let \mathcal{D} be a transaction database over Ω and $X \subseteq \Omega$ an itemset. We say that X is a maximal frequent itemset in \mathcal{D} given a minimum threshold λ , if $X \in FI(\mathcal{D}, \lambda)$, and there exists no other itemset Y s.t. $X \subset Y$ and $Y \in FI(\mathcal{D}, \lambda)$.

That is, if the itemset X is frequent and no superset of X is frequent, then we say that X is a maximal frequent itemset. This condensed representation is the one which store most of the information contained in frequent itemsets using less space.

In this work, we are interested in the problem of mining maximal frequent itemsets, abbreviated as **MFI**. More formally,

$$MFI(\mathcal{D}, \lambda) = \{X \subseteq \Omega \mid Supp(X) \geq \lambda \text{ and } \nexists Y \supset X, \text{ s.t. } Y \in FI(\mathcal{D}, \lambda)\}$$

Given a transaction database \mathcal{D} , it is important to note that the set of maximal frequent itemsets is a subset of frequent closed ones, i.e., $MFI(\mathcal{D}, \lambda) \subseteq CFI(\mathcal{D}, \lambda)$.

3.3 SAT-Based Itemset Mining

This section presents a brief overview of the SAT-based approach for enumerating all frequent itemsets in a transaction database proposed in [16,20]. The authors have shown that such mining task can be encoded as a propositional formula whose models are in bijection with the patterns to be mined.

The basic idea consists in the use of two kinds of propositional variables: the *i*-variable p_a to represent each item $a \in \Omega$, and the *t*-variable q_i to represent each transaction t_i .

Next, the SAT encoding is based on the following three CNF formulas built over the previous propositional variables.

$$\bigwedge_{i=1}^n (\neg q_i \leftrightarrow \bigvee_{a \in \Omega \setminus t_i} p_a) \tag{1}$$

The first constraint (1) allows to model the transaction database and then to catch the itemsets. So, an itemset appears in a transaction t_i (i.e., $q_i = 1$) iff the boolean variables associated to items not involved in t_i are set to false. Notice that the formula $(\neg q_i \leftrightarrow \bigvee_{a \in \Omega \setminus t_i} p_a)$ can be translated into the following CNF formula:

$$\bigwedge_{a \in \Omega \setminus t_i} (\neg q_i \vee \neg p_a) \wedge (q_i \vee \bigvee_{a \in \Omega \setminus t_i} p_a)$$

$$\sum_{i=1}^n q_i \geq \lambda \tag{2}$$

Constraint (2) allows us to consider the itemsets having a support greater than or equal to the minimum threshold λ . This encoding is defined as a 0/1 linear inequality, usually called *cardinality constraint*. Because of the presence of such constraint in several applications, many efficient CNF encodings have been proposed over the years. Mostly, such encodings try to derive the best compact representation while maintaining constraint propagation (e.g. [19]).

$$\bigwedge_{a \in \Omega} ((\bigvee_{a \notin t_i} q_i) \vee p_a) \tag{3}$$

Formula (3) expresses the closure property. Intuitively, if the itemset is involved in all transactions containing the item a , then a must be added to the candidate itemset. In other words, when in all the transactions where a does not appear, the candidate itemset is not included, this implies that the candidate itemset appears only in transactions containing the item a . Consequently, to be closed, the item a must be added to the final candidate itemset.

The main advantage of the SAT-based approach is its ability to easily integrate other user constraints. For instance, enumerating itemsets of size at most k can be expressed by simply adding the linear constraint $\sum_{a \in \Omega} p_a \leq k$.

4 SAT-Based Approach for Efficient MFI Mining

In this section, we introduce our SAT-based formulation that enables us to specify in term of constraints maximal frequent pattern mining problem. Given a transaction database and a user specified threshold value, our goal is to provide a simple and efficient way to model and enumerate all maximal frequent itemsets.

As mentioned in Sect. 3.2, an itemset X is maximal if X is frequent and each superset of X is not frequent. Clearly, this requirement can be expressed by the following constraint:

$$\bigwedge_{a \in \Omega} \neg p_a \rightarrow \left(\sum_{t_i | a \in t_i} q_i < \lambda \right) \tag{4}$$

That is, formula (4) expresses that if the item a is not added to the final candidate itemset X , this means that the occurrence frequency of X in the transactions containing a is lower than the minimum threshold λ . Notice that the constraint (4) represents a conditional cardinality constraint.

Interestingly, we can naturally translate the formula (4) to a *Pseudo Boolean constraint*¹ as follows:

$$\bigwedge_{a \in \Omega} (((\lambda - Supp(\{a\}) - 1) \times \neg p_a + \sum_{t_i | a \in t_i} q_i) < \lambda) \tag{5}$$

In the literature, various approaches proposed different efficient encodings of Pseudo Boolean constraints as CNF formula [1, 11, 31]. This transformation can be useful if the number of items and their associated transactions are small. Unfortunately, it is ineffective for large datasets, since it can lead to large CNF formulas. Indeed, each item will be associated with a Pseudo Boolean constraint (5). Consequently, the weakness of SAT-based approaches resides in the size of the encoding, which for large formulas can outgrow available memory or can make SAT solving otherwise inefficient.

Alternatively, another way to manipulate such Pseudo Boolean constraints is to associate a propagator to each constraint of the form (5), as done in constraint programming. However, this case can be challenging since we have to go through each constraint at each decision to check the satisfiability of such constraints.

In order to avoid the addition of conditional cardinality constraints to our initial encoding, we propose in the sequel an original method that allows to insert additional clauses in an incremental manner, throughout the search process, with the aim of ensuring that the found models correspond exactly to the maximal frequent itemsets of the given transaction database. For this purpose, we consider as our SAT solver a DPLL-like procedure that firstly assigns the i -variables. To illustrate our approach, we assume that the solver assigns the truth value *true* to the i -variables. Let us refer to a given model of the CNF formulas encoding

¹ A pseudo Boolean constraint over boolean variables is defined by $\sum_i c_i.l_i \triangleright k$ where c_i are the coefficients, k an integer constant, l_i are literals and \triangleright is one of the operators $\{=, <, \leq, >, \geq\}$.

the FIM task as \mathcal{B} , and $P(\mathcal{B}) = \{a \mid \mathcal{B}(p_a) = 1\}$ will denote the corresponding frequent itemset. Clearly, the first found model \mathcal{B} corresponds to a maximal frequent itemset $P(\mathcal{B})$. In fact, assigning to true the i -variables is a way to derive a maximal itemset.

Now, in order to discard retrieving a model \mathcal{B}' such that $P(\mathcal{B}') \subset P(\mathcal{B})$, one need to eliminate (or block) all itemsets $X \subset P(\mathcal{B})$. To do so, it is sufficient to add the blocking clause $C = (\bigvee_{a \in \Omega \setminus P(\mathcal{B})} p_a)$ to the original encoding. The solver can then backtrack and explore new search space by performing positive assignment of the i -variables.

So, the main idea of our approach consists in adding blocking clauses every time a model is found. It is worth to remark that such clauses are composed of the literals that are assigned to false under the current assignment. This means that such clauses are false before backtracking. In order to enumerate more effectively the set of all maximal frequent itemsets, one need to take the level of literals of each blocking clause C into account to backtrack at the adequate level. This can be seen as a new form of clause learning.

For real-word problems, the items not taking part in each transaction t_i are generally more numerous than those involved in t_i , i.e., $|t_i| \leq |\Omega \setminus t_i|$. Consequently, each blocking clause C used to discard non-maximal itemsets can be large. For example, let us suppose that the current itemset appears in the transaction t_i . Clearly, the clause C can be written as $C = (\bigvee_{a \in t_i \setminus P(\mathcal{B})} p_a \vee \bigvee_{a \in \Omega \setminus t_i} p_a)$.

On the other hand, using the constraint (1), the size of C can be considerably reduced by rewriting it as $C = (\bigvee_{a \in t_i \setminus P(\mathcal{B})} p_a \vee \neg q_i)$.

Additionally, we can choose the most suitable clause C by choosing the smallest transaction t_i containing $P(\mathcal{B})$. Roughly speaking, the size of the blocking clause C clearly depends on the choice of the transaction t_i .

Example 3. Let us consider the transaction database depicted in Table 1. We further assume a minimum threshold $\lambda = 2$. Now, suppose that the SAT solver chooses the following variables ordering during the search process: $p_a, p_b, p_c, \neg p_d$, and $\neg p_e$ (see the search tree depicted by Fig. 1). Then, a first model $\mathcal{B} = \{p_a, p_b, p_c, \neg p_d, \neg p_e\}$ can be obtained by assigning p_a at level 1, p_b at level 2, and p_c at level 3. Hence, the added blocking clause is $C = (p_d \vee p_e)$. In this case, the solver must backtrack to the level 2 since C becomes falsified in level 3 and causes a conflict.

Next, we show the potential behind using blocking clauses in order to significantly improve the mining efficiency. Let us first remark that the blocking clauses involve positive i -variables. More specifically, let $C = (p_{a_1} \vee \dots \vee p_{a_k})$ be a blocking clause. According to the constraint (1), each item a_i is involved in many negative binary clauses of the form:

$$\bigwedge_{t_j \in \mathcal{D} \mid a_i \notin t_j} (\neg p_{a_i} \vee \neg q_j)$$

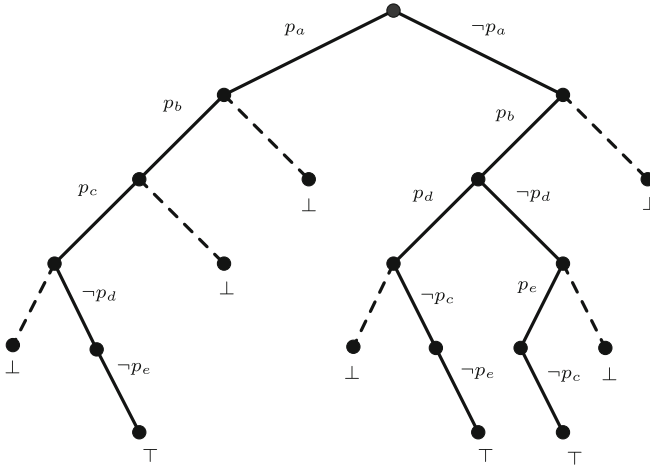


Fig. 1. Search tree of Example 3

Now, one of the most known form of resolution called *hyper binary* resolution [17] can be applied between C and the previous set of negative binary clauses. This gives us the following general constraint of the form:

$$\bigwedge_{i_1 \notin \text{Cover}(\{a_1\}) \dots i_k \notin \text{Cover}(\{a_k\})} (\neg q_{i_1} \vee \dots \vee \neg q_{i_k}) \tag{6}$$

Interestingly enough, the constraint (6) involves only negative clauses (i.e., disjunction of negative literals) over t -variables. These clauses can help improving the efficiency of the frequency constraint (2) by requiring that at least one of the t -variables $\{q_{i_1}, \dots, q_{i_k}\}$ must be false. Unfortunately, when the length of C is large, a great number of clauses can be derived by hyper binary resolution, which leads to excessive space complexity that might slowdown the solver. An alternative is to limit the application of hyper binary resolution to the case where the derived clauses are relevant or of small size. For efficiency reason, we consider the case where the constraint (6) gives rise to a unit clause:

$$\bigwedge_{i \notin \bigcup_{1 \leq j \leq k} \text{Cover}(\{a_j\})} (\neg q_i) \tag{7}$$

Intuitively, the constraint (7) aims to exclude each transaction that does not contain none of the i -variables of the blocking clause C . Indeed, any t -variable that do not belong to $\bigcup_{1 \leq j \leq k} \text{Cover}(\{a_j\})$ must be assigned to false by unit propagation. In fact, the clause $C = (p_{a_1} \vee \dots \vee p_{a_k})$ requires that at least one of its literal must be *true* and consequently the transactions not involving none of items corresponding to literals of C must be assigned to false. Doing so, we are able to effectively improve the resolution process when $\bigcup_{1 \leq j \leq k} \text{Cover}(\{a_j\}) \neq \emptyset$.

Example 4. Let us take the transaction database of Example 1. Assume that the first found model is $\{p_a, p_b, p_c, \neg p_d, \neg p_e\}$. Then, using the blocking clause $(p_d \vee p_e)$ and the constraint (7), we deduce that q_5 and q_6 must be assigned to *false*.

As mentioned previously, our method requires the addition of a blocking clause once a model is found. Then, the maximum number of blocking clauses that can be added is equal to the number of maximal frequent itemsets. Fortunately, even if the number of added blocked clauses might be large, the experiments show that it is feasible in practice.

Let us now present our general SATMax algorithm for SAT-based MFI enumeration task. To summarize the idea behind Algorithm 1, we first encode the closed frequent itemset mining task, then we use a DPLL-like algorithm, while adding a blocking clause each time a model is found. In this way, the models are restricted to those corresponding to maximal frequent itemsets in the given transaction database.

At first, our algorithm encodes the closed frequent itemsets mining task $CFI(\mathcal{D}, \lambda)$ into CNF (line 1). Then, a DPLL procedure is called. It iteratively picks an i -variable (line 22), assigns it to true and performs unit propagation (line 5). Then, we can distinguish two cases: (1) when a conflict occurs (line 6), in this case if the level of the conflict is 0, the enumeration terminates and the set of MFI is returned. Otherwise, a simple backtrack is performed; (2) when there is no conflict. Here, the procedure continues by checking the satisfiability of the frequency constraint (line 15). Then, the same later steps are performed. If all the i -variables are assigned without conflict, then a model is found and a maximal frequent itemset is extracted (line 16). Then, a blocking clause is built (line 17) and analyzed (line 18) to determine the backtracking level. A backtracking is performed accordingly and the procedure loops.

Proposition 1 (Correctness). *SATMax returns all and only the maximal frequent itemsets in the given transaction database.*

5 Experimental Validation

In this section, we evaluate the performance of SATMax. Our mining solver is implemented using a model enumeration MiniSAT solver based on the DPLL (Davis-Putnam-Logemann-Loveland) procedure [8], as described in Algorithm 1. All our experiments were performed on a 2.66 Ghz Intel Xeon quad-core PC with 32 GB of memory, running Ubuntu Linux.

In our SATMax algorithm, the i -variables are firstly assigned. Note that our SAT solver does not branch on t -variables. In fact, the i -variables constitute a strong backdoor, i.e., the t -variables are boolean functions of i -variables (constraint (1)). In our algorithm, each time a model is found, we add a blocking clause and perform a backtracking after analyzing the blocking clause as described in Sect. 4. For the variable ordering heuristic, we follow the one used in [10]. We empirically evaluated our novel approach using different datasets

Algorithm 1. SAT-based MFI enumeration (SATMax)

Input: \mathcal{D} : a transaction database, λ : a minimum support threshold
Output: \mathcal{M} : maximal frequent itemsets of \mathcal{D}

```

1  $\Phi \leftarrow \text{encodeCFI}(\mathcal{D}, \lambda)$ ;
2  $\mathcal{B} = \emptyset$ ;                                     /* Current interpretation */
3  $\mathcal{M} = \emptyset$ ;                               /* Set of Maximal Frequent Itemsets */
4 while (true) do
5    $C = \text{propagate}(\Phi)$ ;
6   if ( $C \neq \text{null}$ ) then
7     if (decisionLevel == 0) then return  $\mathcal{M}$ ;
8      $\text{backtrack}()$ ;
9   else
10    if ( $\sum_{i=1}^n q_i < \lambda$ ) then
11      if (decisionLevel == 0) then return  $\mathcal{M}$ ;
12      else
13         $\text{backtrack}()$ ;
14      end
15      if (Satisfiable( $\Phi$ ) == true) then
16         $\mathcal{M} = \mathcal{M} \cup \{\mathcal{B} \cap \Omega\}$ ;
17         $C \leftarrow \bigvee_{a \in \Omega} \neg p_a \wedge \bigwedge_{a \in \mathcal{B}} p_a$ ;
18         $\Phi \leftarrow \Phi \wedge C$ ;
19         $k \leftarrow \text{analyze}(C)$ ;
20         $\text{backtrackUntil}(k)$ ;
21      else
22         $\text{selectVariable}(\Phi)$ ;
23      end
24    end
25 end

```

coming from FIMI² and CP4IM³ repositories. A CPU time limit is fixed to 1200 seconds per instance. We also use the symbol (-) in Table 2 to mention that the algorithm is not able to scale on the considered dataset under the time limit. In our experiments, we considered different minimum support threshold values. For baseline comparison, we retain the dedicated algorithm Eclat [5] and also DMCP [24], a custom CP bitvector solver. For each method, we report the time needed to enumerate all MFI. Table 2 summarizes our empirical results.

While conducting experiments comparing the three different algorithms, we observed that the performance can vary significantly depending on the dataset characteristics and especially the minimum support threshold values. In many cases our SATMax solver is able to compute all MFI, and improves or meets the dedicated solver Eclat. More interesting enough, SATMax achieves better performances than the CP-based baseline on most considered datasets. In addition, on BMS-WebView-1 we find that SATMax is significantly faster than DMCP for all the

² <http://fimi.ua.ac.be/data/>.

³ <http://dtai.cs.kuleuven.be/CP4IM/datasets/>.

Table 2. Maximal Itemsets: SATMax vs Eclat vs DMCP

instance (#item, #trans, density)	min_supp λ	Eclat time(s)	DMCP time(s)	SATMax time(s)
chess (75, 3196, 49%)	2000	0.11	0.09	0.28
	1500	1.09	1.44	0.52
	1000	5.67	10.56	3.75
	500	33.35	104.56	85.46
connect (129, 67558, 35.62%)	40000	0.29	1.29	5.14
	30000	0.66	2.11	6.06
	20000	3.4	4.65	9.22
	10000	36.83	90.95	22.21
	5000	94.46	–	51.38
kosarak (41267, 990002, 0.01%)	3000	2.52	–	30.00
	2500	3.08	–	32.96
	2000	7.97	–	42.94
	1500	31.52	–	59.03
	1000	67.96	–	100.31
pumsb (2113, 49046, 3%)	40000	0.30	2.92	5.51
	35000	1.05	11.43	6.44
	30000	3.48	32.71	11.23
	25000	89.29	473	49.66
	20000	878.02	–	202.71
retail (16470, 88162, 0.06%)	400	0.29	1.67	1.87
	350	0.26	1.19	2.62
	300	0.33	1.48	2.68
	250	0.58	2.34	2.09
	200	1.19	3.67	4.95
T10I4D100K (870, 100000, 1.0%)	500	0.21	1.77	2.88
	400	0.22	1.87	3.28
	300	0.24	2.48	4.14
	200	0.28	3.63	5.62
	100	0.32	6.44	12.43
T40I10D100K (8942, 100000, 4.31%)	10000	0.45	1.09	2.73
	8000	0.62	0.93	4.03
	6000	1.06	1.68	6.53
	4000	1.85	3.03	9.48
	2000	3.50	7.72	21.53
BMS-WebView-1 (497, 59602, 0.5%)	48	0.07	20.51	2.94
	36	0.22	195.68	5.56
	34	0.28	335.13	7.05
	32	0.36	553.39	7.43
	30	0.49	1049.28	7.14
accidents (468, 340183, 7%)	100000	12.92	33.59	54.73
	80000	46.41	50.22	92.29
	60000	128.85	407.75	174.34
	40000	324.49	–	361.40
	20000	1206.27	–	994.07

considered support threshold values. For the dataset **accidents**, our approach outperforms considerably **DMCP**. Moreover, for some minimum support threshold values, **DMCP** fails to get the maximal frequent itemsets in some instances under the time limit.

Compared with the specialized solver **Eclat**, this latter is generally the best. Nevertheless remarkably, for some dataset and minimum support threshold values our approach outperforms **Eclat**. This is the case for **connect**, **pumsb** and **accidents** when the minimum threshold becomes smaller, **Eclat** becomes worst.

Finally, we run our **SATMax** solver on large problem instances to evaluate its robustness and scalability. For this, we used the **kosarak** instance containing 990002 transactions. We find that **DMCP** for such dataset is not able to scale for all the minimum support threshold values under the time limit. Interestingly, **SATMax** enumerates all MFI for the different support values.

Overall, on maximal frequent itemsets task, the results seem to strongly suggest that **SATMax** is very promising.

Finally in Fig. 2, we compare the number of closed and maximal frequent itemsets for some datasets when the minimum support threshold is varied. We have observed that the number of maximal frequent itemsets is limited relatively to closed ones. For the considered datasets, the maximal frequents itemsets does

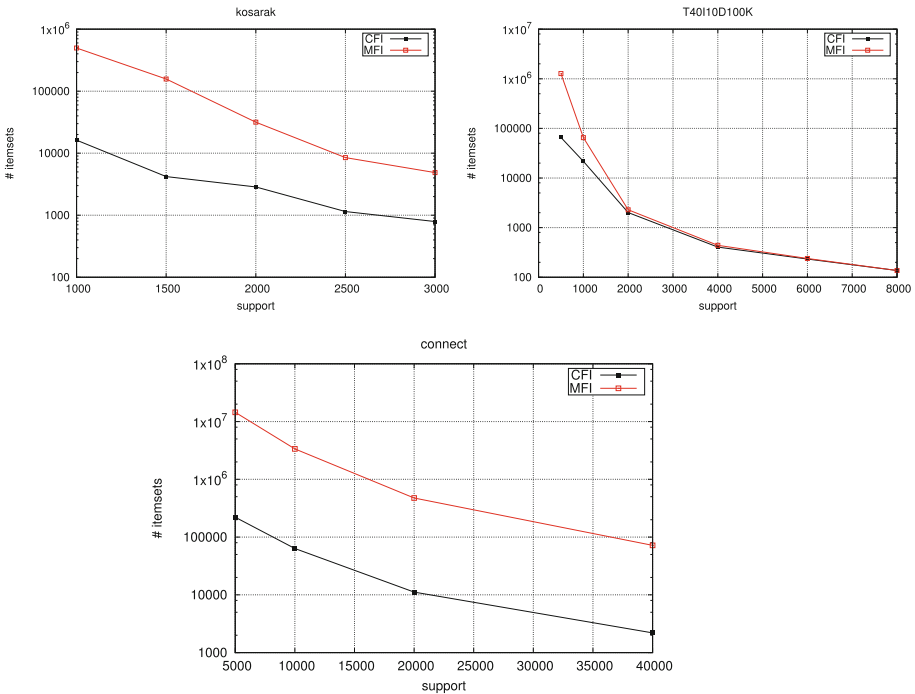


Fig. 2. Frequent itemsets: Closed vs Maximal

not exceed 10% of closed frequent patterns. More interesting enough, this number can be much more limited. This is the case for `connect` and `kosarak` datasets.

6 Conclusion

In this paper, we presented an efficient and scalable approach for computing all maximal frequent itemsets using propositional satisfiability. Based on the closed frequent itemset SAT encoding, an original DPLL-based model enumeration algorithm combined with clauses learning from models allows us to restrict the models to maximal frequent itemsets. Experimental results on several datasets have shown that our approach is very effective compared to Eclat and DMCP, a specialized and CP-based algorithms, respectively. Interestingly, our approach allows us reduce the size of the encoding by avoiding the integration of the maximality constraints.

As a future work, we plan to pursue our investigation in order to improve MFI task using propositional satisfiability. For example, it would be interesting to parallelize our SATMax based approach. Finally, clause learning, an important component for the efficiency of modern SAT solvers, admits several limitations in the context of model enumeration. An important issue is to study how such pivotal mechanism can be efficiently integrated when maximal itemset generation is considered.

References

1. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Mayer-Eichberger, V.: A new look at bdds for pseudo-boolean constraints. *J. Artif. Intell. Res. (JAIR)* **45**, 443–480 (2012)
2. Agarwal, R.C., Aggarwal, C.C., Prasad, V.V.V.: Depth first generation of long patterns. In: *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2000*, pp. 108–118 (2000)
3. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD 1993*, pp. 207–216. ACM, New York (1993)
4. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: *Proceedings of 20th International Conference on Very Large Data Bases VLDB 1994*, pp. 487–499 (1994)
5. Borgelt, C.: Frequent item set mining. Wiley Interdisc. Rev.: Data Min. Knowl. Disc. **2**(6), 437–456 (2012)
6. Burdick, D., Calimlim, M., Gehrke, J.: Mafia: a maximal frequent itemset algorithm for transactional databases. In: *ICDE*, pp. 443–452 (2001)
7. Coquery, E., Jabbour, S., Saïs, L., Salhi, Y.: A sat-based approach for discovering frequent, closed and maximal patterns in a sequence. In: *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, pp. 258–263 (2012)
8. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. *Commun. ACM* **5**, 394–397 (1962)

9. Dlala, I.O., Jabbour, S., Raddaoui, B., Sais, L., Yaghlane, B.B.: A sat-based approach for enumerating interesting patterns from uncertain data. In: Proceedings of 28th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2016, San Jose, CA, USA, pp. 255–262, 6–8 November 2016
10. Dlala, I.O., Jabbour, S., Sais, L., Yaghlane, B.B.: A comparative study of SAT-based itemsets mining. In: Bramer, M., Petridis, M. (eds.) Research and Development in Intelligent Systems XXXIII, pp. 37–52. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47175-4_3
11. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. *JSAT* **2**(1–4), 1–26 (2006)
12. Gebser, M., Guyet, T., Quiniou, R., Romero, J., Schaub, T.: Knowledge-based sequence mining with ASP. In: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9–15 July 2016
13. Gouda, K., Zaki, M.J.: GenMax: an efficient algorithm for mining maximal frequent itemsets. *Data Min. Knowl. Discov.* **11**(3), 223–242 (2005)
14. Guns, T., Nijssen, S., Raedt, L.D.: Itemset mining: a constraint programming perspective. *Artif. Intell.* **175**(12–13), 1951–1983 (2011)
15. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. *SIGMOD Rec.* **29**, 1–12 (2000)
16. Henriques, R., Lynce, I., Manquinho, V.M.: On when and how to use sat to mine frequent itemsets. *CoRR*, abs/1207.6253 (2012)
17. Heule, M., Jarvisalo, M., Biere, A.: Revisiting hyper binary resolution. In: International Conference on Integration of AI and OR Techniques in Constraint Programming, pp. 77–93 (2013)
18. Jabbour, S., Sais, L., Salhi, Y.: Boolean satisfiability for sequence mining. In: Proceedings of 22nd ACM International Conference on Information and Knowledge Management (CIKM 2013), pp. 649–658. ACM (2013)
19. Jabbour, S., Sais, L., Salhi, Y.: A pigeon-hole based encoding of cardinality constraints. *TPLP*, 13(4-5-Online-Supplement) (2013)
20. Jabbour, S., Sais, L., Salhi, Y.: The top-k frequent closed itemset mining using top-k sat problem. In: European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD 2013), pp. 403–418 (2013)
21. Jabbour, S., Sais, L., Salhi, Y.: Mining top-k motifs with a sat-based framework. *Artif. Intell.* **244**, 30–47 (2017)
22. Bayardo, Jr R.J.: Efficiently mining long patterns from databases. In: Proceedings ACM SIGMOD International Conference on Management of Data SIGMOD 1998, Seattle, Washington, USA, pp. 85–93, 2–4 June 1998
23. Lin, D.-I., Kedem, Z.M.: Pincer-search: a new algorithm for discovering the maximum frequent set. In: Schek, H.-J., Alonso, G., Saltor, F., Ramos, I. (eds.) EDBT 1998. LNCS, vol. 1377, pp. 103–119. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0100980>
24. Nijssen, S., Guns, T.: Integrating constraint programming and itemset mining. In: Machine Learning and Knowledge Discovery in Databases, European Conference, ECML PKDD 2010, Proceedings, Part II, Barcelona, Spain, pp. 467–482, 20–24 September 2010
25. Pei, J., Han, J., Lu, H., Nishio, S., Tang, S., Yang, D.: H-mine: hyper-structure mining of frequent patterns in large databases. In: Proceedings IEEE International Conference on Data Mining ICDM 2001, pp. 441–448 (2001)

26. Pei, J., Han, J., Mao, R.: CLOSET: an efficient algorithm for mining frequent closed itemsets. In: 2000 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, pp. 21–30 (2000)
27. Raedt, L.D., Guns, T., Nijssen, S.: Constraint programming for itemset mining. In: ACM SIGKDD, pp. 204–212 (2008)
28. Raedt, L.D., Guns, T., Nijssen, S.: Constraint programming for itemset mining. In: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, pp. 204–212, 24–27 August 2008
29. Tiwari, A., Gupta, R., Agrawal, D.: A survey on frequent pattern mining: current status and challenging issues. *Inform. Technol. J* **9**, 1278–1293 (2010)
30. Uno, T., Kiyomi, M., Arimura, H.: LCM ver. 2: efficient mining algorithms for frequent/closed/maximal itemsets. In: Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations FIMI 2004, Brighton, UK, 1 November 2004
31. Warners, J.P.: A linear-time transformation of linear inequalities into conjunctive normal form. *Inf. Process. Lett.* **68**(2), 63–69 (1998)
32. Zaki, M.J., Hsiao, C.: CHARM: an efficient algorithm for closed itemset mining. In: Proceedings of the Second SIAM International Conference on Data Mining, pp. 457–473 (2002)
33. Zou, Q., Chu, W.W., Lu, B.: Smartminer: a depth first algorithm guided by tail information for mining maximal frequent itemsets. In: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), Maebashi City, Japan, pp. 570–577, 9–12 December 2002



A Parallel SAT-Based Framework for Closed Frequent Itemsets Mining

Imen Ouled Dlala^{1,3}, Said Jabbour¹, Badran Raddaoui², and Lakhdar Sais¹(✉)

¹ CRIL-CNRS, Université d'Artois, 62307 Lens Cedex, France
{dlala,jabbour,sais}@cril.fr

² SAMOVAR, Télécom SudParis, CNRS, Univ. Paris-Saclay, Evry, France
badran.raddaoui@telecom-sudparis.eu

³ LARODEC, University of Tunis, Tunis, Tunisia

Abstract. Constraint programming (CP) and propositional satisfiability (SAT) based framework for modeling and solving pattern mining tasks has gained a considerable audience in recent years. However, this nice declarative and generic framework encounters a scaling problem. The huge size of constraints networks/propositional formulas encoding large datasets is identified as the main bottleneck of most existing approaches. In this paper, we propose a parallel SAT based framework for itemset mining problem to push forward the solving efficiency. The proposed approach is based on a divide-and-conquer paradigm, where the transaction database is partitioned using item-based guiding paths. Such decomposition allows us to derive smaller and independent Boolean formulas that can be solved in parallel. The performance and scalability of the proposed algorithm are evaluated through extensive experiments on several datasets. We demonstrate that our partition-based parallel SAT approach outperforms other CP approaches even in the sequential case, while significantly reducing the performances gap with specialized approaches.

1 Introduction

Frequent itemset mining (abbreviated as FIM) [1, 4] is a fundamental research topic in data mining (DM). It aims to discover important relationships between items in a database arising from numerous applications, ranging from marketing to scientific data analytics. Different kinds of interesting patterns and constraints have been introduced, including closeness and maximality constraints, association rules and its variants [1, 3, 36]. Such progress results in various scalable algorithms designed to deal with specific data mining tasks or constraints. However, such dedicated data mining systems are highly difficult to maintain when additional or combination of constraints came into play. This observation together with the increasing need in terms of user-preference led to the first and seminal paper by De Raedt et al. [28], opening a research avenue on the design of declarative approaches for itemset mining and for pattern mining in general. This new framework offers a flexible representation model that does not require any deep changes in the implementations. In other words, new constraints can be

easily integrated without developing specialized methods. These existing declarative approaches for data mining clearly help pattern selection strategies such as minimizing the redundancy or combining patterns on the basis of their usefulness in a given context.

Encouraged by these promising results, several contributions addressed different data mining tasks using the well-known *constraint programming* (CP), *propositional satisfiability* (SAT) and *answer set programming* (ASP) AI formalisms, such as frequent sequence mining [11, 18, 26], frequent itemset mining [15], closed frequent itemset mining [22, 30], association rules mining [5], clustering [6, 7], and community detection [10, 16]. A high-level language for constraint-based mining, called MiningZinc, is introduced by Guns et al. [12]. It supports a wide variety of different solvers, including DM algorithms and general purpose solvers, and uses a significantly more high-level modeling language. This combination of generic and specialized solvers gives MiningZinc the ability to be both generic and efficient with respect to the performance of the state-of-the-art algorithms on common data mining tasks.

In the first CP-based proposal for itemset mining [13], the problem is expressed through different linear and reified constraints over Boolean variables. This widely adopted model does not exploit some of the well established algorithmic skills from the state-of-the-art specialized algorithms. For the closed itemset mining task, to enhance the efficiency of such CP model, Lazaar et al. [22] proposed a new formulation using a global constraint that incorporates some of the propagation properties borrowed from specialized algorithms. This approach allows to capture the closed frequent itemset mining problem without requiring reified constraints or extra variables, leading to significantly better performances. More recently, Schaus et al. [30] introduced the CoverSize constraint for the itemset mining problem, a global constraint for counting and constraining the number of transactions covered by the itemset decision variables. The authors showed that compared to the ClosedPattern approach [22] using a global constraint for frequent closed itemset mining, both generality and efficiency can be significantly improved. A relation is established between the CoverSize constraint and the well-known table constraint, where the underlined filtering algorithm internally exploits the reversible sparse bitset data structure used for the filtering table. By expressing the size of the cover as a variable, the proposed approach opens up new modelling perspectives. Moreover, a SAT-based framework for enumerating Top- k Motifs in both transaction databases, sequences and sequences of itemsets is proposed in [20]. As a summary, this new framework offers a nice declarative and flexible representation model, while facing a real challenge in terms of scalability. Indeed, all these CP/SAT-based itemset mining approaches flag out good performance on datasets of reasonable size and high support threshold. However, the performance decreases as the database increases in size or the support threshold turns to be low. In this latter case, the size of constraints network/propositional formulas encoding the itemset mining task tends to be huge. Space complexity is clearly identified as

the main bottleneck behind the competitiveness of these new declarative and flexible models w.r.t. specialized data mining approaches.

Some early efforts tried to speed up the specialized pattern mining algorithms by running them in parallel [24, 27, 35]. Several parallel approaches use Spark or MapReduce frameworks [23, 33, 38], through multi-core processors or distributed computing platforms. For an overview of parallel FIM specialized methods, readers are kindly referred to [25]. Moreover, Savasere et al. [29] addressed the problem of generating association rules from large databases by introducing a new algorithm which divides the database into a number of non-overlapping partitions. The partitions are considered one at a time and all large itemsets for that partition are generated.

In spite of the significance of the declarative framework for frequent pattern mining, and despite of their main scalability bottleneck, no advances have been made on parallelizing CP/SAT-based pattern mining approaches. To avoid the generation of a single huge propositional formula encoding the whole transaction database, in [19], the authors proposed an incremental approach allowing to partition the whole problem into sub-problems of reasonable size while maintaining incremental solving. It takes as input a transaction database and a partition of the set of items, then it incrementally generates and sequentially solves a sequence of sub-problems while ensuring completeness. In this paper, we propose a new parallel SAT based framework for Closed Frequent Itemset Mining (in short **paraSatMiner**). Our proposed method is based on a divide-and-conquer paradigm. The main idea is to decompose the input transaction database, using item-based guiding paths, leading to sub-formulas that can be solved independently in parallel. Our paraSatMiner approach is carefully implemented on a multicore architecture, while maintaining workload balanced between the different processor units. Extensive experimental comparative evaluation on several datasets achieves significant performance improvements w.r.t. the two most recent and effective CP approaches [22, 30] even on the sequential case, while reducing the performance gap against LCM, one of the state-of-the-art specialized algorithms. We also show additional performance gains up to 8 cores. Contrary to CP mining systems based on the design of specific global constraints, our approach does not integrate any technique or property from specialized algorithms.

2 Technical Background and Preliminary Definitions

2.1 Propositional Logic and SAT Problem

Let \mathcal{L} be a propositional language defined from a finite set \mathcal{PS} of propositional symbols (p, q, r , etc.). The set of formulas is defined inductively from \mathcal{PS} , the constant \perp denoting false, the constant \top denoting true, and using the classical logical connectives $\neg, \wedge, \vee, \rightarrow$ and the equivalence connective \leftrightarrow . For every two propositional formulas Φ and Ψ from \mathcal{L} , the connective \leftrightarrow is defined by $\Phi \leftrightarrow \Psi \equiv (\Phi \rightarrow \Psi) \wedge (\Psi \rightarrow \Phi)$. Moreover, we refer to the set of symbols from \mathcal{PS} that occur in the formula Φ as $\mathcal{S}(\Phi)$.

A *boolean interpretation* \mathcal{B} of a formula Φ is a truth assignment of \mathcal{PS} , that is, a total function from $\mathcal{S}(\Phi)$ to $\{0, 1\}$ (0 corresponds to false and 1 corresponds to true). For every two formulas Φ and Ψ from \mathcal{L} , we have the following equivalences: $\mathcal{B}(\perp) = 0$, $\mathcal{B}(\top) = 1$, $\mathcal{B}(\neg\Phi) = 1 - \mathcal{B}(\Phi)$, $\mathcal{B}(\Phi \wedge \Psi) = \min(\mathcal{B}(\Phi), \mathcal{B}(\Psi))$, $\mathcal{B}(\Phi \vee \Psi) = \max(\mathcal{B}(\Phi), \mathcal{B}(\Psi))$, and $\mathcal{B}(\Phi \rightarrow \Psi) = \max(1 - \mathcal{B}(\Phi), \mathcal{B}(\Psi))$.

A *model* of a formula Φ is a boolean interpretation \mathcal{B} that satisfies Φ , i.e., $\mathcal{B}(\Phi) = 1$. A formula Φ is *satisfiable* if there exists a model of Φ . We denote by $\mathcal{M}(\Phi)$ the set of all models of Φ . Let $X \subseteq \mathcal{S}(\Phi)$ and $\mathcal{B} \in \mathcal{M}(\Phi)$, we define $\mathcal{B}^X = \{x \in X \mid \mathcal{B}(x) = 1\}$ and $\mathcal{M}^X(\Phi) = \{\mathcal{B}^X \mid \mathcal{B} \in \mathcal{M}(\Phi)\}$.

As usual, every finite set of formulas is considered as the conjunctive formula whose conjuncts are the elements of the set. A formula in *conjunctive normal form* (CNF) is a (finite) conjunction of clauses. A *clause* is a (finite) disjunction of literals. A *literal* is either a propositional variable x of \mathcal{PS} or its negation $\neg x$. Let us also mention that any propositional formula can be translated to a CNF formula equivalent w.r.t. satisfiability, using linear Tseitin's encoding [32]. The propositional satisfiability problem, called **SAT**, is the decision problem of determining the satisfiability of a CNF formula.

2.2 Parallel SAT Solving

Parallel SAT solving has received a lot of attention in the last years. This comes from several factors like the wide availability of cheap multicore platforms combined with the relative performance stall of sequential SAT solvers.

Many parallel SAT solvers have been previously proposed. Most of them are based on the divide-and-conquer principle. These solvers either divide the search space using for example guiding-paths [31, 37] or the formula itself using decomposition techniques. The main issue behind these approaches rises in (i) getting the workload balanced between the different processing units, and (ii) selecting the most relevant guiding paths.

Portfolio-based parallel SAT solving has been recently introduced [14]. It avoids the previous problem by letting several DPLL engines compete and cooperate to be the first to solve a given instance. Each solver works on the original formula, and search spaces are not split or decomposed anymore. To be efficient, the portfolio has to use diversified search engines. This maximizes the chance of having one of them solving the problem. However, when clause sharing is added, diversification has to be restricted in order to maximize the impact of a foreign clause whose relevance is more important in a similar or related search effort. A challenging question is to maintain a good and relevant distance between the parts of the search space explored by the different search efforts which is equivalent to finding of a better diversification and intensification tradeoff.

These two parallel solving paradigms are complementary and admit their own strengths and weaknesses. As our goal is to partition the transactions database in order to generate several formulas of reasonable size, the divide-and-conquer based paradigm is clearly the most convenient for our purpose.

2.3 Itemset Mining Problem Based on Boolean Satisfiability

Let Ω denote a universe of items (or symbols), called alphabet. A *transaction database* is a finite set of n data records, called *transactions*, denoted by $\mathcal{D} = \{T_1, T_2, \dots, T_n\}$. Each transaction T_i ($1 \leq i \leq n$) is defined as a couple (tid_i, \mathcal{I}_i) , where tid_i is the *transaction identifier* and $\mathcal{I}_i \subseteq \Omega$ an itemset, i.e., unordered collection of items from Ω . We use 2^Ω to denote the set of all possible itemsets over Ω . Let $a \in \Omega$, we define $\mathcal{D} \downarrow_a = \{T \mid T = (tid, \mathcal{I}) \in \mathcal{D}, a \in \mathcal{I}\}$ the set of transactions containing a . We also define $\mathcal{D} \uparrow_a = \{(tid, \mathcal{I} \setminus \{a\}) \mid (tid, \mathcal{I}) \in \mathcal{D}\}$.

We associate to each $a \in \Omega$ a propositional variable denoted l_a . We note \mathcal{P} such set of variables encoding the items in Ω and representing the candidate pattern. We also associate with each transaction T_i ($1 \leq i \leq n$) a propositional variable q_i . We also note \mathcal{Q} the set of variables associated to transactions. These variables will be used to express the following notions as propositional formulas.

Definition 1 (Cover). Let $\mathcal{D} = \{T_1, T_2, \dots, T_n\}$ be a transaction database. The cover of an itemset $\mathcal{I} \subseteq \Omega$ in \mathcal{D} , denoted $\mathcal{C}(\mathcal{I}, \mathcal{D})$, corresponds to the following set of transaction identifiers: $\mathcal{C}(\mathcal{I}, \mathcal{D}) = \{tid \mid (tid, \mathcal{J}) \in \mathcal{D} \text{ and } \mathcal{I} \subseteq \mathcal{J}\}$.

For instance, if we consider the transaction database of Table 1, we have $\mathcal{C}(\{b\}, \mathcal{D}) = \{1, 2, 3, 4\}$ while $\mathcal{C}(\{a, b\}, \mathcal{D}) = \{1, 2, 3\}$.

Table 1. Transaction database

TID	Transactions
T_1	a b c d
T_2	a b c e
T_3	a b e
T_4	b d
T_5	d f g
T_6	f g h
T_7	h

Cover constraint: The following constraint allows us to capture all the transactions where the candidate itemset does not appear:

$$\Phi_{\mathcal{D}}^{cov} = \bigwedge_{i=1}^n (\neg q_i \leftrightarrow \bigvee_{a \in \Omega \setminus \mathcal{I}_i} l_a) \tag{1}$$

This constraint means that q_i is true if and only if the candidate itemset is in the transaction T_i .

Definition 2 (Support). Let $\mathcal{D} = \{T_1, T_2, \dots, T_n\}$ be a transaction database. The support of an itemset $\mathcal{I} \subseteq \Omega$ in \mathcal{D} , denoted by $\mathcal{S}(\mathcal{I}, \mathcal{D})$, is defined as follows: $\mathcal{S}(\mathcal{I}, \mathcal{D}) = |\mathcal{C}(\mathcal{I}, \mathcal{D})|$.

From the transaction database of Table 1, we have $\mathcal{S}(\{a, b\}, \mathcal{D}) = 3$.

Definition 3 (Frequent Itemset Mining). *Given a transaction database \mathcal{D} and θ an explicit frequency user-specified support threshold. The problem of frequent itemset mining, denoted $FIM(\mathcal{D}, \theta)$, can be defined as follows:*

$$FIM(\mathcal{D}, \theta) = \{I \subseteq \Omega \mid \mathcal{S}(I, \mathcal{D}) \geq \theta\}$$

As an example, consider again the transaction database depicted by Table 1, we have $FIM(\mathcal{D}, 2) = \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{g\}, \{h\}, \{a, b\}, \{a, c\}, \{a, e\}, \{b, c\}, \{b, e\}, \{b, d\}, \{f, g\}, \{a, b, c\}, \{a, b, e\}\}$.

Let us note that the frequency is a monotonic property w.r.t. set inclusion, meaning that if an itemset is not frequent, none of its supersets are frequent. Similarly, if an itemset is frequent, all of its subsets are frequent.

Frequency constraint: To express that the candidate pattern occurs at least θ times, we use the following 0/1 linear inequality:

$$\Phi_{\mathcal{D}, \theta}^{freq} = \sum_{i=1}^n q_i \geq \theta \tag{2}$$

Then, the SAT-based encoding of $FIM(\mathcal{D}, \theta)$ is expressed as: $\Phi_{\mathcal{D}, \theta}^{fim} = \Phi_{\mathcal{D}}^{cov} \wedge \Phi_{\mathcal{D}, \theta}^{freq}$. Obviously, the monotonic property is implicitly satisfied by the frequency constraint. Notice also that the constraint (2) corresponds to the well known boolean cardinality constraint, subject of several efficient CNF encoding that maintain generalized arc consistency via unit propagation [2, 17, 34].

Now, in order to reduce the size of the huge number of extracted itemsets, condensed representations have been introduced, by exploiting the structure of the itemsets data. We define below *closed frequent itemsets* as one of these condensed representations.

Definition 4. (Closed Itemsets): *Let \mathcal{D} be a transaction database. We say that the itemset \mathcal{I} is closed if and only if for all $\mathcal{J} \supset \mathcal{I}$, $\mathcal{S}(\mathcal{I}, \mathcal{D}) > \mathcal{S}(\mathcal{J}, \mathcal{D})$.*

Closeness constraint: The constraint allowing to force the candidate itemset to be closed can be expressed by the following propositional formula:

$$\Phi_{\mathcal{D}}^{clos} = \bigwedge_{a \in \Omega} ((\bigvee_{(tid_i, \mathcal{I}_i) \in \mathcal{D}, a \notin \mathcal{I}_i} q_i) \vee l_a) \tag{3}$$

That is, this formula means that if we have $\mathcal{S}(\mathcal{I}, \mathcal{D}) = \mathcal{S}(\mathcal{I} \cup \{a\}, \mathcal{D})$, then $a \in \mathcal{I}$ holds. This condition is necessary and sufficient to force the candidate itemset to be closed. Let us note that an expression of the form $a \in \mathcal{I}_i$ corresponds to a constant, i.e., $a \in \mathcal{I}_i$ corresponds to \top if the item a is in \mathcal{I}_i , to \perp otherwise.

The closed frequent itemset mining task **CFIM** can then be encoded as $\Phi_{\mathcal{D}, \theta}^{cfim} = \Phi_{\mathcal{D}}^{clos} \wedge \Phi_{\mathcal{D}, \theta}^{fim}$, i.e., the conjunction of the formulas (1), (2) and (3). In the sequel, for clarity reasons and when there is no ambiguity, we simply note $\Phi_{\mathcal{D}, \theta}^{cfim}$ as $\Phi_{\mathcal{D}}$.

Let us consider again the transaction database \mathcal{D} of Table 1. Assume that the minimum support threshold $\theta = 2$. Then, the closed frequent itemsets are: $\{b\}$, $\{h\}$, $\{a, b\}$, $\{b, d\}$, $\{f, g\}$, $\{a, b, c\}$ and $\{a, b, e\}$.

3 Partition-Based Parallel SAT Approach for CFIM

Through an illustrative example, we start by highlighting some weaknesses of the SAT-based encoding of CFIM task described in Sect. 2.3.

Example 1. Let \mathcal{D} be a transaction database made of two sub-bases \mathcal{D}_1 and \mathcal{D}_2 built over two disjoint sets of items Ω_1 and Ω_2 , respectively. Formally, let $\mathcal{D}_1 = \{T_1, T_2, \dots, T_j\}$ and $\mathcal{D}_2 = \{T_{j+1}, T_{j+2}, \dots, T_n\}$, such that for each $T_i = (tid_i, \mathcal{I}_i)$, we have $\mathcal{I}_i \subseteq \Omega_1$ (resp. $\mathcal{I}_i \subseteq \Omega_2$) for $1 \leq i \leq j$ (resp. $j + 1 \leq i \leq n$). In the SAT encoding of $CFIM(\mathcal{D}, \theta)$, the propositional variables associated to the items in Ω_1 are used in the encoding of the sub-base \mathcal{D}_2 and vis-versa. Indeed, the cover constraint $\Phi_{\mathcal{D}}^{cov}$ (resp. closeness constraint $\Phi_{\mathcal{D}}^{clos}$) involves for each transaction $T_i = (tid_i, \mathcal{I}_i) \in \mathcal{D}$ the propositional variables associated to the items $a \in \Omega \setminus \mathcal{I}_i$ (resp. $a \in \Omega$), whereas \mathcal{D} is made of two independent transaction databases \mathcal{D}_1 and \mathcal{D}_2 . In this worst case illustrative example, the encoding leads to a formula with a high number of large clauses. Indeed, the weakness of SAT-based approaches is the size of the encoding, which for large formulas can outgrow available memory or can make SAT solving otherwise inefficient.

To overcome this drawback, a possible encoding is to express the problem as two independent propositional formulas $\Phi_{\mathcal{D}_1}$ and $\Phi_{\mathcal{D}_2}$. An alternative compact encoding of the itemset mining task on the whole database \mathcal{D} can be expressed as a single propositional formula:

$$\Phi_{\mathcal{D}} = [y \rightarrow (\Phi_{\mathcal{D}_1} \wedge \bigwedge_{l_a \in \Omega_2} \neg l_a)] \wedge [\neg y \rightarrow (\Phi_{\mathcal{D}_2} \wedge \bigwedge_{l_a \in \Omega_1} \neg l_a)] \quad (4)$$

The formula (4) allows us to split in an efficient way the patterns of \mathcal{D}_1 and \mathcal{D}_2 . Assigning y to *true* (resp. *false*) leads to the enumeration of the models of $\Phi_{\mathcal{D}_1}$ (resp. $\Phi_{\mathcal{D}_2}$), where the propositional variables associated to the items from Ω_2 (resp. Ω_1) are assigned by unit propagation to *false*. Interestingly, the formula (4) can be easily translated into clausal form.

Clearly, the previous example illustrates the motivation behind partitioning the transaction database for an efficient SAT based itemsets enumeration tasks. The benefits are twofold: the size of the encoding can be reduced significantly while improving solving efficiency. Unfortunately, disjoint sub-bases are not very common in real datasets and their recognition is not an easy task. Nevertheless, partitioning can be performed differently. In fact, for a transaction database \mathcal{D} and an item a , the set of frequent closed itemsets can be partitioned into those containing a (models of $\Phi_{\mathcal{D}} \wedge l_a$) and those without a (models of $\Phi_{\mathcal{D}} \wedge \neg l_a$). Interestingly, the models of $\Phi_{\mathcal{D}} \wedge l_a$ are those of $\Phi_{\mathcal{D}_{\downarrow a}}$. While the models of

$\Phi_{\mathcal{D}} \wedge \neg l_a$ correspond to those of $\Phi_{\mathcal{D}\uparrow_a} \wedge \Psi_{\mathcal{D},a}$, where $\Psi_{\mathcal{D},a}$ is defined as:

$$\Psi_{\mathcal{D},a} = \bigvee_{(tid_i, \mathcal{I}_i) \in \mathcal{D}, a \notin \mathcal{I}_i} q_i$$

Adding $\Psi_{\mathcal{D},a}$ allows to avoid such redundancies. In this way, any model of $\Phi_{\mathcal{D}\uparrow_a} \wedge \Psi_{\mathcal{D},a}$ must correspond to a pattern that covers at least one transaction not containing a . Let us note that the constraint $\Psi_{\mathcal{D},a}$ can be derived from the closure constraint (3) by assigning l_a to *false*.

As a summary, the assignment of the variable l_a to *true* allows to restrict the mining process to the transactions containing the item a , while setting l_a to *false* allows to remove the item a from the database \mathcal{D} with the addition of $\Psi_{\mathcal{D},a}$, so that to avoid redundancy.

Clearly, a less frequent item a_1 might lead to a CNF formula $\Phi_{\mathcal{D}} \wedge l_{a_1}$ of reasonable size. But, $\Phi_{\mathcal{D}} \wedge \neg l_{a_1}$ can remain very huge. Fortunately, the previous partitioning principle can be applied recursively on $\Phi_{\mathcal{D}} \wedge \neg l_{a_1}$ by choosing other items. For example, if we choose a second item a_2 , the formula $\Phi_{\mathcal{D}} \wedge \neg l_{a_1}$ can also be divided into $\Phi_{\mathcal{D}} \wedge \neg l_{a_1} \wedge l_{a_2}$ and $\Phi_{\mathcal{D}} \wedge \neg l_{a_1} \wedge \neg l_{a_2}$. Let us note that $\Phi_{\mathcal{D}} \wedge \neg l_{a_1} \wedge l_{a_2}$ is equivalent to $\Phi_{\mathcal{D}\uparrow_{a_1}\downarrow_{a_2}}$. Figure 1 shows this recursive partitioning process.

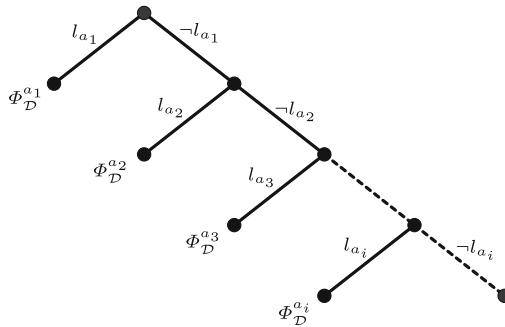


Fig. 1. Item-based guiding paths tree

Finally, if $\Omega = \{a_1, \dots, a_m\}$, then the models of $\Phi_{\mathcal{D}}$ can be partitioned into disjoint sets of models as stated in Proposition 1.

Proposition 1. *Let \mathcal{D} be a transaction database over Ω . Then,*

$$\mathcal{M}^{\mathcal{P}}(\Phi_{\mathcal{D}}) = \bigcup_{i=1}^m \mathcal{M}^{\mathcal{P}}(\Phi_{\mathcal{D}}^{a_i}), \text{ and } \mathcal{M}^{\mathcal{P}}(\Phi_{\mathcal{D}}^{a_i}) \cap \mathcal{M}^{\mathcal{P}}(\Phi_{\mathcal{D}}^{a_j}) = \emptyset \ (1 \leq i < j \leq m)$$

where $\Phi_{\mathcal{D}}^{a_i} = (\Phi_{\mathcal{D}\downarrow_{a_i}\uparrow_{a_1}\dots\uparrow_{a_{i-1}}} \wedge \bigwedge_{1 \leq j < i} (\Psi_{\mathcal{D}\downarrow_{a_i}, a_j}))$.

Proof (Sketch). Using the sequence of totally ordered set of items a_1, a_2, \dots, a_m , the formula $\Phi_{\mathcal{D}}$ can be decomposed into a sequence of formulas $\Phi_{\mathcal{D}}^{a_1}, \Phi_{\mathcal{D}}^{a_2}, \dots$, and $\Phi_{\mathcal{D}}^{a_m}$. This partition by the set of item-based guiding paths is complete as

depicted in Fig. 1. Indeed, starting with the item a_1 , the models of $\Phi_{\mathcal{D}}^{a_1}$ correspond to the patterns of \mathcal{D} restricted to transaction containing a_1 . The models of $\Phi_{\mathcal{D}}^{a_2}$ correspond to the patterns of \mathcal{D} restricted to transactions containing a_2 while removing the item a_1 . The constraint $\Psi_{\mathcal{D}\downarrow_{a_2}, a_1} = \bigvee_{(tid_i, \mathcal{I}_i) \in \mathcal{D}\downarrow_{a_2}, a_1 \notin \mathcal{I}_i} q_i$ allows us to generate patterns that cover at least one additional transaction containing a_2 and not a_1 , and so on.

Remark 1. As $\Phi_{\mathcal{D}}$ corresponds to the SAT-based encoding of $CFIM(\mathcal{D}, \theta)$, the constraint $\Psi_{\mathcal{D}, a}$ avoiding patterns redundancy is derived from the closeness constraint (3) each time an item a is removed from \mathcal{D} (i.e., $\mathcal{D} \uparrow_a$) or equivalently l_a is assigned to *false* (i.e., $\Phi_{\mathcal{D}} \wedge \neg l_a$). Obviously, the formulas $\Phi_{\mathcal{D}}^{a_i} \wedge l_{a_i}$ and $\Phi_{\mathcal{D}} \wedge l_{a_i} \wedge \neg l_{a_1} \wedge \dots \wedge \neg l_{a_{i-1}}$ admit the same models over \mathcal{P} . However, as our goal is to partition the encoding of \mathcal{D} into several independent formulas of reasonable size, it is better to consider $\Phi_{\mathcal{D}}^{a_i}$, where the encoding is done after restricting \mathcal{D} to transactions containing a_i , removing the items a_1, \dots, a_{i-1} and adding the constraint $\bigwedge_{1 \leq j < i} (\Psi_{\mathcal{D}\downarrow_{a_i}, a_j})$, than propagating the literals $l_{a_i} \wedge \neg l_{a_1} \wedge \dots \wedge \neg l_{a_{i-1}}$ on the formula encoding the whole transaction database.

As explained in Sect. 2.2, divide-and-conquer is an usual way for parallelizing SAT solvers. This strategy uses the notion of guiding path to split the search space. Each derived sub-formula is solved using a sequential SAT solver running on a particular processor. In fact, partitioning the search space can be done statically or dynamically in order to avoid the idleness of threads. Additionally, workload balancing is another criteria allowing to distribute approximately equal amounts of work among processors over time. In this paper, we consider a static divide-and-conquer approach, i.e., the set of guiding paths are generated in a preprocessing step. Our partition-based approach is based on the set of formulas $\Phi_{\mathcal{D}}^{a_i}$ (see Proposition 1). We denote by g^{a_i} the i th guiding path defined as: $g^{a_i} = \neg l_{a_1} \wedge \dots \wedge \neg l_{a_{i-1}} \wedge l_{a_i}$. The guiding path g^{a_i} allows us to both restrict the search on a subset of transactions involving only a_i , with the items a_j ($1 \leq j < i$) removed. Each guiding path g^{a_i} leads to a formula $\Phi_{g^{a_i}} = \Phi_{\mathcal{D}} \wedge g^{a_i}$ associated to the i th branch of the item-based guiding paths tree depicted in Fig. 1. As $\Phi_{\mathcal{D}}$ is the propositional formula encoding $CFIM(\mathcal{D}, \theta)$, the two formulas $\Phi_{g^{a_i}}$ and $\Phi_{\mathcal{D}}^{a_i}$ are equivalent. Naturally, each guiding path g^{a_i} can be extended with additional items to further reduce the size of the associated formula.

Notice that the total ordering over Ω used to generate the guiding paths greatly impacts the size of the associated sequence of formulas and therefore it can significantly affect the performance. Finding the best ordering for an efficient parallelization is clearly a challenging issue. Our goal is then to decompose the encoding into smaller propositional formulas in order to efficiently balance the load between the different cores. Definitions 5 and 6 formalize this issue.

Definition 5. *Given a transaction database \mathcal{D} on $\Omega = \{a_1, \dots, a_m\}$. Let σ be a permutation over Ω . We define $G^{\Omega, \sigma} = \{g^{\sigma(a_1)}, \dots, g^{\sigma(a_m)}\}$ as the set of guiding paths over Ω w.r.t. σ . We also define $\Phi(\mathcal{D}, \sigma)$ as the set of propositional formulas encoding \mathcal{D} w.r.t. σ , i.e., $\Phi(\mathcal{D}, \sigma) = \{\Phi_{g^{\sigma(a_1)}}, \dots, \Phi_{g^{\sigma(a_m)}}\}$.*

Finding an appropriate permutation is a hard task. In fact, the relevance of such permutation depends on the overall complexity of the model enumeration process on the associated formulas. Considering the size of the formula as a complexity measure, the issue can be formulated as an optimization problem:

Definition 6. Let \mathcal{D} be a transaction database on Ω . We define the *Best Items Decomposition Ordering Problem* as the problem of finding the best permutation σ that leads to a set of formulas encoding \mathcal{D} while minimizing the size of the largest formula in $\Phi(\mathcal{D}, \sigma)$: $\sigma^* = \arg \min_{\sigma \in \text{Perm}(\Omega)} \max\{|\phi|, \phi \in \Phi(\mathcal{D}, \sigma)\}$, where $\text{Perm}(\Omega)$ is the set of all permutations over Ω .

The main idea is to reduce the overall computational complexity of the model enumeration task among all the generated formulas under the set of guiding paths. Note that the size of each formula $\Phi_{g_{\sigma(a_i)}}$ depends on the number of transactions involving $\sigma(a_i)$ and on the the frequency of $\sigma(a_j)$ for $(j < i)$ in such transactions. From this observation, as a decomposition ordering we consider items from the less frequent to the most frequent one. This static ordering, easy to compute, proved to be a good compromise as shown in our experiments.

Algorithm 1, called paraSatMiner, summarizes the main components of our partition-based parallel SAT approach for CFIM. It takes as input a transaction database \mathcal{D} over Ω , a permutation σ on Ω , a minimum support threshold θ and a fixed number of threads (cores), and returns the set of closed frequent itemsets as output. The number of threads is less or equal to the number of items ($n \leq m$). The set of items are sorted in ascending order according to their frequency, i.e., the permutation σ over Ω satisfies the following condition: $\forall 1 \leq i < j \leq m, \mathcal{S}(\mathcal{D}, \{\sigma(a_i)\}) \leq \mathcal{S}(\mathcal{D}, \{\sigma(a_j)\})$. First, the SAT-based model enumeration solvers are initialized. Each one is associated to a given thread or core i , while initializing the set of models \mathcal{M}_i returned by each thread i to an empty-set (lines 1–4). Now, the n model enumeration solvers are launched in parallel (lines 7–11). For example, the solver i is run successively on the formulas $\Phi_{\mathcal{D}}^{\sigma(a_{i+k \times n})}$ corresponding to the guiding paths (or branches) number $(i + k \times n) \leq m$. The set of models computed by the solver i are collected in the set variable \mathcal{M}_i . In this way, the solver i is run on the formulas $\Phi_{\mathcal{D}}^{\sigma(a_i)}, \Phi_{\mathcal{D}}^{\sigma(a_{i+n})}, \Phi_{\mathcal{D}}^{\sigma(a_{i+2 \times n})}, \dots, \Phi_{\mathcal{D}}^{\sigma(a_{i+k \times n})}$, with $i + k \times n \leq m$. As the items are ordered according to their frequency, the sequences of formulas associated to the n model enumeration solvers are approximately of closer sizes. This allow us to ensure load balancing between the different solvers. Finally, in lines 12–14, the set of closed frequent itemsets are collected by merging the model enumerated by each thread, projected on the variables encoding the itemsets \mathcal{P} .

4 Experimental Results

In this section, we evaluate the performance of paraSatMiner. The proposed parallel SAT mining solver is implemented using a model enumeration solver based on MiniSAT [9]. As our approach is based on the enumeration of all models of a

Algorithm 1. Parallel SAT for Closed Frequent Itemset Mining (paraSat-Miner)

Input: \mathcal{D} : a transaction database, $\Omega = \{a_1, \dots, a_m\}$: a set of items, σ : a permutation over Ω (ordering), θ : a minimum support threshold, n : number of Threads

Output: The set of Closed Frequent Itemsets $CFIM(\mathcal{D}, \theta)$

```

1 foreach  $i \in \{1, \dots, n\}$  do
2   |  $initEnumSatSolver(i)$ ;
3   |  $\mathcal{M}_i = \emptyset$ ;
4 end
5  $S = \emptyset$ ;
6  $k = 0$ ;
7 # in parallel;
8 if  $(i + k \times n \leq |\Omega|)$  then
9   |  $\mathcal{M}_i \leftarrow \mathcal{M}_i \cup enumModels(enumSatSolver_i, \Phi_D^{\sigma(a_i+k \times n)})$ ;
10  |  $k++$ ;
11 end
12 foreach  $i \in \{1, \dots, n\}$  do
13  |  $S = S \cup \mathcal{M}_i^P$ ;
14 end
15 Return  $S$ 

```

propositional formula, we propose an extension of MiniSAT solver based on the DPLL (Davis-Putnam-Logemann-Loveland) procedure [8] to deal with this fundamental problem, marginally explored in the SAT community. More precisely, each time a model is found, a chronological backtracking is performed while inhibiting the restart component to ensure completeness. We also use `OpenMP` as an API that supports multi-platform shared memory multiprocessing programming in C and C++. Let us note that the cardinality constraints representing the support constraint is managed on the fly inside the solver. We also use the well-known MOMS variable ordering heuristic [21].

All the experiments were done on Intel Xeon quad-core machines with 32GB of RAM running at 2.66 Ghz. To evaluate the practical performance of our approach, we run the experiments on different datasets, taken from FIMI¹ and CP4IM² repositories. For each instance, we fix a timeout of 1000s of CPU time. Table 2 presents the characteristics of the evaluated datasets. For each instance, we report the number of transactions (#Transactions), the number of items (#Items), the density, and the size of the dataset in bytes.

For our experiments, we denote by `paraSatMiner-*c3` our parallel SAT solver based itemset enumeration where (*) indicates the number of threads. We compare the performance of our approach with the following most prominent

¹ <http://fimi.ua.ac.be/data/>.

² <http://dtai.cs.kuleuven.be/CP4IM/datasets/>.

³ <http://www.cril.univ-artois.fr/decMining/>.

state-of-the-art CP-based CFIM systems: **ClosedPattern**⁴⁵ and **CoverSize**⁶. As mentioned previously, **ClosedPattern** and **CoverSize** encode the closeness constraint as global constraints. These CP mining systems are implemented in C++ and Java using the or-tools solver [22] (or **choco** for the recent version) and the Oscar solver [30], respectively. In addition, we consider **LCM**⁷ known as the best specialized solver for frequent (closed) itemset mining. We perform two kinds of experiments. In the first evaluation, we carry out a comparison with all the considered itemset mining systems. In this case, **paraSatMiner** is run using 1 core as a sequential solver. In the second, we show that the performance of our **paraSatMiner** can be improved by increasing the number of threads. In this case, we vary the number of cores from 1 to 8.

Table 2. Data characteristics

Instance	#Transactions	#Items	Density	Size
chess	3196	75	49.0%	340K
mushroom	8124	119	19.0%	516K
BMS-WebView-1	59601	497	0.5%	940K
T10I4D100K	100000	870	1.0%	3.9M
retail	88162	16470	0.06%	4,2M
connect	67558	129	35.62%	8.9M
T40I10D100K	100000	942	4.31%	15M
pumsb	49046	2113	3.0%	16,7M
kosarak	990002	41267	0.01%	32M
accidents	340183	468	7.0%	34M

4.1 Sequential Evaluation

Table 3 reports the comparative results of the considered systems using different minimum support threshold values. For each dataset, the number of models (patterns) and the total CPU time (in seconds) are given. For **CoverSize**, we also mention in parenthesis the solving CPU time without including the time spent for loading and encoding the instance. (–) means that the solver is not able to finish the enumeration under the fixed time out. According to the results, our **ParaSatMiner-1c** (with 1 core) solver outperforms considerably the CP approaches on almost all the datasets with several factors of magnitudes of gain. For instance, our approach allows to enumerate all the models of **connect** data in 180s for $\theta = 5000$ when both **ClosedPattern** and **CoverSize** timed out. Except for **T10I4D100K** and $\theta = 50$, where **CoverSize** is the best. For all the

⁴ <http://www.lirmm.fr/~lazaar/cpminer.html>.

⁵ <https://lemierev.users.greyc.fr/closedpattern/>.

⁶ <https://www.info.ucl.ac.be/~pschaus>.

⁷ <http://research.nii.ac.jp/~uno/code/lcm.html>.

Table 3. paraSatMiner vs (ClosedPattern, CoverSize, LCM)

Instance	θ	Closed pattern	Cover size	paraSat Miner-1c	LCM	#Models
retail	80	–	265.10 (42.84)	14.06	0.21	$> 8.10^3$
	60	–	295.47 (72.04)	18.07	0.24	$> 1.10^4$
	40	–	334.23 (113.14)	25.33	0.28	$> 2.10^4$
	20	–	439.94 (216.68)	41.93	0.35	$> 5.10^4$
	10	–	586.16 (361.71)	76.49	0.56	$> 1.10^5$
connect	40000	7.54	14.95 (9.67)	7.25	0.17	$> 7.10^4$
	20000	50.22	75.48 (70.40)	22.73	0.55	$> 5.10^5$
	10000	526.43	431.19 (426.34)	68.88	2.68	$> 3.10^6$
	5000	–	–	179.17	9.82	$> 1.10^7$
chess	2000	1.51	1.22 (0.51)	0.25	0.04	$\approx 7.10^4$
	1500	6.30	4.09 (3.38)	0.8	0.20	$> 5.10^5$
	1000	51.35	28.62 (27.93)	5.52	1.75	$> 4.10^6$
	500	577.29	311.47 (310.74)	49.50	18.25	$> 45.10^6$
	250	–	–	186.11	72.96	$\approx 2.10^8$
	100	–	–	484.41	215.30	$> 5.10^8$
accidents	100000	101.68	145.96 (81.84)	74.14	1.72	$\approx 1.10^5$
	80000	319.25	283.98 (220.86)	141.29	3.53	$\approx 4.10^5$
	60000	–	866.21 (804.33)	299.94	5.66	$> 1.10^6$
	40000	–	–	735.39	10.59	$\approx 6.10^6$
pumbs	40000	20.99	389.43 (1.83)	4.78	0.13	$> 2.10^4$
	35000	103.20	325.42 (19.54)	10.33	0.25	$\approx 2.10^5$
	30000	434.25	404.26 (97.85)	27.72	0.59	$\approx 9.10^5$
	25000	–	994.35 (690.09)	147.64	3.18	$\approx 6.10^6$
	20000	–	–	669.93	17.69	$\approx 3.10^7$
T40I10D100K	10000	4.16	51.43 (0.145)	3.15	0.34	$\approx 1.10^2$
	8000	5.08	51.13 (0.352)	4.07	0.54	$> 1.10^2$
	6000	10.38	52.39 (0.912)	6.43	0.79	$> 2.10^2$
	4000	30.51	53.26 (2.221)	9.59	1.00	$> 4.10^2$
	2000	144.89	58.22 (7.378)	22.74	1.64	$> 1.10^3$
T10I4D100K	500	106.87	24.04 (6.91)	2.90	0.35	$> 1.10^3$
	400	147.14	25.56 (8.32)	3.32	0.37	$> 1.10^3$
	300	217.40	27.73 (10.69)	4.31	0.40	$> 4.10^3$
	200	314.17	29.12 (11.98)	6.16	0.44	$> 1.10^4$
	100	497.10	32.40 (15.14)	14.66	0.48	$> 2.10^4$
	50	–	45.05(27.80)	76.37	0.58	$> 4.10^4$
Online-Retail	70	–	211.71 (97.76)	31.66	0.82	$\approx 6.10^3$
	40	–	233.34 (120.48)	34.64	0.85	$> 6.10^3$
	10	–	253.64 (141.61)	39.94	0.82	$\approx 8.10^3$
	5	–	267.96 (156.74)	41.79	0.90	$> 8.10^3$
Kosarak	4000	–	–	29.37	1.32	$> 2.10^3$
	3000	–	–	39.98	1.61	$> 4.10^3$
	2000	–	–	65.12	2.16	$> 3.10^4$
	1000	–	–	145.65	3.43	$\approx 5.10^5$
mushroom	250	1.14	2.54 (0.50)	1.33	0.05	$> 1.10^4$
	100	1.64	1.91 (1.07)	1.94	0.06	$> 3.10^4$
	50	2.70	2.47 (1.63)	2.44	0.09	$> 5.10^4$
	25	2.82	3.16 (2.27)	2.90	0.11	$\approx 8.10^4$
	5	5.57	4.20 (3.33)	3.90	0.17	$> 1.10^5$
BMS-WebView-1	48	203.51	220.67 (1.43)	8.69	0.06	$> 9.10^3$
	36	833.76	220.20 (3.32)	10.08	0.16	$> 6.10^4$
	34	–	219.19 (4.79)	10.50	0.21	$> 8.10^4$
	32	–	227.80 (8.19)	10.89	0.32	$> 1.10^5$
	30	–	231.89 (14.88)	11.37	0.53	$> 1.10^5$

remaining tested values, **ParaSatMiner-1c** is clearly the best. Interestingly, on the **kosarak** data with 990002 transactions the two CP systems are not able to enumerate the whole set of models under the time limit for all θ values. For the comparison with specialized algorithms, even if our approach reduced the performance gap, the LCM algorithm remains the best system.

4.2 Parallel Evaluation

On the parallel side, we perform the same experiments by varying the number of cores from 1 to 8 and by considering several minimum support threshold values. Figure 2 shows the obtained results on a representative sample of dataset. A first remark is that the parallel approach allows us to reduce considerably the computation time. By increasing the number of cores, the time always decreases. The impressive gain is obtained from 1 to 2 cores as shown by Fig. 2. For the

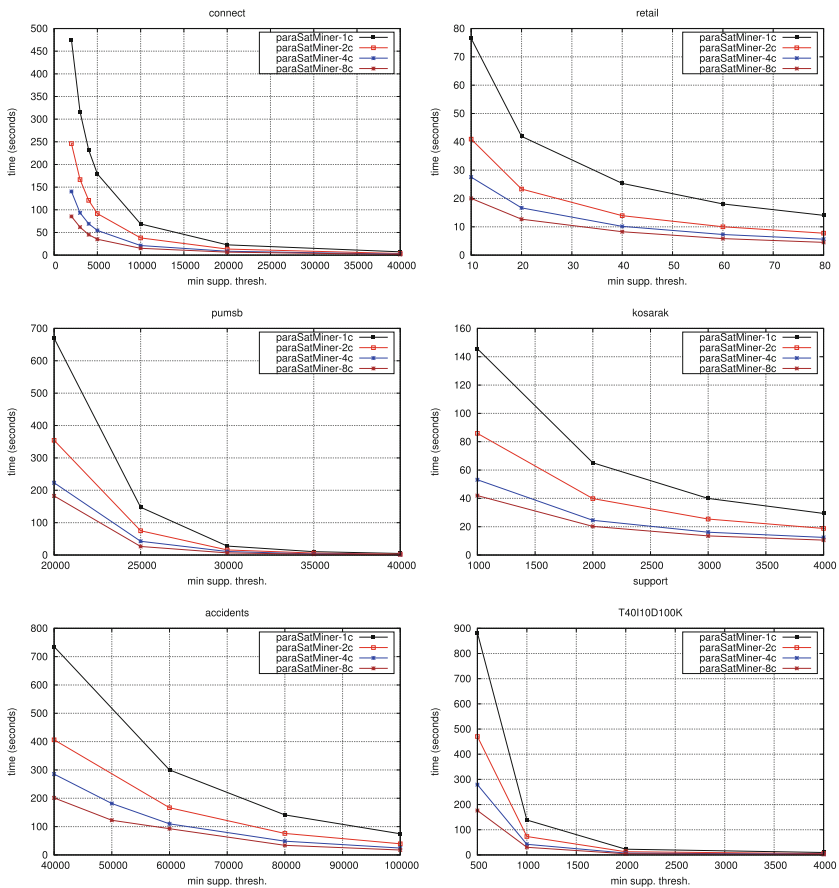


Fig. 2. Performance gain w.r.t. the number of cores

transition from 4 to 8 cores, the gain is not very impressive. Overall, the strategy used to generate guiding paths by considering the less frequent items first is successful. For instance for `pumsb` data and $\theta = 2000$, the time is reduced from 700s with 1 core to less than 200s with 8 cores.

4.3 Load Balancing

Finally, to assess the suitability of our load balancing strategy, we consider the datasets `pumsb` and `T10I4D100K` and we provide an empirical analysis of the number of models discovered by the different cores. Using the 4 (resp. 8) cores configuration, for each value of θ we report the average number of models among the 4 (resp. 8) cores (the curve), while mentioning the minimum and the maximum number of models (the interval) (see Fig. 3). We note that the load balancing strategy is better when the minimum and maximum number of models among those returned by the different cores are close to the average number of models. As we can observe, for `T10I4D100K`, our strategy is clearly interesting. Indeed, the relative load unbalancing is very limited. All the cores find approximately the same number of models. However, the load unbalancing is relatively high for `pumsb`, mainly for the transition between 4 to 8 cores.

Overall, our guiding paths generation strategy allows to balance the number of models between the different threads. These results indicate that our method is both efficient and scalable.

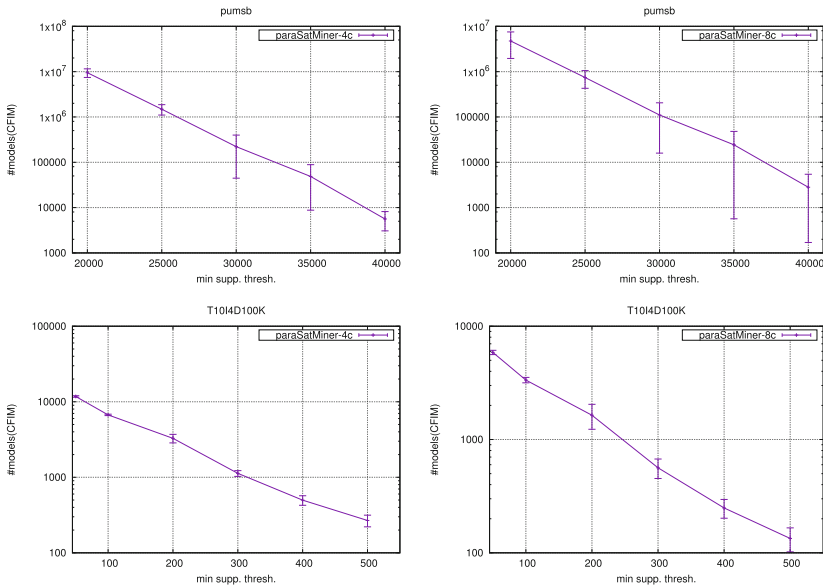


Fig. 3. Load unbalancing between cores

5 Conclusion

In this paper, we proposed a parallel SAT-based framework for CFIM. We show that partitioning the formula using predefined guiding paths allows to push forward the performance of SAT-based itemset mining frameworks. Even in the sequential configuration, our approach outperforms the existing CP-based approaches while reducing considerably the time needed to compute all the itemsets. We have shown that such approach scales well and presents good load balancing among the different cores.

This work can be extended in different ways. First, we would like to consider a dynamic partitioning strategy. We also plan to implement a distributed version to handle very large datasets. Last, we plan to tackle other DM problems like discriminative itemsets, association rules mining, and skypatterns tasks.

References

1. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. In: ACM SIGMOD International Conference on Management of Data, pp. 207–216 (1993)
2. Bailleux, O., Bouffkhad, Y.: Efficient CNF encoding of boolean cardinality constraints. In: International Conference on Principles and Practice of Constraint Programming CP, pp. 108–122 (2003)
3. Bastide, Y., Pasquier, N., Taouil, R., Stumme, G., Lakhal, L.: Mining minimal non-redundant association rules using frequent closed itemsets. In: Lloyd, J., et al. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 972–986. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44957-4_65
4. Borgelt, C.: Frequent item set mining. Wiley Int. Rev.: Data Min. Knowl. Disc. **2**(6), 437–456 (2012)
5. Boudane, A., Jabbour, S., Sais, L., Salhi, Y.: A sat-based approach for mining association rules. In: IJCAI, pp. 2472–2478 (2016)
6. Boudane, A., Jabbour, S., Sais, L., Salhi, Y.: Clustering complex data represented as propositional formulas. In: Kim, J., Shim, K., Cao, L., Lee, J.-G., Lin, X., Moon, Y.-S. (eds.) PAKDD 2017. LNCS (LNAI), vol. 10235, pp. 441–452. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57529-2_35
7. Dao, T., Duong, K., Vrain, C.: Constrained clustering by constraint programming. *Artif. Intell.* **244**, 70–94 (2017)
8. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. *Commun. ACM* **5**, 394–397 (1962)
9. En, N., Srensson, N.: An extensible sat-solver. In: Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), pp. 502–518 (2002)
10. Ganji, M., Bailey, J., Stuckey, P.J.: A declarative approach to constrained community detection. In: International Conference on Principles and Practice of Constraint Programming, pp. 477–494 (2017)
11. Gebser, M., Guyet, T., Quiniou, R., Romero, J., Schaub, T.: Knowledge-based sequence mining with ASP. In: International Joint Conference on Artificial Intelligence, pp. 1497–1504 (2016)

12. Guns, T., Dries, A., Tack, G., Nijssen, S., Raedt, L.D.: Miningzinc: a modeling language for constraint-based mining. In: International Joint Conference on Artificial Intelligence, pp. 1365–1372 (2013)
13. Guns, T., Nijssen, S., Raedt, L.D.: Itemset mining: a constraint programming perspective. *Artif. Intell.* **175**(12–13), 1951–1983 (2011)
14. Hamadi, Y., Jabbour, S., Sais, L.: Manysat: a parallel SAT solver. *JSAT* **6**(4), 245–262 (2009)
15. Henriques, R., Lynce, I., Manquinho, V.M.: On when and how to use sat to mine frequent itemsets. *CoRR*, abs/1207.6253 (2012)
16. Jabbour, S., Mhadhbi, N., Raddaoui, B., Sais, L.: A sat-based framework for overlapping community detection in networks. In: Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, pp. 786–798 (2017)
17. Jabbour, S., Sais, L., Salhi, Y.: A pigeon-hole based encoding of cardinality constraints. *TPLP* **13**(4-5-Online-Supplement) (2013)
18. Jabbour, S., Sais, L., Salhi, Y.: The top-k frequent closed itemset mining using top-k SAT problem. In: *ECML/PKDD*, pp. 403–418 (2013)
19. Jabbour, S., Sais, L., Salhi, Y.: Decomposition based SAT encodings for itemset mining problems. In: Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, pp. 662–674 (2015)
20. Jabbour, S., Sais, L., Salhi, Y.: Mining top-k motifs with a SAT-based framework. *Artif. Intell.* **244**, 30–47 (2017)
21. Jeroslow, R.G., Wang, J.: Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.* **1**, 167–187 (1990)
22. Lazaar, N., Lebbah, Y., Loudni, S., Maamar, M., Lemièrre, V., Bessiere, C., Boizumault, P.: A global constraint for closed frequent pattern mining. In: International Conference on Principles and Practice of Constraint Programming, pp. 333–349 (2016)
23. Lin, Y.C., Wu, C., Tseng, V.S.: Mining high utility itemsets in big data. In: Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, pp. 649–661 (2015)
24. Liu, L., Li, E., Zhang, Y., Tang, Z.: Optimization of frequent itemset mining on multiple-core processor. In: International Conference on Very Large Data Bases (2007)
25. Moens, S., Aksehirli, E., Goethals, B.: Frequent itemset mining for big data. In: IEEE International Conference on Big Data, pp. 111–118 (2013)
26. Négrevèrgne, B., Guns, T.: Constraint-based sequence mining using constraint programming. In: International Conference on Integration of AI and OR Techniques in Constraint Programming, pp. 288–305 (2015)
27. Négrevèrgne, B., Termier, A., Méhaut, J., Uno, T.: Discovering closed frequent itemsets on multicore: parallelizing computations and optimizing memory accesses. In: International Conference on High Performance Computing & Simulation, pp. 521–528 (2010)
28. Raedt, L.D., Guns, T., Nijssen, S.: Constraint programming for itemset mining. In: *ACM SIGKDD*, pp. 204–212 (2008)
29. Savasere, A., Omiecinski, E., Navathe, S.B.: An efficient algorithm for mining association rules in large databases. In: International Conference on Very Large Data Bases, pp. 432–444 (1995)
30. Schaus, P., Aoga, J.O.R., Guns, T.: Coversize: a global constraint for frequency-based itemset mining. In: International Conference on Principles and Practice of Constraint Programming, pp. 529–546 (2017)

31. Schubert, T., Lewis, M.D.T., Becker, B.: Pamiraxt: parallel SAT solving with threads and message passing. *JSAT* **6**(4), 203–222 (2009)
32. Tseitin, G.: On the complexity of derivations in the propositional calculus. In: *Studies in Mathematics and Mathematical Logic*, pp. 115–125 (1968)
33. Wang, S., Yang, Y., Gao, Y., Chen, G., Zhang, Y.: Mapreduce-based closed frequent itemset mining with efficient redundancy filtering. In: *IEEE International Conference on Data Mining Workshops ICDM*, pp. 449–453 (2012)
34. Warners, J.P.: A linear-time transformation of linear inequalities into conjunctive normal form. *Inf Process Lett* **68**(2), 63–69 (1998)
35. Zaïane, O.R., El-Hajj, M., Lu, P.: Fast parallel association rule mining without candidacy generation. In: *IEEE International Conference on Data Mining*, pp. 665–668 (2001)
36. Zaki, M.J.: Mining non-redundant association rules. *Data Min. Knowl. Discov.* **9**(3), 223–248 (2004)
37. Zhang, H., Bonacina, M.P., Hsiang, J.: Psato: a distributed propositional prover and its application to quasigroup problems. *J. Symbolic Comput.* **21**(4), 543–560 (1996)
38. Zitouni, M., Akbarinia, R., Yahia, S.B., Masegla, F.: Massively distributed environments and closed itemset mining: the DCIM approach. In: *International Conference on Advanced Information Systems Engineering*, pp. 231–246 (2017)



Towards Effective Deep Learning for Constraint Satisfaction Problems

Hong Xu^(✉) , Sven Koenig, and T. K. Satish Kumar

University of Southern California, Los Angeles, CA 90089, USA
{hongx,skoenig}@usc.edu, tkskwork@gmail.com

Abstract. Many attempts have been made to apply machine learning techniques to constraint satisfaction problems (CSPs). However, none of them have made use of the recent advances in deep learning. In this paper, we apply deep learning to predict the satisfiabilities of CSPs. To the best of our knowledge, this is the first effective application of deep learning to CSPs that yields $>99.99\%$ prediction accuracy on random Boolean binary CSPs whose constraint tightnesses or constraint densities do not determine their satisfiabilities. We use a deep convolutional neural network on a matrix representation of CSPs. Since it is NP-hard to solve CSPs, labeled data required for training are in general costly to produce and are thus scarce. We address this issue using the asymptotic behavior of generalized Model A, a new random CSP generation model, along with domain adaptation and data augmentation techniques for CSPs. We demonstrate the effectiveness of our deep learning techniques using experiments on random Boolean binary CSPs. While these CSPs are known to be in P, we use them for a proof of concept.

1 Introduction

A lot of research has been dedicated to applying machine learning techniques to *constraint satisfaction problems* (CSPs), such as support vector machines [5], linear regression [27], decision tree learning [10, 12], clustering [15, 23], k -nearest neighbors [21], and so on [16]. However, there are a few drawbacks in these methods. First, they do not consistently produce extremely high ($>99\%$) prediction accuracies. Secondly, to the best of our knowledge, they critically rely on handcrafted features. For different distributions of CSPs coming from different application domains and for different tasks of interest, the optimal features need to be carefully selected by humans accordingly [16]. How to select good features thus requires dedicated research [3, 4].

Deep learning is a class of machine learning methods based on multi-layer (deep) *neural networks* (NNs). Thanks to the advent of “Big Data,” it has significantly advanced during the past decade and achieved great success in many

The research at the University of Southern California (USC) was supported by National Science Foundation (NSF) under grant numbers 1724392, 1409987, and 1319966.

areas, such as computer vision and natural language processing [11]. In these applications, it consistently produces extremely high prediction accuracies, and often approaches or even surpasses human-level performance in many human perception tasks, such as object recognition [8, 14] and speech recognition [6]. Furthermore, it does not rely on handcrafted features. One of the key reasons for the success of deep learning is the availability of huge amounts of training data. However, due to the NP-hardness of CSPs, it is costly to label CSPs with properties such as their satisfiabilities and the best algorithms to solve them. This has become a roadblock for effective deep learning for CSPs. Indeed, a recent study shows that, without a huge amount of labeled data, a *convolutional NN* (cNN) for algorithm selection is ineffective for CSPs [19].¹

In this work, we successfully apply deep learning to predict the satisfiabilities of random Boolean binary CSPs with high prediction accuracies (>99.99%). To the best of our knowledge, this is the first effective application of deep learning to random CSPs whose constraint tightnesses or constraint densities do not determine their satisfiabilities. Accurately predicting satisfiabilities might improve the dynamic variable ordering in a backtracking algorithm for CSPs to increase the likelihood of choosing a variable that results in a satisfiable subproblem so as to minimize backtracking. Further adapting our current method to qualitatively predict the number of solutions, e.g., “0,” “1,” and “ ≥ 1 ,” might further improve the dynamic variable ordering. In addition, *transfer learning* may be potentially used to enable effective deep learning for other tasks such as predicting the most efficient algorithm and its best parameter settings for a given CSP.

In this paper, first, we describe the architecture of our cNN. Then, we address the issue of the lack of labeled data using the asymptotic behavior of *generalized Model A*, a new random CSP generation model, along with domain adaptation and data augmentation techniques for CSPs. We demonstrate the effectiveness of our techniques using experiments on random Boolean binary CSPs. While these CSPs are known to be in P, we use them as a proof of concept.

Preliminaries. A CSP is formally defined as a tuple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X} = \{X_1, \dots, X_n\}$ is a set of variables, $\mathcal{D} = \{D_1, \dots, D_n\}$ is a set of domains corresponding to their respective variables, and $\mathcal{C} = \{C_1, \dots, C_m\}$ is a set of constraints. Each constraint $C_i \in \mathcal{C}$ is a pair $\langle S(C_i), R_i \rangle$, where $S(C_i)$ is a subset of \mathcal{X} and R_i is a $|S(C_i)|$ -ary relation that specifies incompatible and compatible assignments of values to variables in $S(C_i)$. In a *table constraint* C_i , R_i is a set of tuples, each of which indicates the compatibility of an assignment of values to variables in $S(C_i)$. A tuple is compatible if it specifies a compatible assignment of values to variables, and is otherwise incompatible. We focus on CSPs where all constraints are table constraints.

¹ Another related work [9] using a different approach was only publicly available after this paper was accepted, before which we had no access to it. Nevertheless, it only demonstrated low training and test accuracies in the experiments when the number of variables in a CSP is non-trivial (≥ 5) and we do not consider it effective (yet).

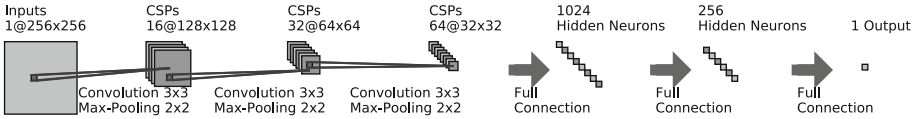


Fig. 1. The architecture of our CSP-cNN.

The concept of a cNN, a class of deep NN architectures, was initially proposed for an object recognition problem [18] and has recently achieved great success [8, 14]. It is a multi-layer feedforward NN that takes a multi-dimensional (usually 2-D or 3-D) matrix as input. While cNNs are mainly used for classification, they are also used for regression [24]. A cNN has three types of layers: *convolutional layers*, *pooling layers*, and *fully connected layers*. A convolutional layer performs a convolution operation. A pooling layer combines the outputs of several nodes in the previous layer into a single node in the current layer. A fully connected layer connects every node in the current layer to every node in the previous layer.

2 Enabling Deep Learning for CSPs

In the context of deep learning for CSPs, each *data point* is a CSP (instance). If a data point is *labeled*, its label is a property of this CSP, such as its satisfiability, its K -consistency, the best algorithm to solve it, or the amount of time required to solve it with a specific algorithm. Our cNN takes a data point (a CSP) as input and predicts its label. In order to enable a cNN to take a CSP as input, we represent a binary CSP using a matrix as follows. Each row/column represents a variable-value pair (X_i, x_i) . The element in (X_i, x_i) 's row and (X_j, x_j) 's column is zero if $\{X_i = x_i, X_j = x_j\}$ is disallowed; otherwise, the element is one. We refer to this matrix as a *CSP matrix*.

The rationale behind using a CSP matrix is the observation that it resembles a 2-D array of pixel values in a gray-scale image. cNNs are known to recognize patterns in a multi-dimensional array of numbers, such as patterns in an image in computer vision applications. Our intuition is that many properties of a CSP depend on the patterns of compatible and incompatible tuples in its constraints. Therefore, we expect a cNN that takes a CSP matrix as input to be able to recognize patterns of compatible and incompatible tuples to make its predictions.

Our cNN has the following architecture. Each node in the input layer corresponds to an element of the input CSP matrix. It has 4 convolutional layers with 3-by-3 kernels with stride 1, each of which is followed by a MaxPool layer with a 2-by-2 kernel with stride 2. We used “same” padding for all convolutional layers and “valid” padding for all MaxPool layers. Following these layers, there are 2 fully connected hidden layers. Finally, there is an output layer with a single node. The node in the output layer uses the *sigmoid activation function*, and all other neurons are *rectified linear units* (ReLU) [13]. The output layer uses L2 regularization with a coefficient of 0.1, and all other layers use L2 regularization with a coefficient of 0.01. We refer to this cNN as CSP-cNN as shown in Fig. 1.

2.1 Efficient Massive Training Data Generation

One of the key reasons for the success of deep learning is its power to use huge amounts of training data, such as hundreds of thousands of data points. However, since CSPs are NP-hard, it is in general elusive to generate such a huge amount of labeled data. In this subsection, we develop a new method that efficiently generates massive amounts of labeled data.

We generalize Model A [26] to create a random binary CSP generation model, henceforth referred to as generalized Model A. Model A generates a binary CSP as follows [26]. It independently selects each one of the $n(n-1)/2$ pairs of variables with a given probability p , and, for each selected pair of variables X_i and X_j , it marks each one of the $|D_i| \cdot |D_j|$ possible pairs of values as incompatible independently with a given probability q . Here, p characterizes how many constraints exist in a CSP, and q characterizes how restrictive the constraints are. In generalized Model A, q can vary from constraint to constraint (denoted by q_{ij} for the pair of variables X_i and X_j). Model A has an important property: It always generates CSPs that are unsatisfiable when $n \rightarrow \infty$ if $p, q > 0$ [2]. Generalized Model A also has this property if $p > 0$ and $\forall X_i, X_j \in \mathcal{X} : q_{ij} > 0$, since it generates CSPs that are more constrained than those generated by Model A with the same p and $q = \min_{X_i, X_j \in \mathcal{X}} q_{ij}$.

By making use of this property of generalized Model A, we are able to generate data points with a low mislabeling rate as follows. To generate a data point with label UNSATISFIABLE, we simply follow generalized Model A with non-zero p and q_{ij} 's. To generate a data point with label SATISFIABLE, we use the same procedure but update the compatibilities of tuples in generated constraints to allow for a randomly selected solution. We refer to this data generation method as *generalized Model A-based method* (GMAM). To avoid data imbalance, we generate comparable numbers of data points labeled SATISFIABLE and UNSATISFIABLE. Using this approach, we can efficiently generate huge amounts (such as millions) of labeled data points for training. The main intuitive reason that we use generalized Model A instead of Model A is that it leads to a distribution of CSPs that is more spread out and may be beneficial for training cNNs.

Although generalized Model A always generates CSPs that are unsatisfiable when $n \rightarrow \infty$ if $p > 0$ and $\forall X_i, X_j \in \mathcal{X} : q_{ij} > 0$, it is still desirable to have some bounds on the probability of mislabeling a CSP for finite n , which can be used to guide the choice of p and q_{ij} . Among all data points labeled SATISFIABLE, there are no mislabeled data points since a solution is guaranteed during data generation. For a data point labeled UNSATISFIABLE, we prove that:

Theorem 1. *Consider a data point with binary CSP $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$. If it is generated using GMAM and is labeled UNSATISFIABLE, the probability of it being mislabeled is no greater than $\prod_{X_i \in \mathcal{X}} |D(X_i)| \prod_{X_i, X_j \in \mathcal{X}} (1 - pq_{ij})$.*

Proof. The probability of mislabeling the CSP equals the probability that it has at least one solution, denoted by $P(n_{\text{sol}} \geq 1)$. Using Markov's inequality, we

have $P(n_{\text{sol}} \geq 1) \leq \mathbb{E}(n_{\text{sol}})$, where $\mathbb{E}(n_{\text{sol}})$ is the expected number of solutions of the CSP. We also have

$$\begin{aligned} \mathbb{E}(n_{\text{sol}}) &= \mathbb{E}\left(\sum_{a \in \mathcal{A}(\mathcal{X})} \mathbb{1}_{a \text{ is a solution}}\right) = \sum_{a \in \mathcal{A}(\mathcal{X})} \mathbb{E}(\mathbb{1}_{a \text{ is a solution}}) \\ &= \sum_{a \in \mathcal{A}(\mathcal{X})} P(a \text{ is a solution}) = \sum_{a \in \mathcal{A}(\mathcal{X})} \prod_{X_i, X_j \in \mathcal{X}} (1 - pq_{ij}) = \prod_{X_i \in \mathcal{X}} |D(X_i)| \prod_{X_i, X_j \in \mathcal{X}} (1 - pq_{ij}), \end{aligned}$$

where $\mathcal{A}(\mathcal{X})$ is the set of all assignments of values to variables in \mathcal{X} . Therefore, the probability that a data point labeled UNSATISFIABLE is mislabeled is no greater than $\prod_{X_i \in \mathcal{X}} |D(X_i)| \prod_{X_i, X_j \in \mathcal{X}} (1 - pq_{ij})$. \square

2.2 Training and Prediction on General CSP Datasets

Applying a deep NN to a small dataset directly may cause overfitting due to the large number of training parameters. Although we can use GMAM to efficiently generate huge amounts of training data, training a deep NN on a dataset from a distribution different from the dataset of interest usually does not lead to good results, even if the training dataset is huge. To overcome this issue, there are two common classes of techniques: *domain adaptation* and *data augmentation*.

Domain adaptation refers to learning from one source of data and predicting on a different source of data with a different distribution, due to the scarcity of available labeled data from the latter source. By using domain adaptation techniques, a small set of labeled data of interest, assumingly generated from an arbitrary distribution different from generalized Model A, can still be made viable. In particular, we can train on a mix of these available data and data generated using GMAM, and then evaluate on the test data of interest.

Data augmentation refers to transforming data without changing their labels, known as *label-preserving transformations*. For example, in object recognition tasks in computer vision applications, to generate more training data, we can augment an image by translating or reflecting horizontally without changing its label [17]. In the context of CSP-cNN, we can augment an input CSP by changing the order of variables or their domain values, i.e., exchanging their corresponding rows and columns of the CSP matrix. This does not alter the satisfiabilities of the CSPs and therefore does not change their labels.

3 Experimental Evaluation

We evaluated CSP-cNN and the relevant methods mentioned above experimentally. We used Keras [7] with the TensorFlow [1] backend, that uses the GPU to accelerate forward and backward propagation to implement NNs.

Evaluation on Data Generated Using GMAM. Using GMAM, we generated 200,000 training data points, 10,000 validation data points, and 10,000 test data points. Each data point is a binary CSP of 128 Boolean variables limited

by the computational capacity of our hardware and the size of our CSP-cNN. For each data point, we randomly chose p and all q_{ij} 's between 0.12 and 0.99. Theorem 1 guarantees that the probability of mislabeling an UNSATISFIABLE data point is $\leq 2^{128} \times (1 - 0.12 \times 0.12)^{128 \times (128-1)/2} = 2.14 \times 10^{-13}$. Half of the data points in each of the training, validation, and test datasets are labeled SATISFIABLE and the others UNSATISFIABLE.

We first trained our CSP-cNN using the training data generated above. We initialized all parameters using *He-initialization* [14]. We trained our CSP-cNN using *stochastic gradient descent* (SGD) with a mini-batch size of 128 for 59 epochs. In each epoch, we randomly shuffled all data points. We used a *learning rate* of 0.01 for the first 5 epochs and a learning rate of 0.001 for the last 54 epochs. We used *binary cross entropy* as the loss function. Each epoch took about 520 seconds to finish on a GPGPU “NVIDIA(R) Tesla(R) K80.”

After training, all training, validation, and test accuracies were greater than 99.99%. Therefore, we conclude that, while constraint tightnesses and constraint densities do not determine the satisfiabilities of CSPs, deep NNs, such as our CSP-cNN, can be capable of accurate predictions when a huge amount of training data are available, at least on Boolean binary CSPs.

To further demonstrate the effectiveness of our CSP-cNN on GMAM generated data, we also compared our CSP-cNN with three other NNs. The first NN, referred to as NN-image, had the same architecture as our CSP-cNN, but its input was a gray-scale image converted from the ASCII codes of its input text file as described in [19]. The other two NNs were *plain*, i.e., had only fully connected hidden layers. The first plain NN, referred to as NN-1, had only 1 fully connected hidden layer with 256 ReLUs and 1 output layer with a single neuron with a sigmoid activation function, i.e., the last two layers of our CSP-cNN. This is a classical shallow NN architecture. The second plain NN, referred to as NN-2, was constructed by inserting 1 more fully connected hidden layer with 1024 ReLUs after the input layer in NN-1. Both plain NNs used the same parameter initialization and regularization as our CSP-cNN. We trained NN-image using a training procedure similar to that of CSP-cNN except that it used one more epoch with a learning rate of 0.01. We trained both plain NNs for 120 epochs using SGD with a learning rate of 0.01 for the first 60 epochs and 0.001 for the last 60 epochs. They both used a mini-batch size of 128.

Our experimental results are shown in Table 1. The test accuracy of our CSP-cNN was better than those of NN-1 and NN-2 and far better than that of NN-image. Thus, the CSP matrix of a CSP seems to provide a better input representation than the approach in [19] and seems to reveal useful structure of the CSP. We also compared with an approach that predicts a CSP’s satisfiability using its number of incompatible tuples, referred to as “M” in Table 1. It selects a threshold and predicts CSPs with a number of incompatible tuples above this threshold to be unsatisfiable and other CSPs to be satisfiable. The best threshold for the test data is 13435 and led to an accuracy of 64.79%.

Table 1. Test accuracies on GMAM generated data.

	CSP-cNN	NN-image	NN-1	NN-2	M
Acc (%)	>99.99	50.01	98.11	98.66	64.79

Table 2. First two columns show test accuracies of CSP-cNN in all 3 rounds of cross validation. Last column shows the test accuracy of CSP-cNN (trained on GMAM generated data) on MMEM generated data.

	Data Trained	Mixed	MMEM	GMAM
Acc (%)	100.00/100.00/100.00	50.00/50.00/50.00	50.00	50.00

Evaluation of Domain Adaptation and Data Augmentation. Due to the lack of small ($n \leq 128$) benchmark instances of Boolean binary CSPs where satisfiabilities need to be determined, we randomly generated 1,200 binary CSPs with 128 Boolean variables using a model similar to Model E [2]. We generated only 1,200 CSPs to mimic most real-world scenarios where labeled CSPs are costly to obtain. We generated these CSPs as follows. We divided all CSPs into two groups. For each CSP in the first group, (a) we divided the 128 Boolean variables into two partitions, with 64 variables each; (b) for each pair of variables from different partitions, we randomly added a binary constraint between them with probability 0.99; (c) within each constraint, we randomly marked exactly 2 (out of 4) tuples as incompatible. For each CSP in the second group, we generated it using a similar approach, except that, in Step (c), we also guaranteed that incompatible tuples do not rule out a randomly generated solution (while it remains that exactly 2 tuples are incompatible in each constraint). We refer to this random CSP generation method as *Modified Model E-based method* (MMEM). Using Choco [22], we labeled 600 CSPs SATISFIABLE and 600 UNSATISFIABLE.

The distribution of CSPs resulting from using MMEM is different from that of the ones resulting from using GMAM, for the following reasons. There are formally proven significant differences between the asymptotic satisfiability properties of Model A and Model E [2]. Step (a) yields a bipartite variable interaction structure. Step (c) guarantees the same tightness in each constraint, which makes the satisfiabilities of the CSPs unrecognizable from their tightnesses. For these reasons, these random CSPs suffice for a proof of concept.

We evaluated the effectiveness of domain adaptation and data augmentation for our CSP-cNN on the CSPs generated by MMEM using stratified 3-fold cross validation, i.e., we divided the 1,200 CSPs into 3 sets with equal numbers of satisfiable and unsatisfiable CSPs. Since there are only 400 data points in each of these 3 sets, for each data point in the training set, we used the augmentation method in Sect. 2.2 124 times to produce 124 more data points each. Therefore, in each round of cross validation, we used $125 \times 800 = 100,000$ training data points. We mixed these 100,000 data points with the 200,000 data points generated using GMAM and trained our CSP-cNN on them. We used SGD and trained for 30 epochs. We used a learning rate of 0.01 in the first 10 epochs and a learning rate of 0.001 in the last 20 epochs. As a baseline, we also trained CSP-cNN by augmenting each data point for 324 times so that the number of training data

Table 3. Test accuracies of all three rounds of cross validation for different percentages of MMEM generated data when domain adaptation is used.

Percentage of MMEM (%)	0.00	33.33	36.00	40.00	46.66	53.33	66.67	70.67	78.67	100.00
Average Accuracy (%)	50.00	100.00	100.00	83.33	66.67	83.33	66.67	66.67	50.00	50.00

points was also $325 \times 800 = 300,000$ in each round. The training procedure was the same. We also directly applied the CSP-cNN previously trained on GMAM generated data to all 1,200 data points.

Our experimental results are shown in Table 2. Our mixed data points produced test accuracies of 100% in all three rounds of cross validation. On the other hand, CSP-cNNs trained only on augmented MMEM or GMAM generated data have high test errors in (cross) validation and always produced the same prediction regardless of their input. When training our CSP-cNN only on augmented MMEM generated data, we were unable to reduce the test error even by tuning hyperparameters, such as the learning rate, initialization, and the optimization algorithm. This shows that our GMAM generated data seem to play a key role in enabling effective deep learning for CSPs via domain adaptation. To further confirm this, we ran similar experiments with various percentages of MMEM generated data in the training data by varying the number of times each data point is augmented. In these experiments, GMAM generated data points were randomly selected to fill the total number of training data points to 300,000. Table 3 shows our experimental results. When MMEM generated data points constituted 33.33–36.00% of the training data, the average test accuracies reached 100%. However, when MMEM generated data points constituted more than 40.00% of the training data, the test accuracies became lower and unstable.

4 Conclusions and Future Work

In this paper, we effectively applied our CSP-cNN, a deep NN architecture, to predict satisfiabilities of CSPs with prediction accuracies higher than 99.99%. To the best of our knowledge, this is the first effective application of deep learning to random CSPs whose constraint tightnesses and constraint densities do not determine their satisfiabilities. Due to the NP-hardness of CSPs, training data are usually too scarce to be effectively used by deep learning. We addressed this issue by generating huge amounts of labeled data using GMAM. We experimentally demonstrated the high effectiveness (>99.99% test accuracy) of applying our CSP-cNN to these data on random Boolean binary CSPs. While these CSPs are known to be in P, we used them as an initial demonstration. For CSPs drawn from a distribution different from that of GMAM, we once again addressed the issue of lack of training data. We did this by augmenting the training data and mixing them with GMAM generated CSPs. Finally, we experimentally demonstrated the superior effectiveness of these techniques on MMEM generated CSPs.

So far, we have only experimented on small easy random CSPs that were generated in two very specific ways. One future research direction is to understand

the generality of our approach, for example, by experimenting on larger, hard, and real-world CSPs, analyzing what our CSP-cNN learns, and evaluating how robust our approach is with respect to the training data and hyperparameters. A second future research direction is to understand exactly how our approach should be used, for example, how the effectiveness of our CSP-cNN depends on the amount of available training data and the amount of data augmentation used to increase them. A third future research direction is to generalize our CSP-cNN to accommodate more types of constraints. (a) For non-binary table constraints, we could naively increase the dimensionality of the CSP matrix to be equal to the maximum arity of the constraints. A more practical method might be to represent input CSPs as constraint graphs and adapt the graph representation methods in [20]. (b) For symmetric global constraints, we could adapt the methods that apply recurrent NNs (rNNs) to Boolean satisfiability [25]. Then, an NN architecture that combines cNNs and rNNs could be used.

References

1. Abadi, M., et al.: TensorFlow: large-scale machine learning on heterogeneous systems. software (2015). <https://www.tensorflow.org/>
2. Achlioptas, D., Molloy, M.S.O., Kirousis, L.M., Stamatiou, Y.C., Kranakis, E., Krizanc, D.: Random constraint satisfaction: a more accurate picture. *Constraints* **6**(4), 329–344 (2001). <https://doi.org/10.1023/A:1011402324562>
3. Amadini, R., Gabbrielli, M., Mauro, J.: An empirical evaluation of portfolios approaches for solving CSPs. In: The International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming, pp. 316–324 (2013). https://doi.org/10.1007/978-3-642-38171-3_21
4. Amadini, R., Gabbrielli, M., Mauro, J.: An enhanced features extractor for a portfolio of constraint solvers. In: The Annual ACM Symposium on Applied Computing, pp. 1357–1359 (2014). <https://doi.org/10.1145/2554850.2555114>
5. Arbelaez, A., Hamadi, Y., Sebag, M.: Continuous search in constraint programming. In: The IEEE International Conference on Tools with Artificial Intelligence, pp. 53–60 (2010). <https://doi.org/10.1109/ICTAI.2010.17>
6. Bourlard, H.A., Morgan, N.: *Connectionist Speech Recognition*. Springer, New York (1994). <https://doi.org/10.1007/978-1-4615-3210-1>
7. Chollet, F., et al.: Keras (2015). <https://keras.io>
8. Ciregan, D., Meier, U., Schmidhuber, J.: Multi-column deep neural networks for image classification. In: The IEEE Conference on Computer Vision and Pattern Recognition, pp. 3642–3649 (2012). <https://doi.org/10.1109/CVPR.2012.6248110>
9. Galassi, A., Lombardi, M., Mello, P., Milano, M.: Model agnostic solution of CSPs via deep learning: a preliminary study. In: The International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research, pp. 254–262 (2018). https://doi.org/10.1007/978-3-319-93031-2_18
10. Gent, I.P., et al.: Learning when to use lazy learning in constraint solving. In: The European Conference on Artificial Intelligence, pp. 873–878 (2010). <https://doi.org/10.3233/978-1-60750-606-5-873>
11. Goodfellow, I., Bengio, Y., Courville, A.: *Deep Learning*. MIT Press, Cambridge (2016)

12. Guerri, A., Milano, M.: Learning techniques for automatic algorithm portfolio selection. In: The European Conference on Artificial Intelligence, pp. 475–479 (2004)
13. Hahnloser, R.H.R., Sarpeshkar, R., Mahowald, M.A., Douglas, R.J., Seung, H.S.: Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature* **405**, 947–951 (2000). <https://doi.org/10.1038/35016072>
14. He, K., Zhang, X., Ren, S., Sun, J.: Delving deep into rectifiers: surpassing human-level performance on ImageNet classification. In: The IEEE International Conference on Computer Vision, pp. 1026–1034 (2015). <https://doi.org/10.1109/ICCV.2015.123>
15. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: ISAC - instance-specific algorithm configuration. In: The European Conference on Artificial Intelligence, pp. 751–756 (2010). <https://doi.org/10.3233/978-1-60750-606-5-751>
16. Kotthoff, L.: Algorithm selection for combinatorial search problems: a survey. In: Data Mining and Constraint Programming: Foundations of a Cross-Disciplinary Approach, pp. 149–190 (2016). https://doi.org/10.1007/978-3-319-50137-6_7
17. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: The Neural Information Processing Systems Conference, pp. 1097–1105 (2012). <https://doi.org/10.1145/3065386>
18. LeCun, Y.: Backpropagation applied to handwritten zip code recognition. *Neural Comput.* **1**(4), 541–551 (1989). <https://doi.org/10.1162/neco.1989.1.4.541>
19. Loreggia, A., Malitsky, Y., Samulowitz, H., Saraswat, V.: Deep learning for algorithm portfolios. In: The AAAI Conference on Artificial Intelligence, pp. 1280–1286 (2016)
20. Niepert, M., Ahmed, M., Kutzkov, K.: Learning convolutional neural networks for graphs. In: The International Conference on Machine Learning, pp. 2014–2023 (2016)
21. O’Mahony, E., Hebrard, E., Holland, A., Nugent, C., O’Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: The Irish Conference on Artificial Intelligence and Cognitive Science (2008)
22. Prud’homme, C., Fages, J.G., Lorca, X.: Choco Documentation. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. (2017). <http://www.choco-solver.org>
23. Pulina, L., Tacchella, A.: A multi-engine solver for quantified Boolean formulas. In: The International Conference on Principles and Practice of Constraint Programming, pp. 574–589 (2007). https://doi.org/10.1007/978-3-540-74970-7_41
24. Sateesh Babu, G., Zhao, P., Li, X.L.: Deep convolutional neural network based regression approach for estimation of remaining useful life. In: The International Conference on Database Systems for Advanced Applications, pp. 214–228 (2016). https://doi.org/10.1007/978-3-319-32025-0_14
25. Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., Dill, D.L.: Learning a SAT solver from single-bit supervision. [arXiv:1802.03685](https://arxiv.org/abs/1802.03685) [cs.AI] (2018)
26. Smith, B.M., Dyer, M.E.: Locating the phase transition in binary constraint satisfaction problems. *Artif. Intell.* **81**(1), 155–181 (1996). [https://doi.org/10.1016/0004-3702\(95\)00052-6](https://doi.org/10.1016/0004-3702(95)00052-6)
27. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.* **32**, 565–606 (2008). <https://doi.org/10.1613/jair.2490>

CP and Music Track



Extending the Capacity of $1/f$ Noise Generation

Guillaume Perez^(✉), Brendan Rappazzo, and Carla Gomes

Department of Computer Science, Cornell University, Ithaca, NY 14850, USA
guillaume.perez06@gmail.com, bhr54@cornell.edu

Abstract. From the emissions of massive quasars scattered across the universe, to the fluctuations in the stock market and the melodies of music, several real world signals have a power spectral density (PSD) that follows an inverse relationship with their frequency. Specifically, this type of random process is referred to as a $1/f$ signal, and has been of much interest in research, as sequences that have this property better mimic natural signals. In the context of constraint programming, a recent work has defined a constraint that enforces sequences to exhibit a $1/f$ PSD, as well as a corresponding constraint propagator. In this paper we show that the set of valid solutions associated with this propagator misses an exponential number of $1/f$ solutions and accepts solutions that do not have a $1/f$ PSD. Additionally, we address these two issues by proposing two non-exclusive algorithms for this constraint. The first one can find a larger set of valid solutions, while the second prevents most non- $1/f$ solutions. We demonstrate in our experimental section that using the hybrid of these two methods results in a more robust propagator for this constraint.

1 Introduction

The power spectral density (PSD) of a signal describes the distribution of power over its consisting frequency components, and is an extremely useful metric for the analysis of stochastic processes. Interestingly, the PSD of many real world phenomena, including quasar emissions, the firing of neurons and the resistivity of semiconductors, exhibit a signal that gives a PSD that is inversely proportional to its frequency [7]. To be precise this means that the PSD value follows a $1/f$ shape, where f is the frequency, and is called $1/f$ noise accordingly. The synthetic generation of $1/f$ signals can be of much use in content generation, for example it can help improve the digital generation of images [14]. Another example, and one of the most interesting phenomena observed, is that music that follows a PSD of $1/f$ empirically sounds better than music that has other distributions [23, 24]. Specifically, music generated using $1/f$ sequences sounds less artificial [5]. Given the potential benefits of generating $1/f$ noise, several methods have been proposed [6, 7]. A notably interesting algorithm is the Voss algorithm, which was developed by the physicist Richard Voss and then later published by Gardner [4]. While this method can generate sequences that have

the $1/f$ property, it only captures a subset of possible solutions [1]. Furthermore, some of the generated sequences are not guaranteed to be $1/f$. One particularly interesting attempt in CP to embed the $1/f$ property into generated music uses the Voss algorithm to produce the Voss constraint [9]. This is interesting as while there are many works in CP that develop efficient models for generating music and text [3, 9–11, 18, 21, 22], embedding the constraint of generating $1/f$ noise in constraint programming solvers has been less explored. Specifically, the Voss constraint is a hard constraint defined over a list of variables that enforces that the assignment of the variables follows $1/f$ noise and utilizes the Voss algorithm. It therefore inherits the limitations of the Voss algorithm, namely the fact that it misses several valid sequences and can generate invalid solutions. The first issue of missing valid sequences is a problem that results from using the same initial synchronization for all sequences, and of hard encoding the frequency of change in the sequence. The second issue of generating invalid solutions is a typical problem of probabilistic modeling in CP.

Out Contributions: (1) We propose to tackle the synchronization problem by allowing a “shifting effect” in our constraint and can thus generate an exponentially larger set of $1/f$ sequences [1]. (2) Additionally, we address the frequency problem by modeling the problem in a completely probabilistic way. We note that the use of the statistical properties to enforce constraints has been demonstrated in many works [8, 10, 12, 13, 16, 17, 19, 20]. (3) We address the second issue of not generating invalid solutions by proposing a hybrid method of our two models. (4) We demonstrate the better performance of our two methods and hybrid method experimentally, by generated sequences.

2 Preliminaries

$1/f$ *Spectrum* Fourier Analysis is an extremely useful analysis tool that decomposes a signal into its frequency components. In particular, one powerful metric that uses Fourier Analysis, called power spectral density (PSD), analyzes how the power of a signal is distributed over its frequency components. Typically, the metric is displayed graphically with the power of the signal plotted against frequency. The PSD of a signal can give intuition into how the signal behaves, and if it is primarily made up of low or high frequencies. For example, a signal with a flat PSD, meaning an even power distribution across all frequencies, represents an entirely random process. A real world example of this kind of signal would be white noise, where each successive value in the sequence is picked in a random way. On the opposite end of the spectrum is Brownian Motion, which is a process where each successive value in a sequence is assigned by a small random fluctuations in the previous value. The result is that Brownian Motion is primarily described by lower frequencies and thus the PSD of such a signal follows the curve of $\frac{1}{f^2}$, where f is frequency. Using the same style of description, i.e. $\frac{1}{f^\alpha}$, white noise is simply the case when $\alpha = 0$ and Brownian noise is described when $\alpha = 2$. Thus $1/f$ spectrum is just the case in this description where $\alpha = 1$, and is sometimes called pink noise. Intuitively, it describes a signal

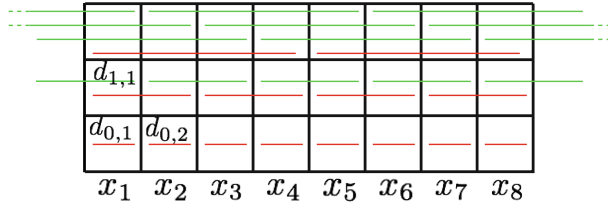


Fig. 1. Each segment represents a rolled die value. In red, no shift (previous model). In green, all possible shifts (our model). (Color figure online)

that is a hybrid of white noise and Brownian motion, in that it is mostly made up of low frequencies, but still has some high frequency components. As discussed before $1/f$ noise is an interesting special case because many natural signals have PSD's that follow the $1/f$ curve. For this reason it is highly valuable to have an algorithm capable of generating $1/f$ sequences.

Voss Algorithm. The Voss algorithm generates $1/f$ sequences by modeling the problem using dice. Essentially, to generate a sequence of length n it uses n iterations of rolling $\approx \log_2(n)$ dice, where each die is re-rolled every 2^i turns and i is the number of the die. At each iteration, the value assigned to the corresponding variable in the sequence is given by the summation of the dice values. To better illustrate the algorithm we unroll it in the table on the right of this paragraph to generate a $1/f$ sequence of length 8, using 3 dice referred to as the red (R), green (G) and blue (B) die as described by Gardner [4].

In the first iteration the R, G and B dice are rolled and have values of 3, 2 and 5 respectively. The first value in the sequence is just the summation of the three values and is thus ten. Additionally, the Binary representation of the iteration number is shown, where each bit corresponds to a die. Since we are starting at iteration 0, all the bits are set to 0. For each successive iteration, the binary representation is incremented by 1, to reflect the new iteration value. Then the dice are re-rolled if their respective bit in this representation flips value, resulting in one, two or all three dice changing value. Lastly, the new dice values are summed and the resulting value is assigned to the next variable in the sequence. This algorithm provides a framework for generating $1/f$ sequences by assigning each successive variable each iteration, but in order to be used in CP it needs to be adapted into a constraint.

Iter	Binary	Dice Roll	Val
i	R G B	R G B	v
0	0 0 0	3 2 5	10
1	0 0 1	3 2 4	9
2	0 1 0	3 1 2	6
3	0 1 1	3 1 6	10
4	1 0 0	6 5 3	14
5	1 0 1	6 5 4	15
6	1 1 0	6 1 2	9
7	1 1 1	6 1 5	12

Voss Constraint. The Voss constraint has been introduced in [9] and is named after the physicist Richard Voss, because its implementation uses the Voss algorithm. It is defined in the following way, let $X = \{x_1, x_2, \dots, x_n\}$ be a list of

variables, let K be the number of dice and R be the maximum value of the dice. Then the Voss constraint, applied to X , ensures that the values of the variables X could have been obtained by rolling dice following the Voss algorithm [4]. Red segments from Fig. 1 represent rolled dice values. As shown, the rolled die value $d_{0,1}$ is defined only for x_1 , while the rolled die value $d_{1,1}$ is define for both x_1 and x_2 . For a given variable, its value is defined by the sum of all the rolled dice values above. For example $x_1 = d_{0,1} + d_{1,1} + d_{2,1}$, while $x_5 = d_{0,5} + d_{1,3} + d_{2,2}$. Specifically, the constraint is defined by the following: let d be the list of dice and let $d_{i,j}$ be the j^{th} rolling value of the i^{th} die then solutions of the constraint ensure that variables in $x_i \in X$ follow the equation:

$$x_i = \sum_{j=0}^{K-1} d_{j, \lceil \frac{i}{2^j} \rceil} \tag{1}$$

The tree shaped structure emanating from the red segment is called the Voss tree, and it represents all the possible solutions of the propagator proposed by [9]. Their propagator consists in constructing the tree shaped ternary sum constraint network. While this method does generate $1/f$ sequences, it only captures a subset of possible solutions [1] as it does not consider different initial conditions or what this paper refers to as dice shifting.

3 Shifted Dice

As known and explicated in [1], the dice rolling configuration from [9] is not the only method to generate $1/f$ sequences. In the original method each die is re-rolled exactly every 2^i turns, where i is the number of the die. While this ensures the generation of $1/f$ noise, it fails to consider that the dice do not need to have the same count for the number of turns. For example die 2 and die 3 should be re-rolled every 2^2 and 2^3 turns respectively, but there is no reason they must have the same count of turns. By allowing for a different counting of turns for each die, this essentially allows for the dice to be shifted with respect to one another. In terms of the algorithm show in 2, this has the same effect as starting with a non zero iteration, i.e. non zero binary representation. In this way, the previous method is actually a special case of the shifted version, where all the dice are in phase with one another. Whereas, in this method we allow each die to be treated independently and we allow for all possible initial shifted conditions.

Green segments from Fig. 1 represent the missing shifts. From a CP point of view, this implies that the proposed definition and propagator are too restrictive and miss an exponential number of solutions. In this section, we aim to model this shifting behaviour and translate it into constraints. Given a vector $S = \{s_1, \dots, s_K\}$ representing the shift of each die, the general formula value of a variable is given by:

$$x_i = \sum_{j=0}^{K-1} d_{j, \lceil \frac{i+s_j}{2^j} \rceil} \tag{2}$$

We used a modified formula derived from [9] as it allows us to have each die indexed by its frequency. First, the S values represent the current state of a die. It is important to note that values of s_j that are $\geq 2^j$ or are $s_j < 0$, are equivalent to the value $s_j \bmod 2^j$. Let the following conditional variable C be defined by: $C = i + s_j < 2^j$. Then the equation for any variable becomes:

$$x_i = \sum_{j=0}^{K-1} C d_{j, \lceil \frac{i}{2^j} \rceil} + \bar{C} d_{j, 1 + \lceil \frac{i}{2^j} \rceil} \tag{3}$$

Proposition 1. *Domain propagation of the Voss constraint with shift is NP-Complete.*

Proof. Consider the subset sum problem, let $V = \{v_1, v_2, \dots, v_k\}$ be the set possible values and a be the desired value for the sum. The subset sum problem consists in finding a subset $V' \in V$ such that $\sum_{v \in V'} v = a$. Consider the Voss constraint, applied to the variables list X , using the following fixed values for the dice, $d_{i,1} = v_i$ and $d_{i,2} = 0$ for all i in $[1, k]$. Let $x_2 = a$, enforcing domain consistency implies solving the subset sum problem. The hint here is that for a given variable we will have to choose between the non-shifted dice or the next one, resulting in an exponential number of choices.

While this propagator can generate an exponentially larger set of solutions compared to the previous definition, it still fails to generate all the $\approx 1/f$ sequences. This is mainly because even this shifted version is a $1/f$ approximation algorithm, and the exact distance of 2^i for each die is restrictive. Moreover, by only defining the dice variable to be independent, this may end up in sequences that are not $1/f$, because we do not enforce the dice to actually change value. This means that it is possible, for example, that a dice i could have the same value for all variables, which would give a signal that does not have a $1/f$ PSD. To combat these issues we developed a probabilistic version of this model that can enforce the dice to change value.

4 Probabilistic Dice

The important part of the Voss algorithm is the frequency in which dice are re-rolled. In this section we translate the re-rolling frequency into a probability, and use the probability as a constraint. First, we define the notion of state, which represents the current value of the dice. A state, defined by S , is a list of value $S_i \in [a, b]$. Thus each S_i represents the value of a die. Then, the value of a variable x_j is given by: $x_j = \sum_{i=0}^{\lfloor S^j \rfloor} S_i^j$.

The frequency of re-rolling dice i is 2^i , thus the probability of modifying S_i , from one state to another is $P(S_i^j \neq S_i^{j+1}) = \frac{1}{2^i}$. This gives us the probability to change the value of each element of S . The new values are chosen using a uniform random distribution, $\forall v \in [a, b], P(v) = \frac{1}{b-a+1}$. Using the probability of changing a value and the probability for getting a new value, we define the state transition process.

Definition 1. Given a state S^t , the next state S^{t+1} is defined as follow:

$$\forall v \neq S_i^t, P(S_i^{t+1} = v) = P(S_i^t \neq S_i^{t+1}) * P(v) = \frac{1}{2^i(b-a+1)} \tag{4}$$

$$P(S_i^{t+1} = S_i^t) = 1 - \sum_{j=1}^{b-a} P(S_i^t \neq S_i^{t+1}) * P(j) = 1 - \frac{b-a}{2^i(b-a+1)} \tag{5}$$

Let M (Eqs. (4, 5)) be a Markov process, let $d_i, \forall i \in [1, P]$ be the a list of variables of size n representing the value of the i^{th} die for each variable. We define a Markov constraint, using M and its automaton, by list d_i with lower and upper bounds on the product of their probability [12, 13]. By using the bounding of the probability as a constraint we can control the number of times the dice are re-rolled. Using this definition, we can generate all possible solutions, in fact without any constraint on the bound, we can generate any sequence, even non-1/f solutions.

The density of 1/f solutions becomes a problem with this method as the sequence length becomes large. By setting an upper bound and/or a lower bound on the probability, we can constrain the number of time a dice is rolled to a new value. However, we do not constrain, how the re-rolls are distributed throughout the sequence. Given a die i with probability $\frac{1}{2^i}$ to be rolled, we can accept $k = \lceil \frac{n}{2^i} \rceil$ re-rolling over a sequence of size n . The number of possible sequences where the die is rerolled k times is $\binom{n}{k}$. Consider that we are looking for the sequence where the space between two re-rolls is exactly 2^i .

Proposition 2. Let the solution density be defined by $D(n, i) = \frac{2^i}{\binom{n}{k}}$. Then

$$\lim_{n \rightarrow \infty} D(n, i) = 0$$

Proof. Since 2^i is fixed, we need to prove that $\binom{n}{k}$ grows to infinity as n grows. While k does depend on n , it can only have two values when n is incremented, k or $k + 1$. Thus we need to prove both that $\binom{n}{k} < \binom{n+1}{k}$ and $\binom{n}{k} < \binom{n+1}{k+1}$. The first case is trivial. For the second case, we have

$$\binom{n}{k} \cdot \frac{n+1}{k+1} = \frac{n!}{k!(n-k)!} \cdot \frac{n+1}{k+1} = \frac{(n+1)!}{(k+1)!(n-k)!} = \binom{n+1}{k+1}$$

Finally, $k < n \implies \frac{n+1}{k+1} > 1 \implies \binom{n}{k} < \binom{n+1}{k+1}$.

Thus with longer sequences, it becomes more likely that this method will find a non-1/f solution. Such an issue can be solved by using higher order Markov processes and manually remove some state/transition, which is close to define a regular constraint on the time-series constraint [2] on die using the regular expression $=^{[0, 2^i]} \neq (=^{2^i \pm \epsilon} \neq)^*$. For example, if $\epsilon = 1$ the generated sequences have distances of 2^{i-1} , 2^i , or 2^{i+1} between reroll. Given that the shifted version of our propagator cannot constrain that the dice actually change value, and

that the probabilistic model cannot constrain the distributions of dice re-rolls, it seemed natural that a hybrid method of the two would give the best results.

The Hybrid Method. In order to solve the issue of solution density in the probabilistic model, and the issue of non- $1/f$ solutions produced by the dice shifting model, we propose to combine the two methods into a single hybrid method. The hybrid method we propose uses the model defined for the shifted dice, but in addition, applies the Markov constraint to the shifting dice. In this way the Markov constraint will be used to give a lower bound to the probability that the dice change value in the shifted dice framework. Specifically we know that the probability to change a dice value is $p_c = \frac{b-a}{b-a+1}$, and to keep the same value is $p_k = \frac{1}{b-a+1}$. One of the advantages to this method is that there are less variables in the d_i lists from the shifted version than the probabilistic model. However, one drawback of this method is that we are enforcing there to be exactly 2^i turns between re-rolling, which is restrictive.

5 Sequence Generation

In the same manner as [9], we sample $1/f$ sequences using a CP solver and the newly introduced propagators. Our goal is to show the following points: **(1)** The shifted version of our propagator generates $1/f$ sequences, while being able to generate a larger scope of solution than its predecessor. **(2)** The probabilistic version, while working well outside of CP, fails to generate $1/f$ sequences. **(3)** Our hybrid approach is able to generate $1/f$ sequences, and is more robust than the simple shifted version.

Protocol. We aim at generating sequences of length 10,000 using $13 \approx \log_2(10,000)$ dice. We use Google ORtools solver [15], and we use arithmetic constraints for the shifted model and the Markov constraint [13] for the probabilistic model. The search is a fully random one.

Results. We tested our three methods, the shifted dice, the probabilistic formulation and the hybrid method, both in CP solvers and in a stand alone process. Specifically we want to compare how the fixed frequency methods, i.e. the shifted dice, and hybrid method, compare to the probabilistic method. As seen in Fig. 3 we show the PSD of the original algorithm, the shifted dice version and the hybrid method, all plotted in log-log scale, where a line with slope -1 indicates $1/f$ noise. As can be seen in the figure the shifted dice method successfully produces $1/f$ noise both using a CP solver and as a standalone process. Additionally, the hybrid method successfully produces $1/f$ noise using a CP solver. This result is expected as the shifted dice method is simply a generalization of the original method which was already shown to produce $1/f$ sequences. With these results we believe we have shown that our general shifted dice method, and our hybrid method successfully generate $1/f$. Further, because of their theoretical formulation we know that they are capable of capturing an exponentially larger set of viable $1/f$ solutions. The time for generating a 10,000 long

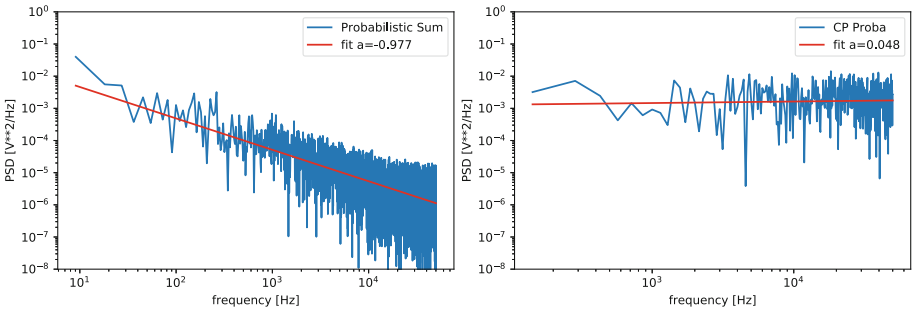


Fig. 2. PSD of probabilistic method in Left: standalone generation Right: in CP solver.

sequences is less than 2 s. For this sample the shift of each dice is given by [0, 0, 3, 4, 9, 23, 41, 13, 182, 198, 263, 1794, 949].

Figure 2 shows the PSD of signals produced using only the probabilistic method, both within a CP solver and as a standalone process. As can be seen as a standalone process the probabilistic method is successful at generating $1/f$ sequences. However, when formulated for use in a CP solver, this approach fails to generate $1/f$ noise. In fact, it appears to generate white noise, given the small slope. Upon closer inspection of our results, it seems the random search very

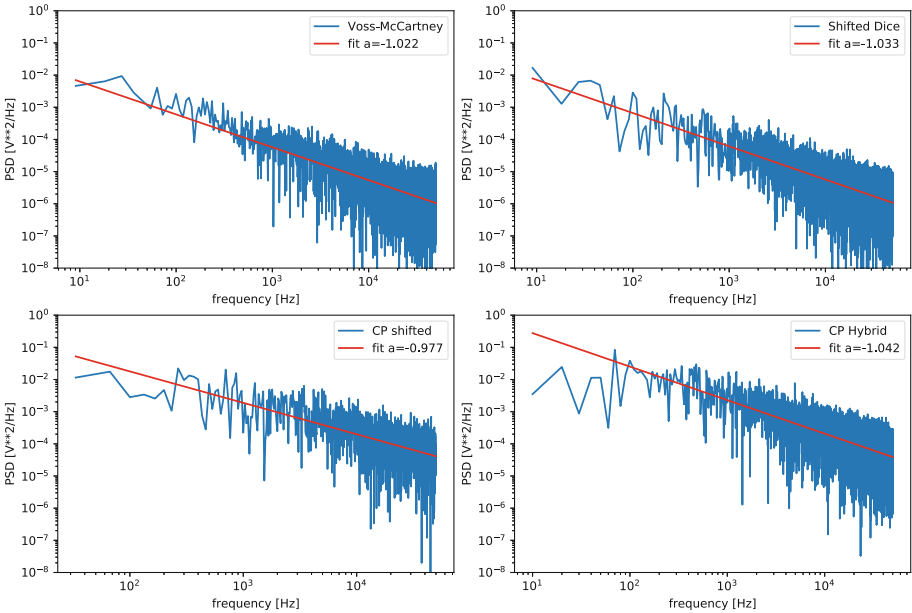


Fig. 3. Top left: PSD of Voss algorithm. Top right: PSD of shifted dice method outside of CP solver. Bottom left: PSD of shifted dice method using CP solver. Bottom Right: PSD of hybrid method using CP solver.

quickly modifies the dice values, and within a relatively small number of assignments reaches its maximum number of modifications as given per the probability constraint. Thus such a constraint alone is not enough for solving the $1/f$ problem. Additionally, because of the larger number of variables ($\#dice * \text{sequence length}$) the time required for generating is substantially larger, but remains less than 2 min.

We believe, given these results, that the hybrid model is the most promising for generating $1/f$ sequences. This method uses the shifted model with the additional Markov constraints, which enforce that the dice change values. We found that this model is as fast as just the shifted method and we believe that it ensures a certain robustness for generating $1/f$ sequences, as well as preventing outlier solutions. Specifically it removes the ability of the shifted version to produce non- $1/f$ solutions. Moreover, the Markov constraint on the dice is less restrictive than a GCC, enforcing to know in advance the values.

6 Conclusion

In this paper we extend the capacity of the Voss constraint definition and give two different propagators, as well as a hybrid of the two, for enforcing it. Specifically, we show how to generalize a Voss tree, to allow each shifted version. Additionally, we answer an open question from [9] by proposing an entirely probabilistic CP model for generating $1/f$ sequences. We experimentally show that this approach fails to generate $1/f$ sequences in a CP solver, as the solution density is very low. However, this method does work outside of CP solvers. Finally, we define a hybrid method combining the strengths of both propagators, and show its robustness for generating $1/f$ sequences.

References

1. Herriman, A., McCartney, J., Burk, P., Downey, A., Whittle, R., Kellet, P.: Generation of pink ($1/f$) noise. <http://www.firstpr.com.au/dsp/pink-noise/>
2. Arafailova, E., et al.: Global constraint catalog, vol. II, time-series constraints. arXiv preprint [arXiv:1609.08925](https://arxiv.org/abs/1609.08925) (2016)
3. Chemillier, M., Truchet, C.: Computation of words satisfying the rhythmic oddity property (after simha arom's works). *Inf. Process. Lett.* **86**(5), 255–261 (2003)
4. Gardner, M.: White and brown music, fractal curves and one-over- f fluctuations. *Sci. Am.* **238**(4), 16–32 (1978)
5. Hennig, H., et al.: The nature and perception of fluctuations in human musical rhythms. *PLoS ONE* **6**(10), e26457 (2011)
6. Kasdin, N.J.: Discrete simulation of colored noise and stochastic processes and $1/f$ /sup/spl alpha//power law noise generation. *Proc. IEEE* **83**(5), 802–827 (1995)
7. Keshner, M.S.: $1/f$ noise. *Proc. IEEE* **70**(3), 212–218 (1982)
8. Morin, M., Quimper, C.-G.: The markov transition constraint. In: Simonis, H. (ed.) CPAIOR 2014. LNCS, vol. 8451, pp. 405–421. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07046-9_29

9. Pachet, F., Roy, P., Papadopoulos, A., Sakellariou, J.: Generating 1/f noise sequences as constraint satisfaction: the voss constraint. In: IJCAI, pp. 2482–2488 (2015)
10. Papadopoulos, A., Pachet, F., Roy, P., Sakellariou, J.: Exact sampling for regular and markov constraints with belief propagation. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 341–350. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_24
11. Perez, G., Régin, J.-C.: Efficient operations on MDDs for building constraint programming models. In: IJCAI, pp. 374–380 (2015)
12. Perez, G., Régin, J.-C.: MDDs are efficient modeling tools: an application to some statistical constraints. In: Salvagnin, D., Lombardi, M. (eds.) CPAIOR 2017. LNCS, vol. 10335, pp. 30–40. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59776-8_3
13. Perez, G., Régin, J.-C.: MDDs: sampling and probability constraints. In: Beck, J.C. (ed.) CP 2017. LNCS, vol. 10416, pp. 226–242. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66158-2_15
14. Perlin, K.: An image synthesizer. ACM Siggraph Comput. Graph. **19**(3), 287–296 (1985)
15. Perron, L.: Or-tools. In: Workshop CP Solvers: Modeling, Applications, Integration, and Standardization (2013)
16. Pesant, G.: Achieving domain consistency and counting solutions for dispersion constraints. INFORMS J. Comput. **27**(4), 690–703 (2015)
17. Rossi, R., Prestwich, S., Tarim, S.A.: Statistical constraints. arXiv preprint [arXiv:1402.5161](https://arxiv.org/abs/1402.5161) (2014)
18. Roy, P., Pachet, F.: Enforcing meter in finite-length markov sequences. In: AAAI (2013)
19. Schaus, P., Deville, Y., Dupont, P., Régin, J.-C.: The deviation constraint. In: Van Hentenryck, P., Wolsey, L. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 260–274. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72397-4_19
20. Schaus, P., Régin, J.-C.: Bound-consistent spread constraint. EURO J. Comput. Optim. (2013)
21. Truchet, C., Assayag, G.: Constraint Programming in Music. ISTE-Wiley, Hoboken (2011)
22. Truchet, C., Codognet, P.: Musical constraint satisfaction problems solved with adaptive search. Soft. Comput. **8**(9), 633–640 (2004)
23. Voss, R.F., Clarke, J.: 1/f noise in music and speech. Nature **258**(5533), 317–318 (1975)
24. Voss, R.F., Clarke, J.: 1/f noise in music: Music from 1/f noise. J. Acous. Soc. Am. **63**(1), 258–263 (1978)

CP and Operations Research Track



Securely and Automatically Deploying Micro-services in an Hybrid Cloud Infrastructure

Waldemar Cruz, Fanghui Liu, and Laurent Michel^(✉)

Computer Science and Engineering Department, School of Engineering,
University of Connecticut, Storrs, CT 06269-4155, USA
{waldemar.cruz, fanghui.liu, laurent.michel}@uconn.edu

Abstract. Modern cloud-based services help deliver distributed software and aim to deliver a cost-effective solution while ensuring that application requirements are met. Deploying a Cloud-based implementation demands the resolution of a resource allocation problem to determine where and how software modules are deployed. For instance, one must decide, for each module, whether to deploy on a commercial elastic cloud provider or an in-house data-center as well as how to secure the communication channels that exist between services hosted with different providers. Each application is a collection of communicating micro-services that provides load-balancing and fault-tolerance to ensure quality of service requirements. There exists many choices as to what to deploy, where and which communication technologies to use. The purpose of this paper is to simultaneously solve the deployment of software services, the selection of suitable technologies for communication channels to meet the functional, performance and security requirements while minimizing economic costs.

1 Introduction

Modern applications are increasingly moving to Software as a Service (SaaS) over Platform as a Service (PaaS) implemented through a collection of communicating micro-services running on a medium to large distributed system. This allow system designer to handle scalability (for load-balancing) as well as fault-tolerance. Vendors typically adopt some form of virtualization technology to host those micro-services for ease of deployment and encapsulation of all the required software dependencies to build a micro-service. Cloud provider like Amazon (AWS) or Microsoft (Azure) deliver *elastic* infrastructures on which to run such software.

At the same time, micro-services must communicate with each other to implement their function and, when they are deployed via VMs hosted by cloud-service provider, their *communication channels* should be secured in a way that matches the requirements of the data owners and service developers. Small applications with a few micro-service instances can be readily implemented by engineers

and deployed with secure communication layers covering the required protocols for each channel. However, larger applications and deployments spanning over multiple data-centers and created by multiple developer teams are much more challenging to deploy *efficiently*. Developers routinely over-provision their requirements to “be safe” and this leads to considerable waste.

In addition, expecting software engineers to anticipate all security measures and protocols to secure such channels is excessive and largely unrealistic. It is better to ask developers and data owners to specify *requirements* on the security to adopt for each class of data, and therefore the services, and have optimization systems *automatically* select the right compromise for any pair of communicating services. This achieves an automatic adoption of security measures and relieves the developers from low level details relating to the adoption of the proper technology.

The purpose of this paper is, therefore, to *simultaneously address the deployment of services and the selection of suitable security stacks in each VM to protect each communication channel*. The problem can be defined by a set of servers, a set of VMs, and a set of services.

Observe, that this level of flexibility entails that the size of the virtual machine does depend on the set of security protocols adopted for the peering services and precludes the use of a direct formulation with knapsack-style constraints as the “weights” are no longer constant leaving the variant of bin packing as proposed in [3] unadapted to this setting.

The paper investigates IP, CP and hybrid formulations suitable for instances with up to 50 services. The CP approach leverages large neighborhood search to obtain good solutions in short run times but is unable to deliver optimality proofs. While the IP approach can sometimes produce an optimality proof, it usually takes a considerable amount of time and memory. The hybrid is an attempt at leveraging CP’s ability to deliver high incumbents quickly to support the IP formulation.

The remainder of the paper is organized as follows. Section 2 discusses relevant work in the context of data centers. Section 3 discusses the mathematical formulation in a technology agnostic style. Section 4 touches on the resolution and search procedures while Sect. 5 offers preliminary experimental results. Section 6 wraps up the paper.

2 Related Work

Energy management of data center is a prevalent topic in recent years. In [6] one can find an optimization model that minimizes the power consumption while optimally balancing the load between working and cooling server. The problem is represented as a non-linear energy utilization function and is solved using local search and a decomposition-based approach detailed in [4]. To consider optimally allocating workload while minimizing the energy cost, in a geographically distributed context, Wahbi et al. [18] use Distributed Constraint Optimization (DCOP) and introduce a DCOP algorithm AGAC-ng (nogood-based asynchronous generalized arc-consistency) to solve the model.

The bin Packing (BP) [16] problem and its variations have been extensively studied for allocating resources in data center. In [3], the authors consider a variant of BP where the use of bins are associated with linear costs, denoted as Bin Packing with Usage Cost (BPUC). BPUC is used to model the management of data center where all servers are viewed as bins and virtual machines are viewed as items, the energy consumption is represented as linear cost in BPUC. To minimize the linear cost, a linear relaxation of BPUC is used to calculate a lower bound for the objective, which is further strengthened with filtering algorithm for updating lower/upper bound of each bin. Another Variance of Bin Packing problem, referred to as Temporal Bin Packing (TBP) [5], was used to minimize the total allocated resources (CPU cycles) in a data center. Armant et al. [1] adopted semi-online bin packing (also know as batch bin packing) [9], and came up with a solution to semi-online task assignment where the data center performs a real-time allocation of users' tasks. Hermenier et al. [10] consider the allocation and reconfiguration problem which focuses on the bin packing of services for migration. In [14], the Dynamic Cache Distribution Problem is formulated as a bin packing problem and the solver finds an allocation that provides load balancing and a fault tolerant deployment. In [11] Hermenier et al., present a scheme for dynamically reallocation of virtual machines to hosts with the objective of minimizing migration costs. Kadioglu et al., introduced in [12] the Core Group Placement Problem (CGPP) which is formulated as a bin-packing problem that minimizes the total load for the deployment of heterogeneous services.

While this prior work is clearly relevant to data centers, it does not consider that the size of virtual machine is itself a variable that depends upon the deployment decision where co-location of VMs on the same host can spare the deployment of costly –from a memory standpoint– communication protocols. Perhaps, even more fundamentally, none of these consider the problem from the point of view of application architects with security concerns. What sets this work apart is its focus on simultaneously addressing the deployment problem *in light of the overhead induced by the security requirements on the deployment*.

3 Model

Given a set of compute nodes C with available memory CM and available bandwidth CB , allocate a set of service instances to compute nodes minimize the total cost of implementation. Namely, deploy each service instance in a VM by itself and host the VMs on servers such that the connectivity and demand requirements are met. The security requirements for each micro-service (VM) is met and the security requirements on the connecting links between services are met.

Consider a cloud based video service as shown in Fig. 1, each service type in the network diagram performs a function within the system. The inter-connections between the service types ensures that the functional requirements are met. For instance, the web front-end (Service Type 0) communicates with the video content server (Service Type 2) to deliver video to a web client. Each service type exist within the confines of a security zones based upon the sensitivity

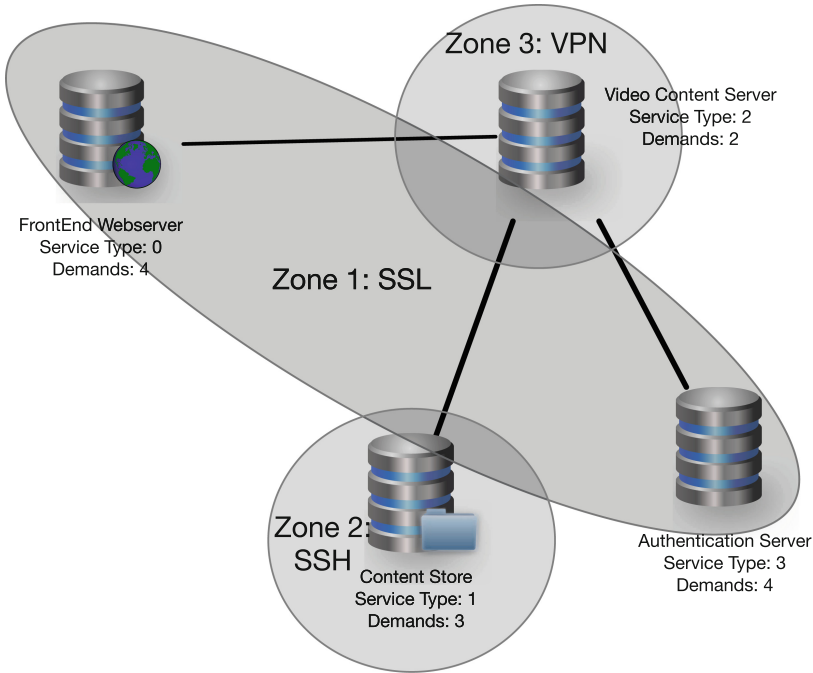


Fig. 1. A network diagram for a video content cloud service.

of the data sources it manipulates. For instance, the video content server exist within zone 3. Accessing the service from the same zone can be done at minimal cost. However, accessing it from a less secure zone mandates the adoption of a VPN to implement the communication channel between the two endpoints. Observe that the content store exist within zone 2, while the web front-end and the authentication server exist within zone 1. To cope with system load, multiple *instances* of each service type must be deployed. For instance, four distinct instances must be deployed for the web front-end. Note that the demands for the service types do not have to be equal. Indeed, 2 instances of the video content server are sufficient to meet the needs of the four authentication server.

The purpose of this paper is to *deploy* the instances of those service types on a cloud platform, meeting demands, resource capacities and security requirements for the communication channels connecting those instances.

Figure 2 demonstrates an example deployment of the cloud based application shown in Fig. 1. Compute nodes are represented as the square boxes and the service instances are represented as the rounded boxes. Each service type (e.g., $T_0, T_1, T_2,$ and T_3) with respective demands (4, 3, 2, 4) is associated with a set of instance identifiers. Namely,

- $\{I_1, I_2, I_3, I_4\}$ is the set of instances for T_0 ,
- $\{I_5, I_6, I_7\}$ is the set of instances for T_1 ,

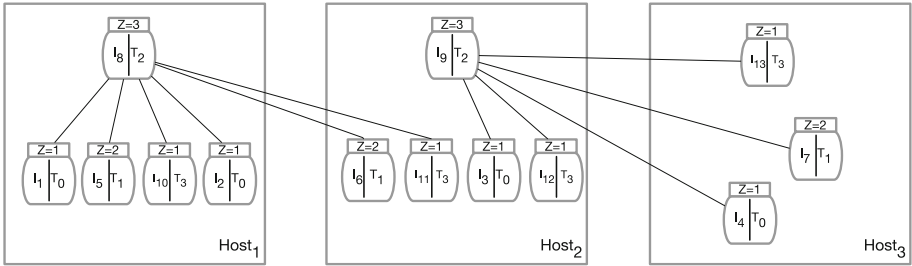


Fig. 2. Deployment of service instances to facilitate the cloud service.

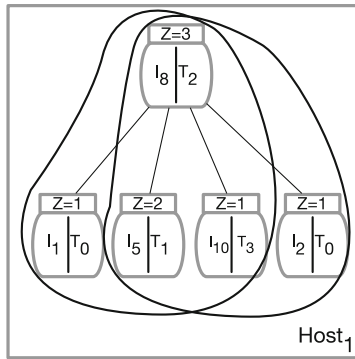


Fig. 3. View of compute node hosting two application instances.

$\{I_8, I_9\}$ is the set of instances for T_2 and $\{I_{10}, I_{11}, I_{12}, I_{13}\}$ is the set for T_3 .

In each box, Z represents the security zone for that service. The lines represent the communication channels between each pair of instances. The placement of the service instances consumes both memory and bandwidth resources on the hosting machine. Each service instance must adhere to its security and connectivity requirements. Service instances can be shared between multiple applications allowing to spread the demand and ensure sensible load balancing. Figure 3 shows two set of instances $\{I_1, I_5, I_8, I_{10}\}$ and $\{I_2, I_5, I_8, I_{10}\}$ that, together, form two applications whose boundaries are shown with the thick black outline. Clearly, Fig. 3 also shows that the services contributing to these two applications are hosted on the same machine. This does not have to be true. Services could be spread across several machine and require networked communication channels.

Figure 4 illustrates three service instances, namely I_9, I_3, I_4 deployed across $Host_2$ and $Host_3$. Within $Host_2$, two virtual machine are used to host services I_3 and I_9 . The boundaries of those virtual machines are the rounded boxes. Hosting these two services in co-located virtual machines allows for the creation of a communication channel with no additional security measures such as encryption. This is expected since that communication channel itself is virtual and

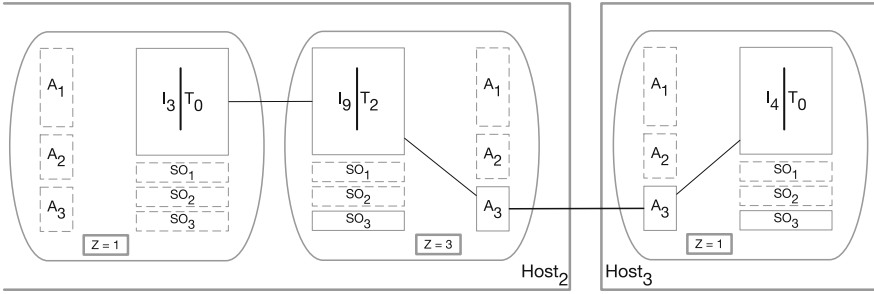


Fig. 4. Implementation of security adapters for communication across compute nodes.

only exists within $Host_2$. In contrast, the deployment of I_9 and I_4 in virtual machines on different hosts requires the adoption of a suitable security layer for their communication channel. Specifically, I_9 being of type T_2 requires a zone 3 (i.e., VPN) communication channel while I_4 (of type T_0) only needs SSL for its channel. The most stringent requirement is therefore adopted and this mandates the deployment of copies of the A_3 adapter (VPN) in the two virtual machines. Observe how different hosting decisions may lead to the usage or sharing of several adapters, each of which carries a cost. In this example, the adoption of A_3 within those virtual machines incurs a memory overhead illustrated by the non-dashed boxes SO_3 .

3.1 Problem Definition

Each problem is define with the following parameters:

- $T \subset \mathbb{N}$: a set of service types defines the unique service components required for the application.
- $I \subset \mathbb{N}$: a set of service instances, where is defined as the set of service instances belonging to type t .

$$I = \cup_{t \in T} I_t$$

I_t is defined as the set of service instances belonging to type t . Formally,

$$\forall t \in T, I_t = \{(\sum_{j=1..t-1} D_j) + 1, \dots, \sum_{j=1..t} D_j\}$$

- $Z \subset \mathbb{N}$: a set of security zones defines the security requirements for service types.
- $C \subset \mathbb{N}$: a set of compute nodes defines the compute nodes available for service deployment.
- $FM \subset \mathbb{N}$: a set of fixed memory usage for each service type.
- $FB \subset \mathbb{N}$: a set of fixed bandwidth usage for each service type.
- $D \in \mathbb{N}$: a demand for each service type.

- $\text{conn} \in \mathbb{N}^{\mathbb{N} \times \mathbb{N}}$: a adjacency matrix describes the network architecture between service types. conn_{t_1, t_2} is the connectivity between service type t_1 and t_2 , where $t_1, t_2 \in T$.
- $\text{SM} \subset \mathbb{N}$: a set of scaling factors for service memory usage for each service type.
- $\text{SB} \subset \mathbb{N}$: a set of scaling factors for service bandwidth usage for each service type.
- $\text{AB} \subset \mathbb{N}$: a set of bandwidth cost for each security adapter in each security zone.
- $\text{AM} \subset \mathbb{N}$: a set of memory costs for each security adapter in each security zone.
- $\text{VO} \in \mathbb{N}$: overhead associated with running a VM.
- $\mathbf{w}_m \in \mathbb{N}$: the cost of memory per MB.
- $\mathbf{w}_b \in \mathbb{N}$: the cost of bandwidth per Mb/s.
- $\mathbf{t}(i)$: the service type associated with service instance i .

3.2 Variables

- $h_i \in C$: The compute node that service i is deployed on.
- H : The set containing the placements of the service instances.

$$H = \{h_i | i \in C\}$$

- $O_i \in \mathbb{N}$: is the overhead associated to the deployment of adapters for service i .
- $SO_i \in \mathbb{N}$: is the overhead of implementing security options for service i .
- $UB_i \in \mathbb{N}$: the bandwidth usage for machine i .
- $UM_i \in \mathbb{N}$: the memory usage for machine i .
- $z_{ij} \in Z$: security technology for securing channel ij between services i and j .
- $A_{ij} \in \mathbb{B}$: implementation of security adapter responsible for securing a communication channel for service instance i and service zone j .
- $links$: a matrix of active link connections between two service instances. $links_{ij}$ define the number of links between service instance i and j .
- CC denotes the sum connections of service instance across different machines. CC_{ijk} defines the all the external connections out of machine $i \in C$, connecting to service $j \in T$, and protected with security zone $k \in Z$.
- V : the set containing all the variables in the model.

3.3 Constraints

Connection Constraints. The problem definition includes how the service types are connected within the network architecture. The demand requirements are defined by D . Each instance represents a single unit of demand for a particular service type. Each service type with a demand of D_t requires D_t service instances for service type t . To meet the demand of a service type, it is required that the

sum of the connections into all the service instances of a service type is equal or exceeds the required demand on that service.

$$\sum_{j \in I_{t_2}} links_{ij} \leq D_{t_2}, \forall i \in I_{t_1}, t_1, t_2 \in T \text{ st. } conn_{t_1, t_2} > 0$$

$$\sum_{i \in I_{t_1}} links_{ij} \leq D_{t_1}, \forall j \in I_{t_2}, t_1, t_2 \in T \text{ st. } conn_{t_1, t_2} > 0$$

Ensure that each instance of type t_1 is connected to at least one service instance:

$$\sum_{j \in I_{t_2}} links_{ij} \geq conn_{t_1, t_2}, \forall i \in I_{t_1}, t_1, t_2 \in T \text{ st. } conn_{t_1, t_2} > 0$$

$$\sum_{i \in I_{t_1}} links_{ij} \geq conn_{t_1, t_2}, \forall j \in I_{t_2}, t_1, t_2 \in T \text{ st. } conn_{t_1, t_2} > 0$$

To ensure that the demand for each of the service instances are uniformly distributed, a set of load balancing constraints defined:

$$(D_{t_2} \geq D_{t_1}) \rightarrow \sum_{j \in I_{t_2}} links_{ij} \geq \frac{D_{t_2}}{D_{t_1}}, \forall i \in I_{t_1}, t_1, t_2 \in T \text{ st. } conn_{t_1, t_2} > 0$$

$$(D_{t_2} \geq D_{t_1}) \rightarrow \sum_{i \in I_{t_1}} links_{ij} \leq \frac{D_{t_2}}{D_{t_1}} + (D_{t_2} \bmod D_{t_1}), \forall j \in I_{t_2}, t_1, t_2 \in T \text{ st. } conn_{t_1, t_2} > 0$$

The connection demand of each instance must be connected uniformly among the connected instances. This applies load balancing to the network and ensure that the demand is met from multiple service instances.

Security Constraints. Each service is required to implement a security technology to ensure security requirements for a particular aspect of the application. A service type is assigned to a security zone that enforces the use of the appropriate security technology. Security zones are a hierarchical structure, a service within a security zone must communicate with security technology that is stronger or equivalent to the security requirements of the zone. Two service instances that belong to different services must communicate with the security technology of the highest security protocol. Suppose there are two services instances i and j , i belongs to service zone 1 and j belongs to service zone 2, therefore the communication channel should be secured with the security requirements of security zone 2. The maximum security policy for two communicating service is given by:

$$\forall i, j \in I, i \neq j : z_{ij} = \max(z_i, z_j) \cdot (h_i \neq h_j) \cdot (link_{ij} > 0)$$

This ensures that the communication channel is secured with the highest security protocol. A service instance is required to implement a security adapter A , only

if there exists a link to another service instance that does not belong to the same machine.

$$\forall i \in I, k \in Z : A_{ik} = (\sum_{j \in I} (z_{ij} = k) > 0)$$

If two services are located on the same machine, then the security adapter does not need to be implemented as the communication channel does not leave the hosting machine. This ensures that connections between hosting machines are secured with the required security technology for each service type.

Memory Consumption Constraints. Each service type implements a suite of software that introduces a fixed amount of memory usage for its instance. A service instance that is connected via an external connection to a service instance with a higher security zone must implement the most stringent security adapter to meet its security requirements for the communication channel. Deployment of the adapter increases the memory usage of the service by a scaling factor and imposes a fixed memory cost.

The overhead is therefore:

$$\forall i \in I : O_i = \sum_{k \in Z} A_{ik} \cdot AM_k$$

Consider that **SM** is an array of percentages that model the relative overhead. The overhead caused by the scaling of the memory consumption of the service:

$$\forall i \in I : SO_i = FM_i \cdot (1 + \sum_{k \in Z} (A_{ik} \cdot SM_k))$$

Finally, The total sum of memory usage from each host is bounded above by the host memory size. Recall that each service instance is contained on a VM by itself. Therefore the total overhead for a service in a VM is O_i and SO_i depicted by:

$$\forall k \in C : (\sum_{i \in I \text{ st. } h_i=k} SO_i + O_i) + |I| \cdot v0 \leq UM_k$$

The memory constraint ensures that the total memory usages on all service deployed on a particular machine do not exceed the memory capacity of the hosting machine.

Bandwidth Constraints. When two connected services are deployed on distinct hosts, their network traffic contributes to the bandwidth utilization of the network link. Securing the communication channel increases the overhead on the bandwidth usage. The total bandwidth for the host is the sum of the bandwidth usage of all the external connections. The external connections from out of each machine can be counted by summing all the connections from services instances communicating to service instance deployed on different machines. Each external connection is secured with a security technology defined by each security

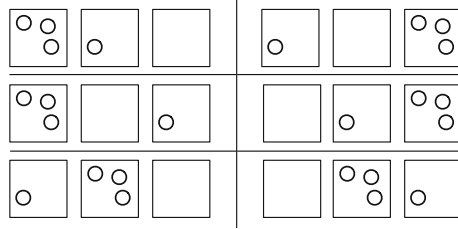


Fig. 5. A deployment configuration that presents 6 symmetric solutions.

zone. **CC** defines the number of external connections for each service type and security zone for a particular machine.

$$CC_{xyz} = \sum_{i \in Sst.h_i=x} \sum_{j \in Ss.t.i \neq j} (z_{ij} = z) \cdot (\mathfrak{t}(j) = y) \cdot links_{ij} \quad \forall x \in C, y \in T, z \in Z$$

The total bandwidth is encoded with the following constraint:

$$\forall i \in C : UB_i \geq \sum_{j \in T} \sum_{k \in Z} ((CC_{ijk} \cdot FB_j \cdot SB_k) + AB_k \cdot (CC_{ijk} > 0))$$

Symmetry Breaking Constraints. Suppose a service deployment scenario where one or more of the hosts are unused. Assume that hosts are strictly equivalent (same memory or bandwidth). As shown in Fig. 5, each host is denoted as the square box, and service instances are represented as circles. The solution presented in the Figure shows three hosts where one of the compute nodes contains no service instance. The number of deployed service instances with respect to the hosts are {3, 1, 0}; this configuration presents 6 symmetric solutions. To reduce the number of symmetric solutions relative to the number of deployed instances, hosts can be arranged such that if host h_i has no deployed instances, then host h_{i+1} should also have no deployed instances. Formally,

$$\forall i \in C : \sum_{j \in I} (h_j = i) = 0 \rightarrow \sum_{k \in I} (h_k = i + 1) = 0$$

Objective Function. The goal is to assign service instances to compute nodes such that the cost of deployment is minimized. The cost is determined by the total memory usage and bandwidth usage required to implement the application across C hosts. Note how the memory usage UM_i and the bandwidth usage UB_i are weighted with w_m and w_b respectively

$$min : \sum_{i \in C} (w_m \cdot UM_i + w_b \cdot UB_i)$$

4 Search Strategy

Finding a satisfiable solution requires finding a deployment configuration for placing service instances on compute nodes, and setting the links between each of the connected service instances. The paper explores three search methods/strategies for attempting to find the optimal configuration, Integer Programming, CP with LNS and an hybrid.

4.1 MIP

Integer programming could be tempting given the packing aspects of the problem. Naturally, Integer Programming expects a linear model and it is thus necessary to linearize components of the model that are not linear. To convert the model, each non-linear constraint must be re-formulated to a linear encoding. Recall, for example, the constraint responsible for channel security:

$$\forall i, j \in \mathbb{I}, i \neq j : z_{ij} = \max(z_i, z_j) \cdot (h_i \neq h_j) \cdot (\text{link}_{ij} > 0)$$

Since z_i and z_j are constants, $h_i \neq h_j$ is a boolean and $\text{link}_{ij} > 0$ is a boolean, it has the form:

$$\mathbb{N} = \mathbf{c} \cdot \mathbb{B} \cdot \mathbb{B}$$

and it must be linearized with:

$$\begin{aligned} b_3 &\leq b_1 \\ b_3 &\leq b_2 \\ b_3 &\leq b_1 + b_2 - 1 \end{aligned}$$

In this case, $b_3 \in \mathbb{B}$ is the output and $b_1 \in \mathbb{B}$ and $b_2 \in \mathbb{B}$ are the inputs and the variable product in the original can be replaced by b_3 . Similar rewrites are completely automated by the linearization module of Objective-CP [7].

4.2 CP with LNS

Large Neighborhood Search (LNS) [15] is a local search strategy that aims to find high quality local minima quickly. The premise behind LNS is to re-optimize on the current best solution by “locking” a subset of variables with the values obtained in the incumbent solution. The set of variables that are not “locked” become the branching variables for the next attempt. The search relies on two phases. The first phase focuses on labeling the host variables to deploy the services on the hardware. It uses a simple dynamic degree heuristic (weighted degrees is effective as well) [2]. The second phase considers the remaining variables (those in $X \setminus \{h_i | i \in \mathbb{I}\}$) and adopts the same variable selection heuristic. In both phases, the value selection heuristic is a simple minimum domain value. In other words, the search prioritizes branching over variables with the smallest domain and that are connected to the most unbounded variables, yet it favors

deployment decisions over connectivity. In isolation, the CP search strategy represents a complete search tree and therefore can obtain the optimal solution (given enough time).

The search attempts to find high quality solutions by first locating an initial solution and then iteratively relaxing a subset of the variables and re-optimizing it *locally*. This is best achieved by *freezing* the complement of the relaxed set to their value in the incumbent solution. Clearly, LNS is incomplete and cannot deliver an optimality proof. Yet it shows good behavior on the synthetic instances considered so far.

4.3 Hybrid

As the problem scales with the number of service instances and the number of available machines, the size of the linearized model will increase. This will result in the expansion of the number of variables and constraints required to encode the model, this will make it difficult for the MIP to scale up. Knowing that LNS has the capability of finding improving solutions quickly, It becomes advantageous to combine the efforts of the LNS and the MIP solver to aid in finding the optimal solution. Using a combinator [7], the LNS and MIP model can be combined to share information about incumbent solutions. This will allow the two solvers to update their bounds on the objective. In this context, LNS provides the MIP solver with high quality solution sooner allowing the solver to update its bounds and prune away any subtree that will not yield a solution. LNS solver will not be able to close the problem, and will rely on the MIP solver to find and prove the optimal solution.

5 Results

While this work is motivated by applications from Comcast, Inc. we are not at liberty to share actual instances. To investigate the approaches proposed here, several *synthetic instances* were created. They correspond to small to medium applications composed of a handful of services. Demands for the services are driven by the need for fault-tolerance and load balancing. The ultimate objective is consider instances with hundreds of services and use the tool both in an operational setting, but also for capacity planning and deciding whether to build up internal data centers or rely on elastic cloud providers.

The implementations all rely on Objective-CP [13, 17] as they rely on Integer Programming, Constraint Programming and an hybrid. The combinator abstractions [7] were particularly effective to easily construct LNS and the parallel hybrid in a style reminiscent of [8].

The remainder of this section contrast results from the *IP*, *LNS(CP)* and the hybrid on 8 synthetic instances of various sizes. While still modest in size, these prove challenging for all solvers and show that further efforts are needed. All times are reported in seconds. All benchmarks were executed on a Xeon(R) CPU E5-2640 v4 @ 2.40 GHz on a single core running Linux kernel 4.4.0-119-generic. The MIP solver is Gurobi 7.5.2.

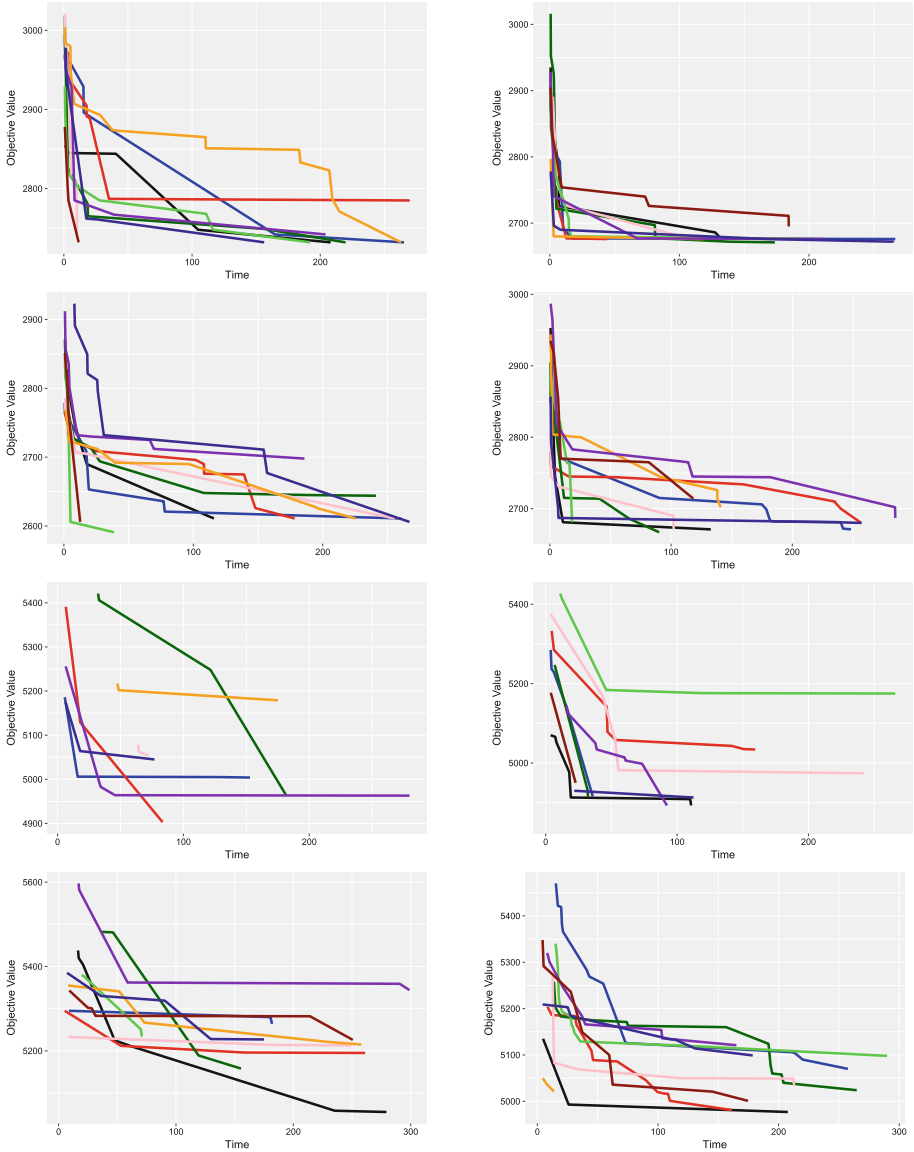


Fig. 6. LNS Performance profile (quality vs. time for 10 runs of 8 instances).

LNS(CP). The performance profile for LNS is probably the most eloquent rendering of the search’s ability to deliver. Figure 6 shows 8 plots (left-right, top-bottom) for the 8 synthetic instances. Each box shows the trajectory followed by all 10 runs. For the smaller instances, LNS tends to deliver an *excellent* (if not the optimal) solution in under 3 min. The behavior is slightly less uniform for the bigger instances, yet the ability to get good solutions is retained.

Table 1. Performance for MIP, LNS and Hybrid

Benchmark	MIP	LNS		Hybrid	
	T	μ_T	σ_T	μ_T	σ_T
1	4,178.00	180.01	96.07	409.01	92.8
2	249.00	171.54	71.21	349.40	143.7
3	343.00	178.08	92.87	264.52	217.9
4	290.00	164.93	90.05	349.57	90.8
5	4,251.00	118.66	84.98	358.69	183.4
6	4,629.00	108.55	90.25	269.98	213.5
7	344.00	218.16	70.22	335.47	141.2
8	5,326.00	192.58	77.30	300.18	157.1

Table 2. Quality for MIP, LNS and Hybrid

Benchmark	MIP	LNS		Hybrid	
	Q	μ_Q	σ_Q	μ_Q	σ_Q
1	2,732.00	2,739.20	16.28	2,752.22	25.8
2	2,642.00	2,676.90	6.69	2,674.71	2.6
3	2,591.00	2,620.00	30.36	2,620.50	32.5
4	2,667.00	2,682.40	15.07	2,684.38	15.8
5	4,903.00	5,002.44	81.52	4,985.50	110.8
6	4,925.00	4,951.40	93.27	4,978.00	92.3
7	5,317.00	5,213.20	73.73	5,229.25	96.9
8	5,142.00	5,042.60	51.49	5,096.30	48.8

Comparative. Figure 7 offers a comparison between the IP , $LNS(CP)$ and the hybrid solvers. The two plots are histograms with one group of 3 bars for each benchmark. Each triplet reports on $LNS(CP)$ (red), the MIP (blue) and the hybrid (green). The left plot is an *absolute* performance comparison that matches the results in Table 2. The right plot is also an absolute comparison of the running times and matches the content of Table 1. What is abundantly clear from the graphs is that, if quality is the sole objective, the three methods are pretty close. In fact, they are within $\pm 4\%$ of each other with $LNS(CP)$ sometimes beating the IP and vice-versa. This percentage is obtained by doing

$$\frac{X - MIP}{\min(X, MIP)}$$

where X is either $LNS(CP)$ or the Hybrid. The remarkable difference is, as expected, the runtime. It is clear that hardness for the IP does not depend on the instance size with some of the easy instances taking a long time to produce a high-quality solution. $LNS(CP)$ does not appear to suffer from this. Naturally,

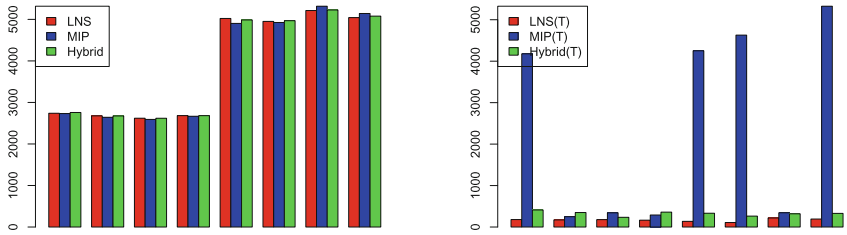


Fig. 7. Comparison between *IP*, *LNS(CP)* and *Hybrid* both in absolute solution quality and runtime performance. (Color figure online)

the hybrid brings the best of both worlds and delivers a far more *robust* behavior consistently delivering good solutions quickly. In most cases, however, the *IP* ran for 5400 s terminating prematurely (by timeout) without a proof of optimality.

6 Conclusion

This paper investigated three optimization approaches to solve the simultaneous deployment of micro-services in a cloud-based infrastructure and the automatic selection of security layers for their communication channels. It offers a technology-neutral model, search procedures and an hybrid. The synthetic instances used for the evaluation show that the problem can be quite challenging for state of the art solvers. Yet, empirical results demonstrate that LNS is capable of delivering high-quality solutions in short execution times, a capability that is critical in an operational setting. Future work will be needed to adapt to larger instances and increasingly comprehensive resource consumption models.

Acknowledgment. This work was supported under the award SOW BL 7891 and project CSI Selected Projects 2017: *Securing Virtualization Configuration and Managing the Attack Surfaces* funded by Comcast Corporation. Special thanks to Jim Fahrny and Vaibhav Garg from Comcast.

References

1. Armant, V., Cauwer, M.D., Brown, K.N., O’Sullivan, B.: Semi-online task assignment policies for workload consolidation in cloud computing systems. *Future Gener. Comput. Syst.* **82**, 89–103 (2018). <http://www.sciencedirect.com/science/article/pii/S0167739X17319143>
2. Boussemart, F., Hemery, F., Lecoutre, C.: Revision ordering heuristics for the constraint satisfaction problem. In: *First International Workshop: Constraint Propagation and Implementation* (2004). <http://www.cril.univ-artois.fr/~lecoutre/research/publications/2004/CPW2004.ps>
3. Cambazard, H., Mehta, D., O’Sullivan, B., Simonis, H.: Bin packing with linear usage costs. *CoRR* abs/1509.06712, <http://arxiv.org/abs/1509.06712> (2015)

4. Castiñeiras, I., Chisca, D.S., Mehta, D., O'Sullivan, B.: Trichotomic search for thermal-aware data centre workload optimisation. In: 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC), pp. 528–533, December 2015
5. Cauwer, M.D., Mehta, D., O'Sullivan, B.: The temporal bin packing problem: an application to workload management in data centres. In: 2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI), pp. 157–164, November 2016
6. Chisca, D.S., Castineiras, I., Mehta, D., OSullivan, B.: On energy- and cooling-aware data centre workload management. In: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 1111–1114, May 2015
7. Fontaine, D., Michel, L., Van Hentenryck, P.: Model combinatorics for hybrid optimization. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 299–314. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_25
8. Fontaine, D., Michel, L., Van Hentenryck, P.: Parallel composition of scheduling solvers. In: Quimper, C.-G. (ed.) CPAIOR 2016. LNCS, vol. 9676, pp. 159–169. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33954-2_12
9. Gutin, G., Jensen, T., Yeo, A.: Batched bin packing. *Discrete Optim.* **2**(1), 71–82 (2005). <http://www.sciencedirect.com/science/article/pii/S1572528605000058>
10. Hermenier, F., Demasse, S., Lorca, X.: Bin repacking scheduling in virtualized datacenters. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 27–41. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23786-7_5
11. Hermenier, F., Lawall, J., Muller, G.: BtrPlace: a flexible consolidation manager for highly available applications. *IEEE Trans. Dependable Sec. Comput.* **10**(5), 273–286 (2013)
12. Kadioglu, S., Colena, M., Sebbah, S.: Heterogeneous resource allocation in cloud management. In: 2016 IEEE 15th International Symposium on Network Computing and Applications (NCA), pp. 35–38. IEEE (2016)
13. Michel, L., Van Hentenryck, P.: A microkernel architecture for constraint programming. *Constraints* **22**(2), 107–151 (2017). <https://doi.org/10.1007/s10601-016-9242-1>
14. Sebbah, S., Bagley, C., Colena, M., Kadioglu, S.: Availability optimization in cloud-based in-memory data grids. In: Rueher, M. (ed.) CP 2016. LNCS, vol. 9892, pp. 666–679. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44953-1_42
15. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M., Puget, J.-F. (eds.) CP 1998. LNCS, vol. 1520, pp. 417–431. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49481-2_30
16. Srikantaiah, S., Kansal, A., Zhao, F.: Energy aware consolidation for cloud computing. In: Proceedings of the 2008 Conference on Power Aware Computing and Systems, HotPower 2008, p. 10. USENIX Association, Berkeley, CA, USA (2008). <http://dl.acm.org/citation.cfm?id=1855610.1855620>
17. Van Hentenryck, P., Michel, L.: The objective-CP optimization system. In: Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming, September 2013
18. Wahbi, M., Grimes, D., Mehta, D., Brown, K.N., O'Sullivan, B.: A distributed optimization method for the geographically distributed data centres problem. In: Salvagnin, D., Lombardi, M. (eds.) CPAIOR 2017. LNCS, vol. 10335, pp. 147–166. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59776-8_12



Improving Energetic Propagations for Cumulative Scheduling

Alexander Tesch^(✉)

Zuse Institute Berlin (ZIB), Takustraße 7, 14195 Berlin, Germany
tesch@zib.de

Abstract. We consider the Cumulative Scheduling Problem (CuSP) in which a set of n jobs must be scheduled according to release dates, due dates and cumulative resource constraints. In constraint programming, the CuSP is modeled as the cumulative constraint. Among the most common propagation algorithms for the CuSP there is energetic reasoning [1] with a complexity of $O(n^3)$ and edge-finding [21] with $O(kn \log n)$ where $k \leq n$ is the number of different resource demands. We consider the complete versions of the propagators that perform all deductions in one call of the algorithm. In this paper, we introduce the energetic edge-finding rule that is a generalization of both energetic reasoning and edge-finding. Our main result is a complete energetic edge-finding algorithm with a complexity of $O(n^2 \log n)$ which improves upon the complexity of energetic reasoning. Moreover, we show that a relaxation of energetic edge-finding with a complexity of $O(n^2)$ subsumes edge-finding while performing stronger propagations from energetic reasoning. A further result shows that energetic edge-finding reaches its fixpoint in strongly polynomial time. Our main insight is that energetic schedules can be interpreted as a single machine scheduling problem from which we deduce a monotonicity property that is exploited in the algorithms. Hence, our algorithms improve upon the strength and the complexity of energetic reasoning and edge-finding whose complexity status seemed widely untouchable for the last decades.

1 Introduction

In the CuSP, we are given a set of non-preemptive jobs $i \in \{1, \dots, n\}$ with processing times $p_i > 0$, resource demands $c_i > 0$, release dates r_i , due dates d_i and a resource capacity C . We assume that $0 \leq r_i \leq d_i - p_i$ for all jobs $i = 1, \dots, n$ without loss of generality. In the CuSP, we compute start times S_i for every job $i = 1, \dots, n$ such that every job is scheduled within its time interval and the resource capacity is never exceeded. More formally, it can be stated as

$$\sum_{\substack{i=1, \dots, n: \\ S_i \leq t < S_i + p_i}} c_i \leq C \quad \forall t \in [0, T] \quad (1)$$

$$r_i \leq S_i \leq d_i - p_i \quad \forall i = 1, \dots, n \quad (2)$$

where T is an upper bound on the time horizon. In general, the CuSP can be seen as a decision variant of the scheduling problem $P|r_j, size_j|L_{max}$ in which a set of n non-preemptive multiprocessor jobs is scheduled onto C machines where every job j allocates $c_j = size_j$ machines at the same time. The objective is to minimize the maximum lateness $L_{max} = \max_{j=1}^n \max\{0, S_j + p_j - d_j\}$ where $L_{max} = 0$ if and only if the CuSP is feasible. The CuSP is NP-complete since already the scheduling problem $1|r_j|L_{max}$ is strongly NP-hard [11].

In practice, the CuSP often occurs as a dedicated subproblem in more complex scheduling or optimization problems [9, 17]. In order to reduce the size of the search tree, the idea is to derive stronger bounds on the release and due dates of the jobs without violating the feasibility of the problem in general. For the CuSP, many propagation algorithms have been suggested in the literature where one can observe a general trade-off between their propagation power and running time.

1.1 Previous Work

Many different propagators exist for the CuSP in the literature as well as relaxations and hybrids of them. In this paper, we focus on *complete* propagation algorithms that perform *all* reductions according to their respective rule. The *energetic reasoning* propagation rule has been first introduced in [6] where the methods were refined in [1] to give a complete $O(n^3)$ algorithm. Since energetic reasoning counts to the most powerful propagation rules for the CuSP many efforts have been made to improve the general $O(n^3)$ complexity, for example by reducing the number of considered intervals [2, 5]. In [3] an algorithm is presented that detects at least one energetic reasoning propagation in $O(n^2 \log n)$ that was improved in [20] to an algorithm that detects at least one possible energetic reasoning propagation for every job in $O(n^2 \log n)$. Both algorithms, however, remain incomplete and do not compute all the maximum propagations in one run. However, they converge to the same fixpoint of energetic reasoning.

Another common propagation rule is *edge-finding* where a first complete algorithm was introduced in [14] with complexity $O(kn^2)$ and also a stronger variant called *extended edge-finding* with the same complexity. This complexity was improved in [21] to $O(kn \log n)$ and in [16] they use a modification of similar techniques to integrate propagations from extended edge-finding and timetabling also in $O(kn \log n)$. In general, energetic reasoning yields stronger propagations than (extended) edge-finding. However, its high complexity of $O(n^3)$ prevents it from being used in practice where edge-finding or hybrids such as *time-table edge-finding* [19, 22] are preferred. Thus, a major open question is whether the $O(n^3)$ complexity of energetic reasoning can be improved to lower the discrepancy between the different propagation schemes.

Further propagation algorithms for the CuSP are *timetabling* [7, 12] and *not-first/not-last* [18] where the latter is incomparable to energetic reasoning.

1.2 Results

In this paper, we introduce the novel *energetic edge-finding* propagation rule that generalizes and combines ideas from energetic reasoning and edge-finding. We first show that energetic edge-finding reaches its fixpoint in strongly polynomial time. But our main contribution is a complete energetic edge-finding algorithm that runs in $O(n^2 \log n)$ and therefore improves upon the previous $O(n^3)$ complexity of energetic reasoning. Moreover, by excluding some difficult time intervals, we obtain a relaxed version of energetic edge-finding that runs in $O(n^2)$ and subsumes the edge-finding rule while simultaneously performing stronger propagations from energetic reasoning. This also improves the general $O(kn \log n)$ complexity of edge-finding [21]. A major component in our algorithms is the interpretation of energetic schedules as a single machine scheduling problem with release dates [8]. Energy profiles of single machine problems yield a monotonicity property that we highly exploit in our algorithms. We implemented and tested the algorithms on instances of the well-established PSPLIB [13]. We believe that the given approaches yields new insights for the development of more efficient propagation algorithms for the CuSP.

2 Energetic Edge-Finding

In this section we formally introduce the energetic edge-finding rule. It can be seen as a generalization and combination of both energetic reasoning and edge-finding. It takes advantage of the propagation strength of energetic reasoning and the stable fixpoint behavior of edge-finding by integrating additional propagations from subintervals. Our version of energetic edge-finding is not to be mixed-up with the one given in [10] who consider a similar but incomplete rule.

We begin with some notation. Let the minimum left-right-shift duration of a job i in a time interval $[t_1, t_2]$ be defined as

$$p_i(t_1, t_2) = \max\{0, \min\{p_i, t_2 - t_1, r_i + p_i - t_1, t_2 - d_i + p_i\}\} \quad (3)$$

where only the left-shift duration of job i in $[t_1, t_2]$ is given by

$$p_i^l(t_1, t_2) = \max\{0, \min\{t_2 - t_1, p_i, r_i + p_i - t_1, t_2 - r_i\}\}. \quad (4)$$

Moreover, let $e(t_1, t_2) = \sum_{i=1}^n c_i \cdot p_i(t_1, t_2)$ denote the energy in the interval $[t_1, t_2]$ where the *energy overload* in $[t_1, t_2]$ is given by

$$\omega(t_1, t_2) = e(t_1, t_2) - C \cdot (t_2 - t_1) \quad (5)$$

that equals the slack between the minimum consumed energy and available energy in $[t_1, t_2]$. If there exists an interval $[t_1, t_2]$ with $\omega(t_1, t_2) > 0$ then the CuSP is already infeasible. Thus we assume that $\omega(t_1, t_2) \leq 0$ for all intervals $[t_1, t_2]$. Analogously, for a job i and time interval $[t_1, t_2]$ let

$$\omega_i^l(t_1, t_2) = \omega(t_1, t_2) + c_i \cdot (p_i^l(t_1, t_2) - p_i(t_1, t_2)) \quad (6)$$

be the energy overload under the condition that job i is left-shifted. If there exists a time interval $[t_1, t_2]$ with $\omega_i^l(t_1, t_2) > 0$ then the release date of job i is invalid and can be increased to $r_i = t_2 - p_i(t_1, t_2) + \frac{\omega(t_1, t_2)}{c_i}$ in order to pull the exceeding energy out of the interval $[t_1, t_2]$. One could also consider right-shifts but they are symmetric to left-shifts at time $t = 0$, so we will stick to left-shifts if not mentioned otherwise. This propagation concept can be formalized by the following rules.

Energetic Reasoning [1]. A complete energetic reasoning algorithm computes for every job $i = 1, \dots, n$ the release dates

$$r_i^* = \max_{\substack{(t_1, t_2) \in \mathcal{T}: \\ \omega_i^l(t_1, t_2) > 0}} \left(t_2 - p_i(t_1, t_2) + \frac{\omega(t_1, t_2)}{c_i} \right) \tag{7}$$

where \mathcal{T} is a set of relevant time intervals that is specified later.

In [1] it is shown that $|\mathcal{T}| \in O(n^2)$ and they present a complete algorithm with running time $O(n^3)$ to perform all propagations. In this paper, we consider an even stronger variant of energetic reasoning that is highly inspired by the edge-finding rule [21] that takes into account additional propagations from subintervals.

Energetic Edge-Finding. A complete energetic edge-finding algorithm computes for every job $i = 1, \dots, n$ the release dates

$$r_i^* = \max_{\substack{(t_1, t_2) \in \mathcal{T}: \\ \omega_i^l(t_1, t_2) > 0}} \max_{\substack{(t'_1, t'_2) \in \mathcal{T}: \\ [t'_1, t'_2] \subseteq [t_1, t_2], \\ \omega(t'_1, t'_2) + c_i \cdot (t_2 - t'_1 - p_i(t'_1, t'_2)) > 0}} \left(t'_2 - p_i(t'_1, t'_2) + \frac{\omega(t'_1, t'_2)}{c_i} \right) \tag{8}$$

where \mathcal{T} is the same set of time intervals as for energetic reasoning. The energetic edge-finding rule can be explained as follows. Once a left-shift overload with $\omega_i^l(t_1, t_2) > 0$ for a job i and interval $(t_1, t_2) \in \mathcal{T}$ is detected, it includes propagations of a subinterval $[t'_1, t'_2]$ if the energy of all jobs *different* from i in $[t'_1, t'_2]$ prevent an earlier processing of i . In this case, its release date r_i can be updated accordingly for $[t'_1, t'_2]$. For the set of relevant time intervals \mathcal{T} let

$$\begin{aligned} T_1 &= \{r_i, d_i - p_i : i = 1, \dots, n\} \\ T_2 &= \{d_i, r_i + p_i : i = 1, \dots, n\} \\ T_3(t) &= \{r_i + d_i - t : i = 1 \dots, n\} \end{aligned}$$

and from these sets we define

$$\begin{aligned} T_{12} &= \{(t_1, t_2) : t_1 < t_2, t_1 \in T_1, t_2 \in T_2\} \\ T_{13} &= \{(t_1, t_2) : t_1 < t_2, t_1 \in T_1, t_2 \in T_3(t_1)\} \\ T_{23} &= \{(t_1, t_2) : t_1 < t_2, t_2 \in T_2, t_1 \in T_3(t_2)\} \end{aligned}$$

where $\mathcal{T} = T_{12} \cup T_{13} \cup T_{23}$. Note that there exist tighter characterizations for the set of relevant time intervals for energetic reasoning [5, 20] but since they

are more complex we will stick to this definition. Moreover, in the first sections we only consider propagations for intervals in T_{12} and T_{23} . The integration of intervals in T_{13} needs special treatment and is examined separately in Sect. 3.4.

Since propagations from energetic edge-finding are not idempotent in general, we apply the rule until a fixpoint is reached. Because their overload conditions are equivalent, energetic edge-finding converges to the same fixpoint as energetic reasoning. However, we show that the additional integration of subintervals in energetic edge-finding leads to a strongly polynomial fixpoint convergence. To the best of our knowledge, the fixpoint complexity for only energetic reasoning (as stated) is unknown. The paper [15] claims a non-strongly polynomial fixpoint complexity for energetic reasoning but they use a weaker update function that can lead to slower convergence.

Theorem 1. *A complete energetic edge-finding algorithm reaches its fixpoint after at most $O(n^2)$ iterations.*

Proof. Let $T_2 = \{t_2^1, \dots, t_2^N\}$ with $t_2^b < t_2^{b+1}$ for all $b = 1, \dots, N - 1$. We show that after at most $b \cdot n$ iterations no further propagation can be detected in any interval $[t_1, t_2^b]$ with $t_1 < t_2^b$.

Consider an arbitrary fixpoint iteration q of the complete energetic edge-finding algorithm and assume by induction hypothesis that no propagation can be found in any interval $[t_1, t_2^b]$ for all $t_2^{b'} < t_2^b$. If job i is propagated in iteration q due to an interval $[t_1, t_2^b]$ with $\omega_i^l(t_1, t_2^b) > 0$ then since $\omega_i^l(t_1, t_2^b) = \omega(t_1, t_2^b) + c_i \cdot (p_i^l(t_1, t_2^b) - p_i(t_1, t_2^b))$ we have that $\omega(t_1, t_2^b)$ or $p_i^l(t_1, t_2^b) - p_i(t_1, t_2^b)$ increased from iteration $q - 1$ to q .

If $p_i^l(t_1, t_2^b) - p_i(t_1, t_2^b)$ has increased then job i must have been propagated in iteration $q - 1$. If $\omega(t_1, t_2^b)$ has not changed for all $t_1 < t_2^b$ from iteration $q - 1$ to q then energetic edge-finding would have found the propagation already in iteration $q - 1$ giving a contradiction¹.

Hence, $\omega(t_1, t_2^b)$ must have changed from iteration $q - 1$ to q . This happens if and only if a job $j \neq i$ with $d_j - p_j < t_2^b < d_j$ is propagated to end after time t_2^b . Then job j has a fixed part² in the interval $[d_j - p_j, t_2^b]$ to the beginning of iteration q . Moreover, the value $p_j(t_1, t_2^b)$ cannot be increased further by propagations from other intervals $[t_1', t_2^b]$, so job j cannot further increase $\omega(t_1, t_2^b)$. Because at most n jobs can increase $\omega(t_1, t_2^b)$ this way, we have that $q \leq b \cdot n$. Since $N \in O(n)$ the fixpoint is reached after $O(n^2)$ iterations. \square

It is an open question whether this fixpoint complexity is tight and if there exist examples where energetic reasoning needs $\Omega(n)$ more iterations to reach the fixpoint.

¹ Unlike energetic reasoning that may need additional iterations to reach the maximum propagation for t_2^b .

² Unlike standard edge-finding that does not consider partial overlaps, which gives a short proof of its $O(n)$ fixpoint complexity [14].

3 Algorithm

Our main algorithm iterates over all $t_2 \in T_2$ in an outer loop. We focus on the resulting subproblem in the time horizon $[0, t_2]$ for times $t \in T_1 \cup T_3(d)$. Time intervals $(t_1, t_2) \in T_{23}$ are integrated separately in Sect. 3.4. Throughout the rest of the paper we will consider a fixed t_2 value and set the due date $d = t_2$ as a *global constant*. For abbreviation, we therefore omit $d = t_2$ as function argument and rewrite $p_i(t) = p_i(t, d)$, $p_i^l(t) = p_i^l(t, d)$, $e(t) = e(t, d)$, $\omega(t) = \omega(t, d)$, $\omega_i^l(t) = \omega_i^l(t, d)$ and $T = \{t : t < d, t \in T_1 \cup T_3(d)\}$.

Our algorithm is divided into three phases: *decomposition phase*, *detection phase* and *update phase*.

3.1 Decomposition Phase

In the decomposition phase, we will decompose the available energy $e(0)$ in the interval $[0, d]$ into energy blocks B_1, \dots, B_m in order to get a stronger representation for the overload function $\omega(t)$.

Let the *energy envelope* at time $t < d$ be defined as

$$E(t) = t + \frac{e(t)}{C} \tag{9}$$

that is a lower bound on the maximum completion time of jobs i with $p_i(t) > 0$. Then we have $\omega(t) = C \cdot (E(t) - d)$.

From this definition, we create energy blocks as follows. Let $T = \{t_1, \dots, t_H\}$ with $t_h < t_{h+1}$ for all $h = 1, \dots, H - 1$. Starting with $h = 1$, we compute the next greater $h' > h$ such that $E(t_h) < E(t_{h'})$. If $e(t_h) > e(t_{h'}) > 0$ we create a new *canonical block* B with release date $r(B) = t_h$, processing time $p(B) = \frac{e(t_h) - e(t_{h'})}{C}$ and resource demand C . Then we set $h = h'$ and repeat the procedure. If no h' can be found we finally set $p(B) = \frac{e(t_h)}{C}$ and stop.

The set of all canonical blocks B_1, \dots, B_m is also denoted as the *canonical decomposition* [8]. In the following, let $T_B = \{r(B_1), \dots, r(B_m)\}$ be the set of release dates of the canonical blocks. Since every canonical block B_l has resource demand C , the canonical decomposition can be interpreted as a scaled single machine schedule, see Fig. 1. Moreover, the energy values $e(t_h)$ can be computed and maintained by the algorithm of Baptiste et al. [1] in $O(n)$, so the canonical decomposition can be computed in $O(n)$ time.

A further important observation is that we can restrict to non-dominated energy envelopes. That is, if $E(t') > E(t)$ for $t' < t$ then we can consider $E(t')$ instead of $E(t)$ for time t . Hence, we can replace the function $E(t)$ by

$$\tilde{E}(t) = \max_{t' \leq t} E(t') \tag{10}$$

where $\tilde{\omega}(t) = C \cdot (\tilde{E}(t) - d)$ is a stronger lower bound on the overload in $[t, d]$. Consequently, we can replace $\omega(t)$ by $\tilde{\omega}(t)$ in our energetic edge-finding rule. In particular, we have $\omega(t) = \tilde{\omega}(t)$ for all $t \in T_B$ and since $\tilde{E}(t)$ is monotonically

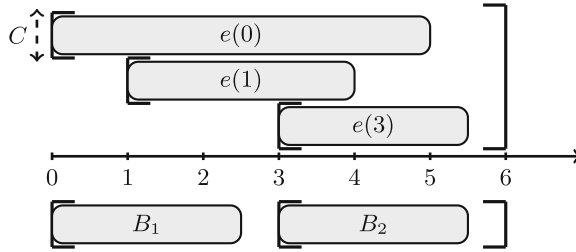


Fig. 1. Canonical decomposition: the three energy blocks (above) show the energy envelopes $E(t)$ for $t \in \{0, 1, 3\}$ with due date $d = 6$ for some CuSP instance. We obtain the canonical decomposition B_1, B_2 (below) that shows more precisely where the energy is induced. If there exists an energy overload for any interval $[t, d]$ then it is always generated by the last canonical block B_m .

non-decreasing in t we also have that $\tilde{\omega}(t)$ is monotonically non-decreasing in t . We will highly exploit this fact in our algorithms.

We can further use the canonical decomposition to efficiently compute $\tilde{\omega}(t)$. For given time t , let B_l be the canonical block with smallest index l such that $t \leq r(B_l) + p(B_l)$ then we have

$$\tilde{\omega}(t) = C \cdot \left(\min\{t, r(B_l)\} + \sum_{l' \geq l} p(B_{l'}) - d \right) \tag{11}$$

where this definition depends only on the canonical decomposition. We will use the canonical decomposition for the next phases that depend only on $T_B \subseteq T$.

3.2 Detection Phase

In the detection phase, we check for every job $i = 1, \dots, n$ if there exists a time $t \in T$ such that $\omega_i^l(t) > 0$. In this case, job i must end strictly after time d and this relation is stored in an n -dimensional array end by setting $end[i] = d$, similar to [21]. For given due date d , we compute all such relationships as follows. The condition $\omega_i^l(t) > 0$ is equivalent to

$$c_i \cdot (p_i^l(t) - p_i(t)) > -\omega(t) \tag{12}$$

where both left- and right-hand side are non-negative since $\omega(t) \leq 0$ for all $t \leq d$. We replace $\omega(t)$ by the stronger overload function $\tilde{\omega}(t)$ which yields

$$c_i \cdot (p_i^l(t) - p_i(t)) > -\tilde{\omega}(t) \tag{13}$$

and this is still a valid condition since $\omega(t) \leq \tilde{\omega}(t)$ and thus not less intervals are checked for energetic edge-finding (8).

Moreover, both functions on the left- and right-hand side of (13) are piecewise linear in t . Let $p_i^s = p_i^l(0) - p_i(0)$ then the function on the left-hand side decomposes into the segments:

$$c_i \cdot (p_i^l(t) - p_i(t)) = \begin{cases} c_i \cdot p_i^s, & t \leq r_i \\ c_i \cdot (p_i^s - t + r_i), & r_i \leq t < r_i + p_i^s \\ 0, & \text{else} \end{cases} \tag{14}$$

for all jobs $i = 1, \dots, n$ where we only consider jobs i with $p_i^s > 0$ or jobs with positive left-shift slack respectively.

We have to compute for every $i = 1, \dots, n$ the time $t \in T$ for which inequality (13) is maximally satisfied. For this, we compute the upper envelope of the line segments (14) of all jobs $i = 1, \dots, n$ by using a sweep line approach [4]. The sweep line data structure comprises a binary status tree W and an event heap H . The status tree W changes dynamically with t and stores the line segments as leaves while maintaining the line segment of maximum value $c_i \cdot (p_i^l(t) - p_i(t))$ for the current t in the root node using a bottom-up approach. The event heap H stores time events at which the order of two line segments in W is changing or the maximum line segment of W must be retrieved from the root node. New events are added dynamically to H . Since there are $O(n)$ line segments, one line segment can be added or deleted in $O(\log n)$ and sweeping over all $t \in T$ takes $O(n \log n)$ since for every node in W at most one event is added dynamically to H . In particular, we can restrict to $T_B \subseteq T$ since $\tilde{\omega}(t)$ is locally maximal there.

Thus, in our algorithm we sweep over all $t \in T_B$ in decreasing order and retrieve the maximum function value $v_i = c_i \cdot (p_i^l(t) - p_i(t))$ of the currently dominating job i from the root node of W . While the currently dominating job i satisfies $v_i > -\tilde{\omega}(t)$ we set $end[i] = d$ and delete all line segments that belong to job i (the dominating job may change) and store the update value $r_i^d = d - p_i(t) + \frac{\tilde{\omega}(t)}{c_i}$. Hence, for the given due date d the detection phase takes $O(n \log n)$ time, compare Algorithm 1.

Algorithm 1. detection phase

Input: due date d , canonical decomposition B_1, \dots, B_m

Output: $end[i]$ and r_i^d for all $i = 1, \dots, n$,

```

1 initialize sweep line tree  $W$  for all jobs  $i$  with  $r_i < d < d_i$ 
2  $E \leftarrow 0$ 
3 forall the  $l = m, \dots, 1$  do
4    $E \leftarrow E + C \cdot p(B_l)$ 
5    $t \leftarrow r(B_l)$ 
6    $sweep\_to(W, t)$ 
7   while  $i \leftarrow W.root.job$  and  $E + c_i \cdot (p_i^l(t) - p_i(t)) > C \cdot (d - t)$  do
8      $end[i] \leftarrow \max\{end[i], d\}$ 
9      $r_i^d \leftarrow d - p_i(t) + \frac{1}{c_i} \cdot (E - C \cdot (d - t))$ 
10    delete line segments of  $i$  from  $W$ 
11 return  $end[i]$  and  $r_i^d$  for all  $i = 1, \dots, n$ 

```

An improvement can be made by omitting the constant line segment for $t \leq r_i$ in (14) since $\tilde{\omega}(t)$ is monotonically non-decreasing and therefore inequality (13) is maximally satisfied at $t = r_i$ but this point is also covered by the second line segment. In this case, we need to add an event at $t = r_i$ to H to retrieve possible overloads from W .

A special case occurs when we have $c = c_i$ for all $i = 1, \dots, n$ respectively. In this case, all line segments have the same slope so we can use a simple queue to process the line segments from right to left where the first line segment that is added remains the dominating one until it is deleted. This allows a processing in $O(n)$ and will yield a complete $O(n^2)$ energetic edge-finding algorithm.

Proposition 1. *Let $t^* \in T_B$ be the time where we set $end[i] = d$ for a job i . Then we have $t^* < r_i + p_i^s$ and t^* also maximizes*

$$\max_{\substack{t \in T: \\ \tilde{\omega}(t) + c_i \cdot (p_i^l(t) - p_i(t)) > 0}} \left(d - p_i(t) + \frac{\tilde{\omega}(t)}{c_i} \right). \tag{15}$$

Proof. As mentioned, we can replace T by T_B in the given formula. Whenever the algorithm sets $end[i] = d$ for $t^* \in T_B$, we have $\tilde{\omega}(t^*) + c_i \cdot (p_i^l(t^*) - p_i(t^*)) > 0$ and therefore $p_i^l(t^*) - p_i(t^*) > 0$ which holds only if $t^* < r_i + p_i^s$.

For all $t < r_i + p_i^s$ we have $p_i(t) = p_i(0)$ is constant. Thus, the overload function turns into $\tilde{\omega}(t) + c_i \cdot (p_i^l(t) - p_i(0)) > 0$ and the update function into $d - p_i(0) + \frac{\tilde{\omega}(t)}{c_i}$ that is equivalent to maximizing $\tilde{\omega}(t)$. Since the update function is monotonically non-decreasing in t , the maximum t that satisfies $\tilde{\omega}(t) + c_i \cdot (p_i^l(t) - p_i(0)) > 0$ is optimal for (15). Since the algorithm iterates over all $t \in T_B$ in decreasing order, we obtain the optimal time $t = t^*$. □

Adding the $O(n)$ iterations of every due date d from the outer loop and adding the intervals in T_{23} treated later yields the following result.

Corollary 1. *Computing the detection phase for every due date $d \in T_2$ yields a complete energetic reasoning algorithm with running time $O(n^2 \log n)$.*

In Practice. The sweep line algorithm improves the theoretical performance of energetic reasoning. The necessity of computing the upper envelope is due to the fact that there might exist instances in which, say $O(n)$ many, line segments are active at the same time t . In this case, we always need to keep track of the maximum line segment that may change for small deviations of t such that the upper envelope needs to be explored completely. In practice however, the line segments of the jobs are mostly disjoint and generally only few are active at the same time. Moreover, upper envelope computations involve complex data structures that need to be build in every iteration that generates overhead for instances where n is small.

Thus, we decided to implement the following more output-sensitive but still complete version: we sweep from right to left but whenever a line segment becomes active we add it into a list L . If it becomes inactive we delete it from L , both can be done in $O(1)$ by storing job pointers. Then for every $t \in T_B$ we

iterate all elements in L to detect the line segment of maximum value. As mentioned, the size of L is practically small (mostly zero). This leads to a complete complexity of $O(n + m \cdot h)$ for the detection phase where h is the maximum number of simultaneously active line segments. Multiplying the $O(n)$ iterations of the outer loop, this is $O(n^3)$ in general but much faster on practical instances.

In order to guarantee a stable fixpoint behavior we additionally include possible propagations from all subintervals. The next section describes how to compute them efficiently in an additional phase.

3.3 Update Phase

For a given due date d , the update phase computes for every job $i = 1, \dots, n$ with $p_i^s > 0$ and $d \leq \text{end}[i]$ stronger release dates by including propagations from subintervals. Again, let $p_i^s = p_i^l(0) - p_i(0)$. By Proposition 1, the maximum propagation for all $t \in T_B$ with $t < r_i + p_i^s$ is already found in the detection phase. Thus, assume that $t \geq r_i + p_i^s$ which leads to the problem of computing

$$r_i^d = \max_{\substack{t \in T_B: \\ t \geq r_i + p_i^s \\ \omega(t) + c_i \cdot (d - t - p_i(t)) > 0}} \left(d - p_i(t) + \frac{\omega(t)}{c_i} \right). \tag{16}$$

Again, we replace $\omega(t)$ by its stronger version $\tilde{\omega}(t)$ and set

$$r_i^d = \max_{\substack{t \in T_B: \\ t \geq r_i + p_i^s \\ \tilde{\omega}(t) + c_i \cdot (d - t - p_i(t)) > 0}} \left(d - p_i(t) + \frac{\tilde{\omega}(t)}{c_i} \right) \tag{17}$$

where we can restrict to $t \in T_B$ by the same argument as for the detection phase. We want to simplify formula (17). Consider the parametrization

$$r_i^d(c) = \max_{\substack{t \in T_B: \\ \tilde{\omega}(t) + c \cdot (d - t) > 0}} \left(d + \frac{\tilde{\omega}(t)}{c} \right) \tag{18}$$

for variable resource demands c . We use the following lemma.

Lemma 1. *Given a job $i \in \{1, \dots, n\}$ then $t \in T_B$ with $t \geq r_i + p_i^s$ is optimal for (17) if and only if t is optimal for (18) with $r_i^d = r^d(c_i) - p_i(t) > t$.*

Proof. Assume first that $t \in T_B$ with $t \geq r_i + p_i^s$ is optimal for (17). Then $r_i^d + p_i(t) = d + \frac{\tilde{\omega}(t)}{c_i}$ and since $r_i^d > t$ we also have that $r_i^d + p_i(t) > t$ that yields $\tilde{\omega}(t) + c_i \cdot (d - t) > 0$, hence the point $t \in T_B$ is valid for (18) that implies $r_i^d + p_i(t) \leq r^d(c_i)$.

Now assume that $t' \in T_B$ is optimal for $r^d(c_i)$ with $r^d(c_i) > r_i^d + p_i(t)$ where $t' > t$ by monotonicity of $\tilde{\omega}(t)$. Since $t \geq r_i + p_i^s$ it holds $p_i(t) - p_i(t') = \max\{0, t' - t\}$. Thus, if $p_i(t') = p_i(t) - t + t'$ we get $\tilde{\omega}(t') + c_i \cdot (d - t' - p_i(t')) \geq \omega(t) + c_i \cdot (d - t - p_i(t)) > 0$ by feasibility of t for (17). Otherwise, if $p_i(t') = p_i(t)$

Algorithm 2. update phase

Input: due date d , canonical decomposition B_1, \dots, B_m ,
 resource demands $\{c_1, \dots, c_k\}$
Output: $t^d(c)$ and $r^d(c)$ for all $c \in \{c_1, \dots, c_k\}$

```

1  $h \leftarrow k, l \leftarrow m, E \leftarrow 0$ 
2 while  $h \geq 1$  and  $l \geq 1$  do
3      $t \leftarrow r(B_l)$ 
4     if  $E - (C - c_h) \cdot (d - t) > 0$  then
5          $r^d(c_h) \leftarrow d + \frac{1}{c_h} \cdot (E - C \cdot (d - t))$ 
6          $t^d(c_h) \leftarrow t$ 
7          $h \leftarrow h - 1$ 
8     else
9          $l \leftarrow l - 1$ 
10         $E \leftarrow E + p(B_l) \cdot C$ 
11 return  $t^d(c)$  and  $r^d(c)$  for all  $c \in \{c_1, \dots, c_k\}$ 
    
```

we also have $\tilde{\omega}(t') + c_i \cdot (d - t' - p_i(t')) \geq \tilde{\omega}(t) + c_i \cdot (d - t - p_i(t)) > 0$. Hence, the solution at $t' \in T_B$ is also feasible for (17) and its objective value is equal to $r^d(c_i) - p_i(t') > r_i^d$ which contradicts the assumption that r_i^d is optimal for (17). It follows that $r^d(c_i) \leq r_i^d + p_i(t)$ and therefore $r^d(c_i) = r_i^d + p_i(t)$ which shows the statement. \square

Lemma 1 allows us to work with the parametrized version (18): if we have computed $r^d(c_i)$ with optimal time $t \in T_B$ we only have to verify that $r^d(c_i) - p_i(t) > t$ and set $r_i^d = r^d(c_i) - p_i(t)$, otherwise there exists no better update.

In the following we show how to compute $r^d(c)$ for all $c \in \{c_1, \dots, c_n\}$ with $k = |\{c_1, \dots, c_n\}|$ efficiently. Substituting $g(c, t) = d + \frac{\tilde{\omega}(t)}{c}$ yields

$$r^d(c) = \max_{\substack{t \in T_B: \\ g(c, t) > t}} g(c, t) \tag{19}$$

that now has a quite simple form. For fixed c , the function $g(c, t)$ is piecewise linear and non-decreasing in t . Thus, for given resource demand $c \in \{c_1, \dots, c_n\}$ the largest value $t \in T_B$ with $g(c, t) > t$ is optimal for $r^d(c)$. However, $g(c, t)$ is also non-decreasing in c since $\tilde{\omega}(t) \leq 0$. Hence, we iterate over all $t \in T_B$ in decreasing order as long as $g(c, t) \leq t$. Starting with the last considered resource demand c , we set $r^d(c) = g(c, t)$ as long as $g(c, t) > t$ for all c in decreasing order. We repeat the same search starting with the next smaller t until either $r^d(c)$ is determined for the smallest c value or the minimum of T_B is reached. This procedure takes $O(k + m)$ to compute all $r^d(c)$, see also Algorithm 2.

Since the full algorithm iterates over all due dates $d \in T_2$ in an outer loop and by including the detection phase every inner iteration takes $O(n \log n + k + m)$ we get a final running time of $O(n^2 \log n)$ for energetic edge-finding that is subsumed in Algorithm 3. After termination of the outer loop, we finally update the release date of every job $i = 1, \dots, n$ with $p_i^s > 0$ by

Algorithm 3. energetic edge-finding

Input: CuSP instance

Output: propagated release dates r_i^* from energetic edge-finding

```

1  $end[i] \leftarrow -\infty$  for all  $i = 1, \dots, n$ 
2 forall the  $d \in T_2$  in decreasing order do
3    $B_1, \dots, B_m \leftarrow decomposition\_phase(d)$ 
4    $(end, r^d) \leftarrow detection\_phase(d, B)$ 
5    $(t^d(c), r^d(c)) \leftarrow update\_phase(d, B)$ 
6 forall the  $i = 1, \dots, n$  do
7    $r_i = \max\{r_i^d, r^d(c_i) - p_i(t^d(c_i)) : d \leq end[i], d \in T_2\}$ 
8 return  $r_i$  for all  $i = 1, \dots, n$ 

```

$$r_i^* = \max\{r_i, \max\{r_i^d : d \leq end[i], d \in T_2\}\} \tag{20}$$

that takes an additional $O(k \cdot n)$ time.

3.4 Integration of Symmetric Intervals

In this section, we show how the remaining set of time intervals T_{23} can be integrated into the given concepts. The main problem here is that d depends on t and not vice versa. A first important observation is that left-shift propagations on T_{23} are symmetric to *right-shift* propagations on T_{13} . Thus, our approach is to include the line segments of the corresponding right-shift slack function $c_i \cdot (p_i^r(t) - p_i(t)) > -\tilde{\omega}(t)$ with $p_i^r(t) = \max\{0, \min\{p_i, d - t, d_i - t, d - d_i + p_i\}\}$ with the propagation $d_i^* = t + p_i(t) - \frac{\tilde{\omega}(t)}{c_i}$ and set $start[i] = t$. In order to perform the update phase we have to build the canonical decomposition in $[t, \infty)$ but now according to due dates and from right to left and proceed symmetrically. The last step is not implemented in our algorithm. However, it yields a complete energetic reasoning algorithm. In respect to the scope of this paper, we will not explicitly formulate this case during the next sections.

4 Relation to Standard and Extended Edge-Finding

Energetic edge-finding is strongly motivated by standard edge-finding [21]. In particular, energetic edge-finding can be slightly modified to give a complete $O(n^2)$ edge-finding but that additionally performs stronger propagations from energetic reasoning.

For every interval $[t, d]$ standard edge-finding considers only propagations of jobs i with $t \leq r_i < d$. We can replace our detection phase to include all such intervals for fixed d in linear time as follows: we iterate over all r_i with $r_i < d$ in non-increasing order and check immediately if $\tilde{\omega}(t) + c_i \cdot (p_i^l(t) - p_i(t)) > 0$ holds at $t = r_i$. Under the condition $t \leq r_i$, we have by monotonicity of $\tilde{\omega}(t)$ that $t = r_i$ maximally satisfies this inequality. If the inequality is satisfied we set

$end[i] = d$, otherwise, no propagation can be found for job i . The update phase is executed as previously introduced. Note that we simultaneously iterate over $t \in T_B$ in decreasing order to get the $\tilde{\omega}(t)$ values.

Consequently, for fixed due date d the detection phase takes $O(n+m)$, see also Algorithm 2. In total, this yields a complete edge-finding algorithm in $O(n^2)$ time but that additionally performs stronger propagations because the basic energy in every interval $[t, d]$ is taken from energetic reasoning. Since the currently best complete edge-finding algorithm has complexity $O(kn \log n)$ [21], our algorithm constitutes a further improvement in the landscape of energetic propagators.

Corollary 2. *There exists a complete $O(n^2)$ algorithm for edge-finding that additionally takes energetic reasoning as energy lower bound in each time interval.*

A further open question asks for the relation to *extended edge-finding* [14] that includes propagations for partially overlapping jobs. In particular, the case of partially overlapping jobs is exactly the bottleneck of our algorithm since by the 'usual' approach of fixing one interval limit, say $t_2 = d$, and computing the partially overlapping job i in an interval $[t, d]$ with $r_i < t < r_i + p_i^s$ that *maximally* contributes to the energy in $[t, d]$ will always lead to an upper envelope problem. Hence, it is unlikely that there exists a complete $O(n^2)$ algorithm for extended edge-finding by using known techniques. In turn, if there exists an efficient method to compute such propagations then we believe it can also be used for energetic reasoning, thus indicating that extended edge-finding and energetic reasoning are of the same complexity.

5 Improving Propagations by Detectable Precedences

In this section we present a relaxation of energetic edge-finding and a possible improvement for energetic edge-finding.

In every propagation that is performed for job i due to a left-shift overload in the time interval $[t, d]$ we can improve the propagation by considering the minimum earliest completion time of the jobs that are contributing to the energy in $[t, d]$. In other words, if $\tilde{\omega}(t) + c_i \cdot (p_i^l(t) - p_i(t)) > 0$ then we can propagate

$$r_i^d = \min_{\substack{j \neq i: \\ p_j(t) > 0}} (r_j + p_j). \tag{21}$$

To the best of our knowledge, we believe that this rule, in its generality to energetic reasoning, has not been investigated so far. Since our algorithms iterates over all t in non-increasing order we can include such propagations by storing an update value that is set to $r_j + p_j$ whenever $t = r_j + p_j$ for jobs j with $p_j(0) > 0$. As for energetic edge-finding, we can extend this rule by propagations from subintervals of $[t, d]$ or subsets of jobs respectively.

In the following we show that detectable precedences naturally extends energetic edge-finding since both propagations are incomparable.

Example 1. Given a CuSP instance of four jobs with $p_1 = p_2 = p_3 = p$ and $p_4 = 2p$ for some large p . Moreover, let $c_i = 1$ for all $i = 1, \dots, n$ and $C = 2$. The scheduling intervals $[r_i, d_i]$ for $i = 1, \dots, 4$ are given by $[0, 3p - 1]$, $[0, 2p]$, $[0, 2p]$ and $[0, \infty)$. Energetic reasoning propagates job 4 according to the interval $[0, 2p]$ since for $d = 2p$ we have $\tilde{\omega}(0) + c_4 \cdot (p_4^l(0) - p_4(0)) = (2p + 1 - 4p) + (2p - 0) = 1 > 0$ and thus we propagate $r_4^{2p} = d - p_4(0) + \tilde{\omega}(0) = 2p - 0 + (2p + 1 - 4p) = 1$. After that, no further propagation is detected, so the fixpoint is reached for energetic edge-finding and all dominated rules. In contrast, detectable precedences (21) finds that $\min\{r_j + p_j : p_j(0) > 0, j \neq 4\} = p$ and thus we propagate $r_4^{2p} = p$. This is the strongest possible propagation. Note that the not-first/not-last rule [18] does not consider the partial overlap of job 1, so nothing is propagated.

Example 2. Given a CuSP instance of three jobs with $p_1 = p_2 = p$ and $p_3 = 1$ for some $p > 0$. Moreover, let $c_1 = c_2 = c_3 = 1$ and $C = 1$. The scheduling intervals $[r_i, d_i]$ for $i = 1, 2, 3$ are given by $[0, 2p]$, $[0, 2p]$, $[0, \infty)$. The interval $[0, 2p]$ contains full energy of $2p$. Thus, detectable precedences updates $r_3 = p$ while energetic reasoning (even edge-finding) updates the strongest possible propagation $r_3 = 2p$.

Corollary 3. *Detectable precedences and energetic reasoning are incomparable.*

The given examples reveal the weakness of current energetic approaches: the lack of knowledge about the inner structure of an interval and its possible realizations. Future propagators should make stronger predictions by analyzing how possible realizations of a certain interval may look like.

6 Computational Results

In the following we give a rough analysis of the computational performance of the presented methods. We use the test sets J30, J60 and J120 that contain 480, 480 and 600 instances respectively from the well-established PSPLIB [13] for the Resource-Constrained Project Scheduling Problem (RCPSp). The name of the test set indicates the number of considered jobs and every instance has four resources with additional precedence constraints.

Our algorithms are programmed in C language with GCC compiler version 7.3.0 and executed on a Intel Xeon CPU E5-2660 with 2.60 GHz using a single thread. We compute lower bounds to the RCPSp by destructive improvement, that is we start with a lower bound on the makespan and increase it as long as we detect infeasibility or the time limit of 600 seconds is reached.

We use a *static branching rule* [17] that schedules the jobs in order of earliest release dates. Ties are broken by minimum domain value $d_i - p_i - r_i$. Additionally, we apply a dominance rule that skips the right branch of the currently branched job i if it does not contribute to the earliest resource conflict when all jobs are scheduled at their release dates. We consider a static branching rule rather than a dynamic one to compare the real performance between the different propagators. For the computation of best possible solutions to the RCPSp, we need more sophisticated methods [17] that exceed the scope of this paper.

In every search node we first apply timetabling [7, 12] in combination with one of the following: energetic reasoning as given in [1] with complexity $O(n^3)$, energetic edge-finding, energetic edge-finding without the update phase and the incomplete $O(n^2)$ energetic edge-finding algorithm of Sect. 4. All energetic edge-finding propagators include propagations from detectable precedences.

Table 1 displays the number of optimal solutions found (opt), the deviation in the total sum of lower bounds (Δ LB) normed to the weakest propagator and the average number of backtracking nodes per second for instances that took longer than five seconds to solve. As expected the $O(n^3)$ energetic reasoning propagator is slower than energetic edge-finding (EnEF) such that energetic edge-finding is able to compute better lower bounds in total. The exclusion of the update phase and the relaxed variant yield speedups only on small instances. We conclude that integrating all subintervals has mainly a theoretical rather than practical relevance. Surprisingly, the relaxed version of energetic edge-finding cannot profit from its $O(n^2)$ complexity on large instances. The reason is that our output-sensitive implementation of the detection phase runs in almost the same time because the left-shift slack intervals of the jobs are mostly disjoint.

Table 1. Computational results for the J30, J60 and J120 test sets.

	J30			J60			J120		
	Opt	Δ LB	Nodes/s	Opt	Δ LB	Nodes/s	Opt	Δ LB	Nodes/s
ER	384	0	48.70	347	0	15.96	141	0	8.32
EnEF	393	126	581.35	349	86	249.68	143	77	92.93
EnEF (w/o up)	394	174	662.76	349	86	246.44	143	77	94.12
EnEF (relaxed)	405	221	680.25	348	85	245.26	143	79	96.63

7 Conclusion

We believe that the monotonicity in the single machine interpretation of energetic schedules can lead to new ideas for faster propagation algorithms for the CuSP and related problems. Since energetic arguments have natural limitations, one future direction can be to combine energetic propagations with arguments about the inner structure of time intervals to derive stronger propagations. A first approach is made with detectable precedences. Another open question is to resolve the fixpoint complexity of energetic reasoning.

Acknowledgements. The author would like to thank anonymous reviewers for their helpful comments on the paper, especially for the advice to take a simpler representation of $\tilde{\omega}(t)$ as given in the current version of the paper.

References

1. Baptiste, P., Le Pape, C., Nuijten, W.: Satisfiability tests and time bound adjustments for cumulative scheduling problems. *Ann. Oper. Res.* **92**, 305–333 (1999)
2. Berthold, T., Heinz, S., Schulz, J.: An approximative criterion for the potential of energetic reasoning. In: Marchetti-Spaccamela, A., Segal, M. (eds.) *TAPAS 2011*. LNCS, vol. 6595, pp. 229–239. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19754-3_23
3. Bonifas, N.: A $O(n^2 \log(n))$ propagation for the Energy Reasoning, Conference Paper, Roadef 2016. (2016)
4. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.C.: Computational Geometry. In: de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.C. (eds.) *Computational Geometry*. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-662-04245-8_1
5. Derrien, A., Petit, T.: A new characterization of relevant intervals for energetic reasoning. In: O’Sullivan, B. (ed.) *CP 2014*. LNCS, vol. 8656, pp. 289–297. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_22
6. Erschler, J., Lopez, P.: Energy-based approach for task scheduling under time and resources constraints. In: *2nd International Workshop on Project Management and Scheduling*, pp. 115–121 (1990)
7. Gay, S., Hartert, R., Schaus, P.: Simple and scalable time-table filtering for the cumulative constraint. In: Pesant, G. (ed.) *CP 2015*. LNCS, vol. 9255, pp. 149–157. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_11
8. Goemans, M.X.: A supermodular relaxation for scheduling with release dates. In: Cunningham, W.H., McCormick, S.T., Queyranne, M. (eds.) *IPCO 1996*. LNCS, vol. 1084, pp. 288–300. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61310-2_22
9. Hooker, J.N.: A hybrid method for planning and scheduling. In: Wallace, M. (ed.) *CP 2004*. LNCS, vol. 3258, pp. 305–316. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30201-8_24
10. Kameugne, R., Fetgo, S.B., Fotso, L.P.: Energetic extended edge finding filtering algorithm for cumulative resource constraints. *Am. J. Oper. Res.* **3**(06), 589 (2013)
11. Lenstra, J.K., Kan, A.R., Brucker, P.: Complexity of machine scheduling problems. *Ann. Discrete Math.* **1**, 343–362 (1977)
12. Letort, A., Beldiceanu, N., Carlsson, M.: A scalable sweep algorithm for the *cumulative* constraint. In: Milano, M. (ed.) *CP 2012*. LNCS, pp. 439–454. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33558-7_33
13. Kolisch, R., Sprecher, A.: PSPLIB—a project scheduling problem library: OR software-ORSEP operations research software exchange program. *Eur. J. Oper. Res.* **96**(1), 205–216 (1997)
14. Mercier, L., Van Hentenryck, P.: Edge finding for cumulative scheduling. *INFORMS J. Comput.* **20**(1), 143–153 (2008)
15. Mercier, L., Van Hentenryck, P.: Strong polynomiality of resource constraint propagation. *Discrete Optim.* **4**(3–4), 288–314 (2007)
16. Ouellet, P., Quimper, C.-G.: Time-table extended-edge-finding for the cumulative constraint. In: Schulte, C. (ed.) *CP 2013*. LNCS, vol. 8124, pp. 562–577. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_42
17. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Explaining the cumulative propagator. *Constraints* **16**(3), 250–282 (2011)

18. Schutt, A., Wolf, A.: A new $\mathcal{O}(n^2 \log n)$ not-first/not-last pruning algorithm for cumulative resource constraints. In: Cohen, D. (ed.) CP 2010. LNCS, vol. 6308, pp. 445–459. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15396-9_36
19. Schutt, A., Feydy, T., Stuckey, P.J.: Explaining time-table-edge-finding propagation for the cumulative resource constraint. In: Gomes, C., Sellmann, M. (eds.) CPAIOR 2013. LNCS, vol. 7874, pp. 234–250. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38171-3_16
20. Tesch, A.: A nearly exact propagation algorithm for energetic reasoning in $\mathcal{O}(n^2 \log n)$. In: Rueher, M. (ed.) CP 2016. LNCS, vol. 9892, pp. 493–519. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44953-1_32
21. Vilím, P.: Edge finding filtering algorithm for discrete cumulative resources in $\mathcal{O}(kn \log n)$. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 802–816. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_62
22. Vilím, P.: Timetable edge finding filtering algorithm for discrete cumulative resources. In: Achterberg, T., Beck, J.C. (eds.) CPAIOR 2011. LNCS, vol. 6697, pp. 230–245. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21311-3_22

CP, Optimization, and Power System Management Track



A Fast and Scalable Algorithm for Scheduling Large Numbers of Devices Under Real-Time Pricing

Shan He^{1,2}(✉), Mark Wallace¹, Graeme Gange¹, Ariel Liebman¹,
and Campbell Wilson¹

¹ Faculty of Information Technology, Monash University, Melbourne, Australia
{shan.he,mark.wallace,graeme.gange,ariel.liebman,
campbell.wilson}@monash.edu

² Data61/CSIRO, Melbourne, Australia

Abstract. Real-time pricing (RTP) is a financial incentive mechanism designed to encourage demand response (DR) to reduce peak demand in medium and low voltage distribution networks but also impacting the generation and transmission system. Though RTP is believed to be an effective mechanism, challenges exist in implementing RTP for residential consumers wherein manually responding to a changing price is difficult and uncoordinated responses can lead to undesired peak demand at what are normally off-peak times. Previous research has proposed various algorithms to address these challenges, however, they rarely consider algorithms that manage very large numbers of houses and devices with discrete consumption levels. To optimise conflicting objectives under RTP prices in a fast and highly scalable manner is very challenging. We address these issues by proposing a fast and highly scalable algorithm that optimally schedules devices for large numbers of households in a distributed but non-cooperative manner under RTP. The results show that this algorithm minimises the total cost and discomfort for 10,000 households in a second and has a constant computational complexity.

1 Introduction

Demand response (DR) refers to activities performed by consumers to reduce or time-shift electrical demand in response to appropriate, often financial, incentives. It has been proposed and trialled over several decades as an effective low-cost way to improve the efficient utilisation of electricity infrastructure. A key goal of DR is to reduce peak electricity demand, which requires significant infrastructure to support.

Real-time pricing (RTP) is a type of financial incentive proposed for DR that mirrors the type of pricing seen in deregulated wholesale electricity markets. This method offer 'dynamic prices' that increase with increasing demand and change over time. Economists argue that RTP is the most effective way to incentivise peak demand reduction [1]. However, it is difficult to implement for residential

consumers due to three challenges [18,21,22,24,28]: (1) manually responding to a constant changing price is difficult, (2) the lack of insights into existing consumption patterns of consumers and (3) *load synchronization* or *rebound peak*, referring to undesired demand peaks that may be higher than the original peak as a result of uncoordinated responses.

In recent years much work has been done to overcome these challenges. Many algorithms have been designed to shift electrical demand from peak to off-peak times in ways that minimise total cost for all consumers or best balance the needs for comfort and cost savings.

Existing methods for shifting demand can be categorized into two types: centralised and distributed. Centralised methods concentrate all decisions within a single agent that schedules all households' devices to minimise electricity costs against prices provided a day in advance. Many argue that these methods are impractical because they are not computationally scalable and require the details of *all shiftable devices in advance* [12,13,15,23,25,29,31,33,35]. Distributed methods attempt to address these drawbacks by allowing consumers to independently schedule devices to by minimising their own cost and discomfort locally while providing some central coordination. However, the naive distributed alternative without any coordination leads to each household optimising their schedules against a common RTP, leading to *load synchronization* problem where consumption will be concentrated during the cheaper periods [11,18,21,23,32].

Improved distributed methods have been proposed to address *load synchronization* problem that can be divided into two types: cooperative and non-cooperative. Cooperative methods coordinate consumers by requiring them to iteratively broadcast their demand profiles to all other consumers. One of the criticisms of this approach is that this process poses an extra data communication burden on the network and consumers may not want to share their demand information due to privacy concerns. In response other research has focussed on non-cooperative methods that only require iterative communication with a central agent hosted by their utility or a specialised demand aggregator entity.

Our work focuses on distributed non-cooperative (DN) methods. Some existing methods focus on a central agent calculating the optimal demand levels for households that will minimise the overall costs and reduce the peak demand without considering details of devices. In this sort of scheme a household is required to schedule its own shiftable devices to ensure their peak demand remains below the given optimal demand level. Other DN methods focus on finding the best times to use devices for each household. However, the problem of minimising total costs by scheduling devices of multiple households with discrete consumption constraints and preferences against prices that are not known in advance is NP-hard. Some DN methods either drop the integer constraint nature of device demand and solve a relaxed version of the original problem or use heuristic methods resulting in sub-optimal solutions. Moreover, some require manual parameter tuning to ensure convergence. Additionally, existing methods rarely consider multiple global constraints for each household such as the maximum demand constraint or sequential constraints that defines the order of some shiftable devices,

e.g. dryers can only start after washing machines have finished. Finally, most existing methods schedule devices at every half-hour or hour period, which provides an unrealistically limited flexibility for demand shifting and thus limited potential for peak demand reduction.

We therefore developed a new DN method based on the Frank-Wolfe algorithm, which can schedule devices for large numbers of households under RTP without requiring manual parameter tuning. Particularly, this method has low computation time and high scalability while allowing devices to be scheduled at smaller time slots under both the maximum demand constraint and the sequential constraint and minimising multiple conflicting objectives including the overall cost and discomfort, which has not been achieved by existing methods. The results show that this algorithm minimises the total cost and discomfort for 10,000 households in a second and has a constant computational complexity.

2 Related Work

Centralised methods schedule devices, against prices given 24 h ahead, optimally to minimise the cost and discomfort, while assuming complete knowledge of all households. These methods often employ a mixed integer linear programming (MILP) model or a linear programming (LP) model to schedule devices within single households [2, 4, 21, 28] or devices, *distributed generators* (DG's) (e.g. diesel generators, solar panels and small wind turbines) and electricity storage systems (or batteries) within a microgrid [14, 20].

While centralised scheduling methods operating across many consumers provide a way to optimise demand by minimising costs and discomfort, many argued that centralised methods are impractical for scheduling devices for many households [12, 13, 15, 23, 25, 29, 31, 33, 35]. First, these methods require knowledge of all devices and user preferences across all consumers in advance, which is impractical and can cause privacy concerns for some consumers. Second, centralised methods do not scale well. The size of the optimisation problem for multiple households is very large leading to very long computation times, because the computational complexity of MILP models increases exponentially with the problem size.

Alternatives such as distributed methods have been proposed to address the issues of practicality and scalability of centralised methods. Distributed methods allow consumers to schedule their devices independently without needing to expose their device details and preferences while remaining coordinated to ensure the cost and peak demand is reduced. Coordination is essential in distributed methods. A naive distributed alternative without coordination will lead to each household optimising their schedules against a common RTP. Thus peak demand can increase above normal levels as consumers schedule devices at the same cheapest times, causing the known *load synchronization* issue of implementing RTP for residential consumers [11, 18, 21, 23, 32]. Two types of distributed methods with coordination have been proposed to address this *load synchronization* issue: cooperative and non-cooperative methods.

Cooperative methods assume consumers are willing to communicate with each other about their aggregate demand profiles for the next 24 h. Based on the

demand profiles of all other consumers, each consumer will selfishly decide the optimal schedule for its devices to minimise its own costs. When all consumers iteratively communicate with others about their aggregate demand profiles and optimally and selfishly reschedule their devices based others' demand profiles, they will eventually reach a Nash equilibrium when no consumer can reschedule its devices to further reduce its cost. These methods are also known as cooperative game theoretic methods [9, 16, 21, 32]. However, some other research argue that cooperative methods will cause privacy issues and introduce large communication burden on communications networks as a result of the constant data exchange among consumers [13, 19, 34].

Similarly to cooperative methods, non-cooperative methods allow consumers to selfishly optimise their consumption schedules, however, the households are coordinated by a third-party entity, such as an utility company or distributed electricity service operator, instead of through iterative communication among consumers [6–8, 17, 19, 25, 27, 30, 33]. This third-party (referred to as 'utility' for the rest of this paper) collects demand profile forecasts from all consumers, calculates the total demand and provides feedback that may encourage consumers to reschedule their devices. This information often includes prices that reflect the actual cost of supplying electricity for the aggregate demand profile across all consumers in the scheme. Using these prices, consumers will reschedule their devices based on their consumption preferences and constraints to reduce their costs and report the updated demand back to the utility. Then, as in the case of cooperative methods, consumers and the utility will iteratively communicate with each other until no consumer can reschedule a device to receive a lower price from the utility. Although consumers are required to reveal their demand profiles in uncooperative methods, they reveal them only to the utility that already has access to their consumption data.

Many distributed and non-cooperative (DN) methods involve decomposing the device scheduling problems with unknown prices into sub-problems and a master problem that can be solved independently by consumers and the utility, respectively. The master problem solved by the utility calculates the total demand profile, the prices, the total cost and any other feedback information. The sub-problem solved by consumers is the scheduling of devices based on the prices and other feedback information given by the utility, which can be considered equivalent to the device scheduling problem in centralised methods.

Some DN methods focus on computing the optimal demand level for every time period, at a given resolution, over the next 24 h for each household and ignore the devices details [7, 8, 25, 33]. Some other DN methods include the devices by calculating the best start times for devices while considering consumption constraints and preferences for each device. [6] proposes an iterative method where each household use a greedy algorithm to schedule devices given prices from the utility. The utility then calculates the total cost of the total demand profile using a convex cost function and a penalty cost function. This penalty cost penalises undesired differences between the total demand profiles found at the previous and the current iterations, which is designed to prevent

the start times of devices oscillate between iterations and ensure the convergence of the algorithm. [17] proposes a gradient-based iterative algorithm where households optimally schedule devices at each iteration against prices given by the utility. They also consider various types of devices such as electric vehicles, heating and cooling devices, entertainment devices, shiftable devices and must-run devices. However, this research assume the power rates of devices can be varied continuously between minimum and maximum levels, which allows the iterative algorithm to improve the total demand profile with a fixed step size. [19] proposes a fast iterative method that first applies dual decomposition with Lagrange relaxation to the original problem, then a double smoothing techniques to the non-differentiable dual function to obtain a Lipschitz-continuous gradient for the dual function, and finally a fast gradient method to recover a near-optimal solution for the original problems in fewer iterations. It also includes a penalty cost for the changes between the total demand profiles found at the previous and current iterations. Their experiment results show that this fast gradient algorithm has very low computational time—requiring only 0.15 s for each iteration. However, this algorithm requires a fixed step size to ensure convergence. Moreover, devices are scheduled at hourly resolution only offering limited potential for peak demand reduction.

3 Models

Two types of participants are modelled in this research: households and a utility company. All households are served by the utility that purchases electricity from the wholesale market on behalf of their consumers and then sell electricity to consumers using a retail real-time price (RTP).

Devices are scheduled at the beginning of each ten-minute interval and the RTP is calculated based on the average demand across a thirty-minute period. The half-hour pricing resolution is designed to reflect the wholesale electricity price in Australia to date. The length of a scheduling interval is shorter than the typical length in the existing research, providing more flexibility for demand shifting and more potential for demand reduction. Let us write $T^s = 144$ as the total number of scheduling intervals per day, s as the index of a scheduling interval, $T^p = 48$ as the total number of pricing periods per day and t as the index of a pricing period.

Each household contains two types of devices: must-run and shiftable devices. Must-run devices must be turned on when required, such as lights and fridges. Shiftable devices can be used at different times of a day without significant negative effect on comfort and they can be further divided into interruptible and non-interruptible categories. For example, dish washers and washing machines are *non-interruptible and shiftable* devices that can be used in the middle of the night instead of evening and they are usually not interrupted once turned on, heating and cooling are *interruptible and shiftable* devices that can be suspended and resumed at a later time.

This research models shiftable devices with eight attributes: (1) an earliest start time (EST), (2) a latest finish time (LFT), (3) a preferred start time (PST),

(4) a care factor (CF), (5) a duration, (6) a power rate, (7) predecessors and (8) a maximum succeeding delay (MSD). Each device must start after its EST and finish before the LFT. If the device does not start at its PST, a discomfort cost will incur weighted by its CF. This CF can be one which means full flexibility is allowed for rescheduling this device, ten which means very little flexibility is allowed or anything between one and ten. The duration of any device is assumed to be a whole number of scheduling intervals. Each device may have predecessors that must finish running before this device and this device may have a MSD, which limits the time between the finish of its predecessors and the start of this device. Not all devices have predecessors or a MSD. Precedences and MSDs are used to model dependency or interruptibility of devices. For example, an interruptible device can be considered as multiple small non-interruptible devices that must be operated in a certain sequence, or a device (e.g. clothes dryer) must be turned on after another device (e.g. washing machine). For simplicity, this research does not consider different operating modes for each device or thermal loads such heating and cooling devices, therefore the power rate of each device is a constant. However, the method proposed in this research does allow incorporating these two features without changing the algorithms.

This research uses a step function to set the price for each pricing period based on the total demand of all consumers in that period. This step function is strictly increasing, which reflects the wholesale electricity price that indeed increases in steps with the number of and types of electricity generators required in each pricing period. This pricing step function include several demand thresholds and a price level for each demand threshold. A demand threshold can be interpreted as the total supply capacity of all required electricity generators whose unit costs of providing electricity are lower than the associated price level.

4 Fast, Scalable, Distributed and Noncooperative Device Scheduling Method

This fast, scalable, distributed and noncooperative device scheduling (FSDN-DS) method schedules devices of large numbers of households to minimise the expected total cost and discomfort cost under RTP prices while flattening the expected total peak demand. These RTP prices are calculated based on the total demand in real time. This device scheduling problem for multiple households under RTP (DSP-MH-RTP) considers multiple global constraints for each household, such as the maximum demand constraint and sequential constraint and discrete power rate for each device.

This FSDN-DS method is designed based on Frank-Wolfe (FW) algorithm that solves optimisation problems by solving its linear approximation problem iteratively, as described in [10, 26]. We adopt FW for three reasons: it is fast, for linearly constrained convex problems (such as DSP-MH-RTP) it converges to the global optimum, and does not require tuning of step-sizes.

This section will firstly explain the iterative communication process in the FSDN-DS method, secondly the device scheduling problem for households (DSP-H) and the pricing problem (PP) solved at each iteration as part of the FW algorithm, and lastly the method for creating the probability distribution.

4.1 Iterative Communication Process and Frank-Wolfe

Based on the FW algorithm, FSDN-DS finds the actual schedules for devices through an iterative communication process between households and the utility in an uncooperative manner described as Algorithm 1 and Fig. 1. At each iteration, FW calculates a tentative solution and a step size that will best improve the objective value of the tentative solution. When this step size becomes zero or very small (e.g. less than 0.001) at any iteration, it means the objective value cannot be improved any further, therefore the optimal solution is found and the iterative process is converged.

FSDN-DS interprets the step size at each iteration as the probability for the tentative solution found at that iteration to be the optimal solution, and the optimal solution as the solution that optimises the expected objective value. In our research, this optimal solution of FW is the expected total demand profile that minimises the expected total cost and discomfort. When the FW algorithm converges, FSDN-DS uses all step sizes to compute a probability distribution, which is then used by each household to select one from all the tentative solutions as the actual solution (the actual device schedule). The resulting actual total demand profile will be very close to the optimal expected total demand profile.

Algorithm 1. Iterative Communication Process

- 1: initialisation:
 - (a) each household schedules each device at its PST and reports its initial demand profile $\mathbf{I}_h^{(0)}$ to the utility company.
 - (b) the utility calculates the total demand profile $\bar{\mathbf{I}}^{(0)}$ and updates the price profile $\mathbf{p}^{(0)}$, and then returns $\mathbf{p}^{(0)}$ to each household.
 - 2: at each iteration $k > 0$:
 - (a) each household solves a device scheduling problem and reports its new demand profile $\mathbf{I}_h^{(k)}$ and the discomfort cost $U_h^{(k)}$ to the utility company.
 - (b) the utility calculates the total demand profile $\bar{\mathbf{I}}^{(k)}$ and total discomfort cost $\bar{U}^{(k)}$, then solves a pricing problem and returns a new price profile $\mathbf{p}^{(k)}$ and a new step size $\alpha^{(k)}$ to each household.
 - (c) **go to** (a) if $\alpha^{(k)} > \epsilon$ where ϵ is a very small number.
 - 3: each household will use $\{\alpha^{(k)} \mid k > 0\}$ to calculate the optimal probability distribution for selecting the actual device schedule.
-

This design of FSDN-DS is inspired by the *averaging* algorithm, which is the traditional mechanism for achieving convergence [5]. For each iteration k , the *averaging* algorithm calculates the total demand per period not simply from the

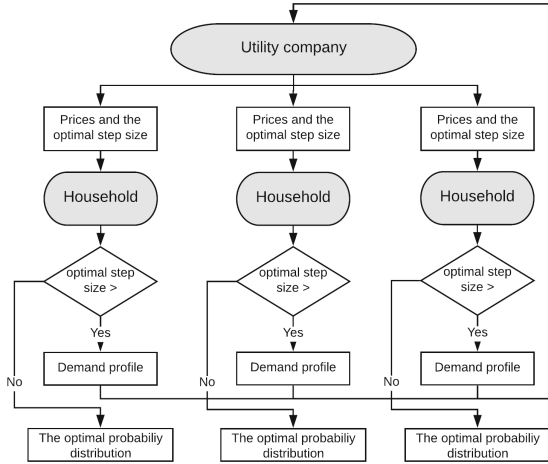


Fig. 1. Iterative communication process

device schedules of households at this iteration, but an average of all demands yielded by the device schedules at iteration 1 up to k . The utility finds the new prices based on the average total demand per period and send them to households. The averaging method can be a mathematical manipulation that, at each iteration, can be understood as moving partial demands of the devices, or moving the devices with a certain probability (assuming the number of similar devices is large). However, this method converges slowly, and is dependent on the maximum electricity price level, which could be very high if a punitive price is imposed above a grid-imposed maximum total demand. For this reason, we seek for another method that converges faster. Frank-Wolfe was chosen for its fast convergence, low computational complexity and automatic tuning of step-sizes.

4.2 Device Scheduling Problem for Households

The device scheduling problem for households (DSP-H) schedules devices against the price profile given by the utility to minimise the cost and discomfort of a household and the consumption constraints and preferences given by the consumer. Solving this problem is understood as solving the linear approximation of the original problem at each iteration of the FW algorithm. Once a new device schedule is found, each household send its demand profile to the utility.

Problem Formulation. Similarly to [9], this DSP-H considers both local constraints (constraints that only involve one device) and global constraints (constraints that involve multiple devices). The local constraint in this DSP-H is the start time constraint. The global constraints include the maximum demand constraint and the sequential constraint. The input parameters, decision variables, constraints and objective functions are written as the following:

Input Parameters - Let us write H as the total number of households, h as the index of one household, D_h as the total number of devices in household h , d as the index of one device, $s_{h,d}^e \in [1, 144]$ as the EST of device d in household h , $s_{h,d}^l \in [1, 144]$ as the LFT, $s_{h,d}^p \in [1, 144]$ as the PST, $e_{h,d} \in \mathbf{R}^+$ as the power rate when a device is turned on, $\eta_{h,d} \in [1, 10]$ as the CF, $\theta_{h,d} \in [1, 144]$ as the duration, $\mathbf{D}_{h,d}^p$ is the indices of the preprocessing devices and $\bar{\theta}_{h,d}$ as the MSD. Note that $\mathbf{D}_{h,d}^p = \emptyset$ if the device has no predecessor.

Decision Variables and Constraints - Let us write $s_{h,d}^a$ as the decision variables—the optimal start time for the device d in the household h . The start time constraints of devices can be written as the following:

$$s_{h,d}^e \leq s_{h,d}^a \leq s_{h,d}^l + 1 - \theta_{h,d} \tag{1}$$

If this device has predecessors, the sequential constraint can be written as the following:

$$\forall \hat{d} \in \mathbf{D}_{h,d}^p : s_{h,\hat{d}}^a + \theta_{h,\hat{d}} \leq s_{h,d}^a \leq s_{h,\hat{d}}^a + \theta_{h,\hat{d}} + \bar{\theta}_{h,d} \tag{2}$$

Let us write the maximum demand of a household h as E_h . The maximum demand constraint can be written as the following:

$$\forall s \in [1, T^s] : \sum \{e_{h,d} \mid d \in [1, D_h], s_{h,d}^a \leq s < s_{h,d}^a + \theta_{h,d}\} \leq E_h \tag{3}$$

Objectives - Costs and discomfort are two conflicting objectives considered in this DSP-H. This DSP-H aims to schedule devices in a way that best balances the needs for savings and comfort. Let us write C_h as the cost of the household h , p_s as the price of the scheduling interval s and this cost function as the following:

$$C_h = \sum_{s=1}^{144} \sum \{p_s \times e_{h,d} \mid d \in [1, D_h], s_{h,d}^a \leq s < s_{h,d}^a + \theta_{h,d}\} \tag{4}$$

The discomfort is an additional cost that penalizes the difference between the actual start time (AST) and the PST of each devices weighted by its CF. This cost, denoted as U_h , is a linear function designed as the following:

$$U_h = \sum_{d=1}^{D_h} |s_{h,d}^p - s_{h,d}^a| \times \eta_{h,d} \tag{5}$$

This DSP-H can be written as the following:

$$\begin{aligned} & \text{minimise } f_h = C_h + U_h \\ & \text{subject to } (1), (2), (3) \end{aligned} \tag{6}$$

Note that the objective function for each household is linear in the amount of energy required at each period, since the price per period is fixed. However, the overall cost of all consumers is discrete convex, because the overall cost is calculated the total demand profile and the price profile and the price profile is

calculated form a strictly increasing step function. The overall objective function is therefore piece-wise linear or discrete convex. Plus the constraints are all linear. These conditions allow us to solve the DSP-MH-RTP with the FW algorithm. This DSP-H can be considered as the linear approximation of the original problem at each iteration of FW.

Solution Methods. Solving this DSP-H includes two steps: a pre-processing algorithm and a scheduling algorithm. Two versions of the household scheduling algorithm have been proposed: an *Optimal Scheduling* algorithm and a *Optimistic Greedy Search* algorithm (OGS). The implementation of these versions can be found at <https://tinyurl.com/yak8anmm> and <https://tinyurl.com/ydy5s6s2>, respectively.

Pre-processing - This algorithm pre-computes the objective value of scheduling each device at each scheduling interval. Particularly, the objective value is increased to a big number if the scheduling interval violates the start time constraint. This way, the start time constraint is ensured implicitly by this algorithm without needing to be incorporated in the scheduling algorithm. The outcome of this algorithm is a matrix of precomputed objective values, written as $\mathbf{obj} = \{obj_{h,d,s} \mid d \in [1, D_h] \text{ and } s \in [1, 144]\}$.

Optimal Scheduling Algorithm - This is an optimisation model that finds the best start time for each device that has the smallest objective value and satisfies all constraints. This is a mixed integer linear programming (MILP) model, described as the following:

$$\begin{aligned}
 &\text{minimise} && \sum_{d \in [1, D_h], s \in [1, 144]} obj_{h,d,s} \cdot x_{d,s} \\
 &\text{s.t.} && \sum_{s \in [1, 144]} x_{d,s} = 1 && \forall d \in [1, D_h] \\
 &&& \sum_{d \in [1, D_h]} e_{h,d} \cdot x_{d,s} \leq E_h && \forall s \in [1, 144] \\
 &&& s_d = \sum_{s \in [1, 144]} s \cdot x_{s,d} && \forall d \in [1, D_h] \\
 &&& s_{\hat{d}} + \theta_{h,\hat{d}} \leq s_d \leq s_{h,\hat{d}} + \theta_{h,\hat{d}} + \bar{\theta}_{h,d} && \forall d \in D, \hat{d} \in D_{h,d}^p
 \end{aligned}$$

Optimistic Greedy Search Algorithm - this is a heuristic algorithm that searches for a good start time for each device in sequence. First, this algorithm schedules a device at the interval that has the smallest objective value and satisfies the sequential constraints if predecessors exist. Second, the algorithm checks if the maximum demand constraint is violated. If yes, the algorithm will re-schedule this device to the next cheapest interval that satisfies all constraints or satisfies the sequential constraint but violates the maximum demand constraint the least.

4.3 Pricing Problem

This pricing problem (PP) calculates the optimal step size that best reduces the total cost and discomfort of all households, which is equivalent to calculating the optimal step size of FW at each iteration.

Problem Formulation. First, PP calculates the total demand profile from the demand profiles of all households. This total demand profile is the new tentative solution at the current iteration. Second, PP calculates the optimal step size using the current tentative solution and the improved tentative solution from the previous iteration. Third, this step size is applied to improve the current tentative solution. The resulting improved tentative solution is then used to calculate a new price profile for the next iteration and a new step size in the next iteration.

Input Parameters - Let us write $\bar{\mathbf{I}}^{(k)}$ (which is the total demand profile of all households) as the tentative solution at the iteration k , $\bar{l}_t^{(k)}$ as the tentative total demand at period t , $\mathbf{x}^{(k)}$ as the improved tentative solution, $x_t^{(k)}$ as the improved tentative total demand at period t , $\bar{U}^{(k)}$ as the total discomfort of $\bar{\mathbf{I}}^{(k)}$ and $\hat{U}^{(k)}$ as the total discomfort of $\mathbf{x}^{(k)}$. Note that each pricing period has three scheduling intervals. The demand of one pricing period is the average demand of all scheduling intervals in that period.

Decision Variables and Objectives - The step size $\alpha^{(k)} \in [0, 1]$ is the decision variable. The objective is to minimise the total cost and discomfort given $\mathbf{x}^{(k-1)}$ and $\bar{\mathbf{I}}^{(k)}$. Let us write $g(\cdot)$ as this objective function. Based on the FW algorithm, PP first calculates a descent direction $\mathbf{d}^{(k)}$ as the following:

$$\mathbf{d}^{(k)} = \bar{\mathbf{I}}^{(k)} - \mathbf{x}^{(k-1)} \tag{7}$$

Then it finds the optimal step size by solving the following optimisation problem:

$$\begin{aligned} \alpha^{(k)} &= \operatorname{argmin}_{\alpha \in [0,1]} g(\mathbf{x}^{(k-1)} + \alpha \mathbf{d}^{(k)}) \\ &= \operatorname{argmin}_{\alpha \in [0,1]} (\mathbf{x}^{(k-1)} + \alpha(\bar{\mathbf{I}}^{(k)} - \mathbf{x}^{(k-1)})) \cdot \mathbf{p}^{(k)}(\mathbf{x}^{(k-1)} + \alpha(\bar{\mathbf{I}}^{(k)} - \mathbf{x}^{(k-1)})) \\ &\quad + \hat{U}^{(k)} + \alpha(\bar{U}^{(k)} - \hat{U}^{(k)}) \end{aligned} \tag{8}$$

This problem is equivalent to finding $\alpha^{(k)}$ at which the gradient of $g(\mathbf{x}^{(k-1)} + \alpha \mathbf{d}^{(k)})$ reaches zero, described as the following:

$$\begin{aligned} h(\alpha) &= \nabla g(\mathbf{x}^{(k-1)} + \alpha \mathbf{d}^{(k)}) \\ &= (\bar{\mathbf{I}}^{(k)} - \mathbf{x}^{(k-1)}) \cdot \mathbf{p}^{(k)}(\mathbf{x}^{(k-1)} + \alpha(\bar{\mathbf{I}}^{(k)} - \mathbf{x}^{(k-1)})) + (\bar{U}^{(k)} - \hat{U}^{(k)}) \\ &= 0 \end{aligned} \tag{9}$$

However, the cost function in this research is piece-wise linear, therefore the gradient changes in steps instead of continuously and it may turn from negative

to positive without reaching zero. The optimal $\alpha^{(k)}$ is instead found at which the gradient is before turning positive. This optimal $\alpha^{(k)}$ is then used for improving the current tentative solution as the following:

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + \alpha^{(k)}(\bar{\mathbf{I}}^{(k)} - \mathbf{x}^{(k-1)}) \tag{10}$$

At last, a new price profile $\mathbf{P}^{(k)}$ is calculated from the $\mathbf{x}^{(k)}$ using the pricing step function. $\mathbf{P}^{(k)}$ and $\alpha^{(k)}$ are then returned to each household.

Solution Method. The solving method for PP involves two steps: calculating (1) the price profile from a demand profile and (2) $\alpha^{(k)}$. Calculating the price requires simply comparing the total demand of each pricing period with the demand thresholds. The price level associated with the smallest demand threshold that is above the total demand of a period is the price for that period. Calculating $\alpha^{(k)}$ requires an iterative algorithm that involves three main steps: (1) for each pricing period, move the total demand to the next higher demand threshold if $(\bar{I}_t^{(k)} - x_t^{(k-1)}) > 0$ or to the next lower demand threshold otherwise, (2) compute the fractions of demand movements in each period, (3) move the demands in all periods by the smallest fraction. This iterative process continues until the gradient $h(\alpha)$ become positive. The implementation of this method can be found at <https://tinyurl.com/y7ek738o>.

4.4 Schedule Probability Distribution

Once the iterative communication process converges, each household uses the $\alpha^{(k)}$ computed at each iteration to construct the optimal probability distribution. Let us write \mathbf{pd} as the probability distribution, K as the number of iteration where the iterative process converges at and pd_k is the probability of choosing the schedule calculated at iteration k . The calculation of \mathbf{pd} is as the following:

$$\begin{aligned} \mathbf{pd} &= \{pd_k \mid k \in [1, K]\} \\ pd_k &= \begin{cases} \prod_{i=1}^K (1 - \alpha^{(i)}), & \text{if } k = 1 \\ \alpha^{(k)} \prod_{i=k+1}^K (1 - \alpha^{(i)}), & \text{if } 1 < k < K \\ \alpha^{(K)}, & \text{if } k = K \end{cases} \end{aligned} \tag{11}$$

pd_k is understood as the probability for the device schedule found by the household at the k^{th} iteration to be the optimal schedule. Households use this probability distribution to select one schedule from those computed at all iterations to be the final schedule. When large numbers of households schedule devices in this way, the resulting total demand profile of all households will be very close to the expected optimal total demand profile [29] found by the FW algorithm.

5 Experiments

Experiment Data and Environment. This research requires preference data, such as EST, PST, LST and CF for each device, which does not yet exist. Existing research that considers preference data generates synthetic data based on assumptions of the real-world and real device data [19, 29]. This research adopts the same method and generated experiment data in the following steps: (1) derive a probability distribution from a typical total demand profile in summer of Queensland, Australia (2) this probability distribution was used to sample PSTs for devices, (3) the Rayleigh distribution was used to sample durations for devices, (4) a list of commonly used devices, created by Ausgrid in Australia [3], was used to sample the power rates, (5) the ESTs and the LFTs were randomly selected with a uniform distribution, (6) the CFs were randomly chosen between 1 and 10 with a uniform distribution, (7) a 0 or 1 was selected from a uniform distribution to decide if a device had predecessors. If yes, the maximum succeeding delay would then be randomly selected from 1 to 144 with an uniform distribution. For simplicity, the experiments assume the maximum demand limit of each household is set to the sum of all devices in that household. Note that each device was allowed to start yesterday or finish tomorrow. When calculating the demand profile, the demand occurred yesterday would be added to the end of today and the demand occurred tomorrow would be added to the start of today. This method was used to ensure the resulting demand profile was realistic, otherwise very little demand would occur at the beginning and the end of the day. The device data generated by this method yield a total demand profile that matches those published in the Australian Electricity Market Operator website before optimisation.

The pricing step function was derived from a supply curve based on a years historical data from the Australian National Energy Market where we used the average relationship between the wholesale market spot price and the total supply. This is the price that changes every half hour, and is paid by all electricity retailers for all the bulk purchases they make on behalf of their retail customers. It is also the price that all power station operators earn for power generated in the same half hour.

The optimisation model for solving DSP-H was implemented in the Python interface for Gurobi. The rest of the FSDN-DS method was implemented in Python and tested on a computer with a CORE i5 4690 quad core.

5.1 Results

Peak and Cost Reductions and Run Times. These experiments show the results of two versions of the FSDN-DS method: one uses the MILP model for solving the DSP-H of households and the other uses the OGS algorithm. 60 problem instances were generated for these experiments. The number of households ranged from 2,000 to 10,000, with each number of households had 12 instances. The results for each number of households was averaged of its 12 instances. Table 1 show the average iterations for the FSDN-DS method to converge, the

Table 1. Experiment results

Num of houses	Method version	Average iterations	Average run time (second)	Average peak reduction	Average cost reduction
2000	MILP	12.25	0.21	17%	56%
	OGS	12.00	0.01	17%	55%
4000	MILP	13.83	0.24	17%	60%
	OGS	14.58	0.01	17%	59%
6000	MILP	16.58	0.30	18%	60%
	OGS	15.92	0.01	17%	59%
8000	MILP	17.75	0.32	18%	61%
	OGS	17.25	0.01	17%	60%
10000	MILP	17.00	0.30	18%	62%
	OGS	16.83	0.01	17%	60%

average run time (assuming all households run in parallel), the average peak reduction and the average cost reduction. The run time corresponded to one run for a household of all iterations. The results show that on average the FSDN-DS method, regardless of which version, converges in 20 iterations under a second. The average run time and iteration increases slightly from 2,000 to 6,000 households but remains almost the same from 6,000 to 10,000 households. The average peak reduction is almost the same for any number of households. The peak reduction rates of both version are very close, however, the MILP version outperformed the OGS version by 1%. The cost reduction rates increases slightly with the number of devices. The cost reduction rates of both versions are again very close, however, the MILP version outperformed the OGS version by 1–2%.

Sampling Actual Schedules. These experiments show the actual total demand profiles of all households that are calculated from the actual device schedules chosen using the optimal probability distribution. Two optimal probability distributions were calculated for these experiments. One was calculated from the results of the FSDN-DS method with the OGS version and the other is from the results of the MILP version. Figure 2 shows the actual total demand profiles generated by sampling actual device schedules from these two probability distributions for ten times, and the expected total demand profiles calculated by both versions. The problem instance used for generating these results had 10000 households.

5.2 Analysis

The results show that the FSDN-DS method proposed in this research indeed has fast convergence, low computational time and high scalability, making it suitable for scheduling devices in real time. On average, this method converges

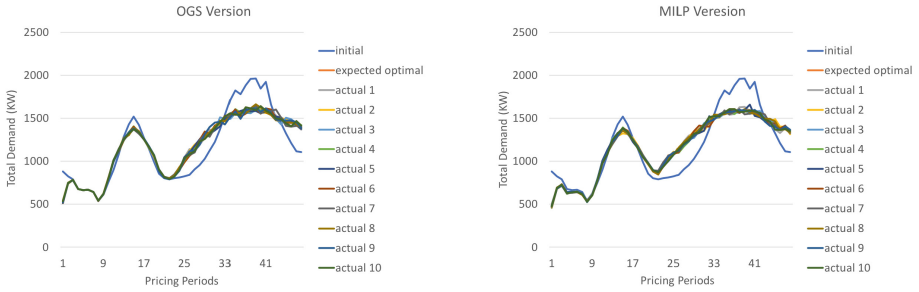


Fig. 2. Actual and optimal expected total demand profiles

in 20 iterations under a second for any number of households. Although the MILP version of the FSDN-DS method outperforms the OGS version by 1–2%, the average run time of the OGS version is more than 20 times faster than the MILP version. Moreover, the implementation of the OGS version requires no optimisation solvers, making it easier to be implemented in practice than the MILP version.

6 Conclusions

This paper describes the solution of a device scheduling problem for multiple households under RTP (DSP-MH-RTP) where the total cost and discomfort of all households are minimised. In particular, we focus on the case where the prices considered are not known in advance but are a function of the total demand. Moreover, multiple global constraints such as the maximum demand constraint and the sequential constraint are considered in this problem. Additionally, this research considers each device to be scheduled at every ten-minute interval, which provide more flexibility and potential for peak demand reduction than the thirty-minute or one-hour interval that is commonly used in the literature.

To solve the problem we develop a fast, scalable, distributed and non-cooperative device scheduling method (FSDN-DS) based on Frank-Wolfe Algorithm. This method finds the optimal expected total demand profile that minimises the expected total cost and discomfort through an iterative communication process between households and the utility company. Particularly, this method has low computation time and high scalability while allowing devices to be scheduled at smaller time slots under the maximum demand constraint and the sequential constraint, which has not been achieved by existing methods.

Two versions of the FSDN-DS method have been presented in this paper: FSDN-DS with a mixed-integer programming model version and FSDN-DS with an optimistic greedy search algorithm version. The experiment results show that both versions have fast convergence, low computational time and high scalability, making them suitable for scheduling devices in real time. Although the results of the MILP version outperforms the OGS version by 1–2%, the run time of the

OGS version is 20 times faster than the MILP version. Moreover, the implementation of the OGS requires only a programming language with no optimisation solvers, making it easier and more practical for individual households than the MILP version.

The full implementation of the FSDN-DS method and a website for visualising full experiment results can be found at <https://bitbucket.org/monash-dr/deterministic-rtp-ad/src/master/>.

References

1. Albadi, M.H., El-Saadany, E.F.: Demand response in electricity markets: an overview, June 2007
2. Anvari-Moghaddam, A., Monsef, H., Rahimi-Kian, A.: Optimal smart home energy management considering energy saving and a comfortable lifestyle. *IEEE Trans. Smart Grid* **6**(1), 324–332 (2015)
3. Ausgrid: Appliance usage guide. http://www.ausgrid.com.au/Common/Customer-Services/Homes/Energy-efficiency/Energy-efficiency-at-home-tips/Energy-usage-calculators/~/_media/Files/Custom
4. Barbato, A., Capone, A., Carello, G., Delfanti, M., Merlo, M., Zaminga, A.: House energy demand optimization in single and multi-user scenarios. In: 2011 IEEE International Conference on Smart Grid Communications (SmartGridComm), pp. 345–350, October 2011
5. Chapman, A.C., Rogers, A., Jennings, N.R., Leslie, D.S.: A unifying framework for iterative approximate best-response algorithms for distributed constraint optimization problems. *Knowl. Eng. Rev.* **26**(4), 411–444 (2011)
6. Chavali, P., Yang, P., Nehorai, A.: A distributed algorithm of appliance scheduling for home energy management system. *IEEE Trans. Smart Grid* **5**(1), 282–290 (2014)
7. Chen, L., Li, N., Low, S.H., Doyle, J.C.: Two market models for demand response in power networks. In: 2010 First IEEE International Conference on Smart Grid Communications (SmartGridComm), pp. 397–402, October 2010
8. Fan, Z.: Distributed demand response and user adaptation in smart grids. In: 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops, pp. 726–729, May 2011
9. Fioretto, F., Yeoh, W., Pontelli, E.: A multiagent system approach to scheduling devices in smart homes. In: Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, pp. 981–989. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2017). <http://dl.acm.org/citation.cfm?id=3091125.3091265>
10. Frank, M., Wolfe, P.: An algorithm for quadratic programming. *Naval Res. Logist. Q.* **3**(1–2), 95–110 (1956)
11. Goudarzi, H., Hatami, S., Pedram, M.: Demand-side load scheduling incentivized by dynamic energy prices. In: 2011 IEEE International Conference on Smart Grid Communications (SmartGridComm), pp. 351–356, October 2011
12. He, S., Liebman, A., Rendl, A., Wallace, M., Wilson, C.: Modelling RTP-based residential load scheduling for demand response in smart grids. In: Ansoetegi, C. (ed.) Proceedings of the Thirteenth International Workshop on Constraint Modelling and Reformulation (ModRef 2014), pp. 36–51. Universitat de Lleida (2014)

13. Joe-Wong, C., Sen, S., Ha, S., Chiang, M.: Optimized day-ahead pricing for smart grids with device-specific scheduling flexibility. *IEEE J. Sel. Areas Commun.* **30**(6), 1075–1085 (2012)
14. Kanchev, H., Lu, D., Colas, F., Lazarov, V., Francois, B.: Energy management and operational planning of a microgrid with a PV-based active generator for smart grid applications. *IEEE Trans. Ind. Electron.* **58**(10), 4583–4592 (2011)
15. Kim, S.J., Giannakis, G.B.: Scalable and robust demand response with mixed-integer constraints. *IEEE Trans. Smart Grid* **4**(4), 2089–2099 (2013)
16. Kuschel, C., Köstler, H., Rüde, U.: Multi-energy simulation of a smart grid with optimal local demand and supply management. *Smart Grid Renew. Energy* **06**(11), 303–315 (2015)
17. Li, N., Chen, L., Low, S.H.: Optimal demand response based on utility maximization in power networks. In: 2011 IEEE Power and Energy Society General Meeting, pp. 1–8, July 2011
18. Li, Y., Ng, B.L., Trayer, M., Liu, L.: Automated residential demand response: algorithmic implications of pricing models. *IEEE Trans. Smart Grid* **3**(4), 1712–1721 (2012)
19. Mhanna, S., Chapman, A.C., Verbic, G.: A fast distributed algorithm for large-scale demand response aggregation. CoRR abs/1603.00149 (2016). <http://arxiv.org/abs/1603.00149>
20. Mohamed, F.A., Koivo, H.N.: Microgrid online management and balancing using multiobjective optimization. In: 2007 IEEE Lausanne Power Tech, pp. 639–644, July 2007
21. Mohsenian-Rad, A.H., Leon-Garcia, A.: Optimal residential load control with price prediction in real-time electricity pricing environments. *IEEE Trans. Smart Grid* **1**(2), 120–133 (2010)
22. Mohsenian-Rad, A.H., Wong, V.W.S., Jatskevich, J., Schober, R., Leon-Garcia, A.: Autonomous demand-side management based on game-theoretic energy consumption scheduling for the future smart grid. *IEEE Trans. Smart Grid* **1**(3), 320–331 (2010)
23. Ramchurn, S.D., Vytelingum, P., Rogers, A., Jennings, N.: Agent-based control for decentralised demand side management in the smart grid. In: The 10th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2011, vol. 1, pp. 5–12. International Foundation for Autonomous Agents and Multiagent Systems, Richland (2011). <http://dl.acm.org/citation.cfm?id=2030470.2030472>
24. Ren, D., Li, H., Ji, Y.: Home energy management system for the residential load control based on the price prediction. In: 2011 IEEE Online Conference on Green Communications, pp. 1–6, September 2011
25. Samadi, P., Mohsenian-Rad, A.H., Schober, R., Wong, V.W.S., Jatskevich, J.: Optimal real-time pricing algorithm based on utility maximization for smart grid. In: 2010 First IEEE International Conference on Smart Grid Communications (SmartGridComm), pp. 415–420, October 2010
26. Sheffi, Y.: *Urban Transportation Networks: Equilibrium Analysis with Mathematical Programming Methods*. Prentice-Hall, Englewood Cliffs (1985)
27. Shi, W., Xie, X., Chu, C.C., Gadh, R.: Distributed optimal energy management in microgrids. *IEEE Trans. Smart Grid* **6**(3), 1137–1146 (2015)
28. Sou, K.C., Weimer, J., Sandberg, H., Johansson, K.H.: Scheduling smart home appliances using mixed integer linear programming. In: 2011 50th IEEE Conference on Decision and Control and European Control Conference, pp. 5144–5149, December 2011

29. Van Den Briel, M., Scott, P., Thiébaux, S.: Randomized load control: a simple distributed approach for scheduling smart appliances. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, IJCAI 2013, pp. 2915–2922. AAAI Press (2013). <http://dl.acm.org/citation.cfm?id=2540128.2540548>
30. Veit, A., Xu, Y., Zheng, R., Chakraborty, N., Sycara, K.: Demand side energy management via multiagent coordination in consumer cooperatives. *J. Artif. Int. Res.* **50**(1), 885–922, May 2014. <http://dl.acm.org/citation.cfm?id=2693068.2693091>
31. Voice, T.D., Vytelingum, P., Ramchurn, S.D., Rogers, A., Jennings, N.R.: Decentralised control of micro-storage in the smart grid. In: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, pp. 1421–1427. AAAI Press (2011). <http://dl.acm.org/citation.cfm?id=2900423.2900648>
32. Vytelingum, P., Voice, T.D., Ramchurn, S.D., Rogers, A., Jennings, N.R.: Agent-based micro-storage management for the smart grid. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1, AAMAS 2010, vol. 1, pp. 39–46. International Foundation for Autonomous Agents and Multiagent Systems, Richland (2010). <http://dl.acm.org/citation.cfm?id=1838206.1838212>
33. Wang, Y., Mao, S., Nelms, R.M.: Distributed online algorithm for optimal real-time energy distribution in the smart grid. *IEEE Internet Things J.* **1**(1), 70–80 (2014)
34. Yu, R., Yang, W., Rahardja, S.: Optimal real-time price based on a statistical demand elasticity model of electricity. In: 2011 IEEE First International Workshop on Smart Grid Modeling and Simulation (SGMS), pp. 90–95, October 2011
35. Zhang, W., Xu, Y., Liu, W., Zang, C., Yu, H.: Distributed online optimal energy management for smart grids. *IEEE Trans. Ind. Inform.* **11**(3), 717–727 (2015)

Multiagent and Parallel CP Track



Balancing Asymmetry in Max-sum Using Split Constraint Factor Graphs

Liel Cohen and Roie Zivan^(✉)

Ben Gurion University of the Negev, Beer Sheva, Israel
{lielc,zivanr}@post.bgu.ac.il

Abstract. Max-sum is a version of Belief Propagation, used for solving DCOPs. On tree-structured problems, Max-sum converges to the optimal solution in linear time. When the constraint graph representing the problem includes multiple cycles, Max-sum might not converge and explore low quality solutions. Damping is a method that increases the chances that Max-sum will converge. Damped Max-sum (DMS) was recently found to produce high quality solutions for DCOP when combined with an anytime framework.

We propose a novel method for adjusting the level of asymmetry in the factor graph, in order to achieve a balance between exploitation and exploration, when using Max-sum for solving DCOPs. By converting a standard factor graph to an equivalent split constraint factor graph (SCFG), in which each function-node is split to two function-nodes, we can control the level of asymmetry for each constraint. Our empirical results demonstrate that by applying DMS to SCFGs with a minor level of asymmetry we can find high quality solutions in a small number of iterations, even without using an anytime framework. As part of our investigation of this success, we prove that for a factor-graph with a single constraint, if this constraint is split symmetrically, Max-sum applied to the resulting cycle is guaranteed to converge to the optimal solution and demonstrate that for an asymmetric split, convergence is not guaranteed.

1 Introduction

The Max-sum algorithm [4] is an incomplete inference (GDL-based) algorithm for solving Distributed Constraint Optimization Problems (DCOP), a general model for distributed problem solving that has a wide range of applications in multi-agent systems. Max-sum has drawn considerable attention in recent years, including being proposed for multi-agent applications such as sensor systems [20, 23] and task allocation for rescue teams in disaster areas [16]. Max-sum is actually a version of the well known Belief propagation algorithm [25], used for solving DCOPs. Agents in Max-sum propagate cost/utility information to all neighbors. This contrasts with other inference algorithms such as ADPOP [13], which only propagate costs up a pseudo-tree structure overlaid on the agents.

As is typical of inference algorithms, Max-sum is purely exploitive both in the computation of its beliefs and in its selection of values based on those beliefs.

Belief propagation in general (and Max-sum specifically) is known to converge to the optimal solution for problems whose constraint graph is acyclic. On problems with cycles, the agents' beliefs may fail to converge, and the resulting assignments that are considered optimal under those beliefs may be of low quality [4, 25, 30]. This occurs because cyclic information propagation leads to inaccurate and inconsistent information being computed by the agents. Unfortunately, many DCOPs that were investigated in previous studies are dense and indeed include multiple cycles (e.g., [6, 10]). However, in contrast to most DCOP algorithms, Max-sum was found to produce solutions with similar quality when applied to symmetric and asymmetric problems [31].

Damping is a method that decreases the effect of cyclic information propagation in Belief propagation by balancing the weight of the new calculation performed in each iteration and the weight of calculations performed in previous iterations. As a result it increases the chances for convergence [8, 14, 19, 21]. A recent investigation of the effect of damping on Max-sum when applied to DCOPs revealed that damping generates efficient exploration that, when combined with an anytime framework, produces high quality results [2]. However, without the anytime framework [29], a large number of iterations is required for damped Max-sum (DMS) to reach a high quality solution.

In this paper we contribute to the development of incomplete inference algorithms for solving DCOPs by proposing a novel degree of freedom for balancing between exploration and exploitation, in Max-sum. This degree of freedom, is the level of asymmetry in function-nodes representing constraints in the factor graph. The ability to control the level of asymmetry for each constraint is achieved by shifting a standard factor graph to an equivalent split constraint factor graph (SCFG), where each constraint is represented by two function-nodes instead of one¹.

Our empirical evaluation reveals that the level of asymmetry in SCFGs can determine the level of exploration that the algorithm performs. When combining damping with low levels of asymmetry, Max-sum converges very fast to high quality solutions, without the need of an anytime framework.

As part of our investigation of this success, we investigate the effect of a split on a single constraint factor graph. We prove that when the cost table of the single function-node is split symmetrically, Max-sum is guaranteed to converge on the resulting cycle to the optimal solution, regardless of the damping factor used, and demonstrate that this is not the case when the constraint is split asymmetrically.

2 Related Work

The first paper to propose the use of Belief propagation for solving DCOPs and named it *Max-sum* was [4]. This work was followed by a number of studies

¹ A similar factor graph was used in [31] for representing asymmetric DCOPs.

that addressed the in-convergence of the algorithm on graphs that include multiple cycles, including [17], which proposed Bounded Max-sum and [30] which proposed Max-sum_ADVP.

Max-sum was applied to sensor nets both with mobile and static sensors [5, 20, 26], to supply chain management [1] and teams of rescue agents [16]. A number of papers made an attempt to overcome its most apparent drawback, the exponential computation of the content of messages sent by function-nodes [9, 20]. For specific applications with cardinality constraints, the *Tractable High Order Potentials* (THOP) method [22], which was adjusted to DCOPs, and implemented by [15] can be used. It reduces the computation runtime of function-nodes to $O(K \log(K))$, where K is the number of neighboring variable-nodes of the calculating function-node.

Max-sum was applied to asymmetric DCOPs in [31], by having each agent involved in a constraint hold a function-node representing its personal costs for that constraint. Thus, for each binary constraint there were two representing function-nodes. In contrast to other DCOP algorithms, Max-sum versions were found to maintain the quality of solutions they produce when applied to asymmetric problems. The main difference from our work is that, while they have used more than one function-node for a single constraint in order to represent the given natural structure of an asymmetric problem, we initiate a split of a standard function-node to two function-nodes representing the same constraint as an algorithmic method.

The possibility to encourage convergence of Max-sum by splitting nodes in the factor-graph was first suggested in [18]. This study investigated the theoretical conditions for convergence of the algorithm when using constant even splits. It further proposes a sequential version of the algorithm which, under some conditions, is guaranteed to converge to a local optimum and specifies the conditions in which this algorithm converges to the global optimum. They mention that by using damping with damping factor of $\frac{1}{n}$, this algorithm converges in a distributed synchronous execution as well. The main difference from our work is that we investigate the use of splitting nodes to control the level of symmetry in the factor graph and to balance exploitation and exploration, and therefore we investigate theoretical and empirical implications of constant even and uneven, random and limited random splits.

Recently, the effect of the use of damping within Max-sum was investigated [2]. It was found to immensely improve the quality of solutions traversed by Max-sum, and when combined with an anytime framework [29], produce high quality solutions. However, the use of Max-sum within an anytime framework, requires agents to exchange their value assignments in each iteration. This is not a requirement of the algorithm and therefore, such an exchange reduces the privacy of the algorithm. When an anytime framework is not used, a large number of iterations is required for Max-sum to find solutions with low costs.

3 Background

In this section we present background on DCOPs, the Max-sum algorithm and the damping method.

3.1 Distributed Constraint Optimization

Without loss of generality, in the rest of this paper we will assume that all problems are minimization problems (as in many DCOP papers, e.g., [10]). Thus, we assume that all constraints define costs and not utilities.²

A *DCOP* is 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 (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 $d \in D_i$ to X_i by an ordered pair $\langle X_i, d \rangle$. \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 $R_{ij} : D_i \times D_j \rightarrow \mathbb{R}^+ \cup \{0\}$.³ A *binary DCOP* is a DCOP in which all constraints are binary. A *partial assignment* (PA) 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 PA, $\text{vars}(PA) = \{X_i \mid \exists d \in D_i \wedge \langle X_i, d \rangle \in PA\}$. A constraint $R_j \in \mathcal{R}$ of the form $R_j : D_{j_1} \times D_{j_2} \times \dots \times D_{j_k} \rightarrow \mathbb{R}^+ \cup \{0\}$ 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 ($\text{vars}(PA) = \mathcal{X}$). 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, in which all constraints are binary. These assumptions are customary in DCOP literature (e.g., [12, 27]). In our description of algorithms and their properties, we will assume that there are no ties, i.e., that each entry in the constraint tables held by function-nodes has a unique numeric value. This can easily be achieved by using a method similar to the one proposed in [3], which we use in our empirical study.⁴

² The inference algorithm for minimization problems is actually *Min-sum*. However, we will continue to refer to it as *Max-sum* since this name is widely accepted.

³ We say that a variable is *involved* in a constraint if it is one of the variables the constraint refers to.

⁴ For an example of the need to break ties in the factor-graph see [30].

3.2 The Max-Sum Algorithm

⁵Max-sum operates on a *factor-graph*, which is a bipartite graph in which the nodes represent variables and constraints [7]. Each variable-node representing a variable of the original DCOP is connected to all function-nodes that represent constraints, which it is involved in. Similarly, a function-node is connected to all variable-nodes that represent variables in the original DCOP that are involved in the constraint it represents. Variable-nodes and function-nodes are considered “agents” in Max-sum, i.e., they can send and receive messages, and 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 . 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:

$$Q_{x \rightarrow f}^i = \sum_{f' \in F_x, f' \neq f} R_{f' \rightarrow x}^{i-1} - \alpha$$

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 costs included in the message (i.e., for each $d \in D_x$) in order to prevent the costs carried by messages throughout the algorithm run from growing arbitrarily.

A message sent from a function-node f to a variable-node x in iteration i includes for each value $d \in D_x$:

$$\min_{PA_{-x}} \text{cost}(\langle x, d \rangle, PA_{-x})$$

where PA_{-x} is a possible combination of value assignments to variables involved in f not including x . The term $\text{cost}(\langle x, d \rangle, PA_{-x})$ represents the cost of a partial assignment $a = \{\langle x, d \rangle, PA_{-x}\}$, which is: $f(a) + \sum_{x' \in X_f, x' \neq x, \langle x', d' \rangle \in a} Q_{x' \rightarrow f}^{i-1} \cdot d'$, 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 $Q_{x' \rightarrow f}^{i-1} \cdot d'$ is the cost that was received in the message sent from variable-node x' in iteration $i - 1$, for the value d' that is assigned to x' in a . x selects its value assignment $\hat{d} \in D_x$ following iteration k as follows:

$$\hat{d} = \arg \min_{d \in D_x} \sum_{f \in F_x} R_{f \rightarrow x}^k \cdot d$$

⁵ For lack of space we describe the algorithm briefly and refer the reader to more detailed descriptions in [4, 17, 30].

Introducing Damping into Max-Sum. 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. Denote by $\widehat{m}_{i \rightarrow j}^k$ the result of the calculation made by agent A_i of the content of a message intended to be sent from A_i to agent A_j in iteration k . Denote by $m_{i \rightarrow j}^{k-1}$ the message sent by A_i to A_j at iteration $k - 1$. The message sent from A_i to A_j in iteration k is calculated as follows:

$$m_{i \rightarrow j}^k = \lambda m_{i \rightarrow j}^{k-1} + (1 - \lambda) \widehat{m}_{i \rightarrow j}^k$$

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.

Applying Max-sum to Asymmetric Problems. 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. 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 [31] for details).

Exploration and Exploitation in Max-sum. In local search algorithms, an agent commonly selects an assignment that minimizes the cost according to the information available to it, i.e., it exploits the information. On the other hand, the agent can select an assignment that does not minimize (and even enlarges) its cost, hoping this will allow it to find assignments with lower cost in following iterations. Such actions, which do not result in immediate benefit are aka exploration. In distributed local search, even if each agent performs only exploitive actions, concurrent actions by a number of agents can generate exploration, e.g., in DSA when neighboring agents replace assignments concurrently.

In inference algorithms such as Max-sum agents do not propagate assignments. However, each variable-node can select an assignment at each iteration based on the costs it receives. Thus, as in the distributed stochastic algorithm

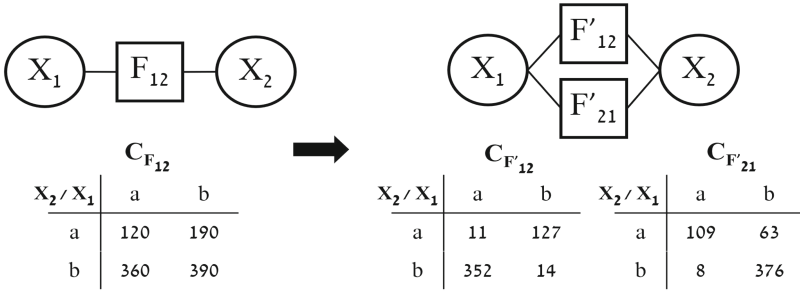


Fig. 1. An acyclic DCOP factor graph (on the left) and its equivalent SCFG (on the right).

(DSA) [28], a global view that examines the quality of the global assignment that can be inferred in each iteration, can reveal whether the agents are improving the global assignment or involuntarily, selecting assignments that are with higher costs and hence, exploring.

4 Split Constraint Factor Graphs

Damping can be used as a degree of freedom, to balance exploration and exploitation in Max-sum [2]. However, when using damping, thousands of iterations are required for the algorithm to find high quality solutions. By combining damped Max-sum (DMS) with an anytime framework [29], the number of iterations required in order to find high quality solutions is an order of magnitude smaller.

We aim at shortening the process of producing high quality solutions by Max-sum and eliminating the dependency on the anytime framework, which requires agents to share value assignments in contrast to the requirements of the algorithm. Thus, we propose an additional degree of freedom, the level of asymmetry of constraints in the factor graph. To this end, we propose the use of *Split Constraint Factor Graphs* (SCFGs) in which each constraint that was represented by a single function-node in the original factor graph, is represented by 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 we can determine the level of asymmetry in the SCFG.

Formally, an SCFG G' is equivalent to a factor graph G , if their sets of variable-nodes (and their domains) are equal, and if for each function-node F in G with cost table C_F , there exist function-nodes F' and F'' in G' , which are connected to the same variable-nodes as F , and their constraint tables satisfy $C_F = C_{F'} + C_{F''}$. Thus, each SCFG has a single equivalent DCOP factor graph to which its function-nodes' constraints sum up to, however, a standard DCOP factor graph can have countless equivalent SCFGs.

Returning to the example portrayed in Fig. 1, given a standard factor graph of a symmetric DCOP as presented on the left, the cost table of the function-node F_{12} in this factor graph is split using a different random ratio for each entry, thus generating a new equivalent SCFG (on the right) containing 2 function-nodes F'_{12} and F'_{21} .

We differentiate between constant SCFGs, in which constraint costs are split according to a predetermined constant ratio, and random SCFGs, in which each cost in each constraint table is split according to a randomly generated ratio.

It is important to notice the difference between the use we make of SCFGs and the use made of factor graphs with two function-nodes representing a constraint in [31]. There, they assume this type of factor graph is required to represent the asymmetric state of the world, while we assume the input problem is symmetric, and the generation of the SCFG is an algorithmic action (represented by the small black arrow between the factor graphs in Fig. 1).

5 Splitting a Single Constraint

In Sect. 6 we present empirical evidence of the success of applying DMS to symmetric SCFGs and SCFGs with minor asymmetry. As part of an attempt to explain this success we investigate the different effect of a symmetric split and an asymmetric split in a single constraint factor graph.⁶

Lemma 1. *On a factor graph with two variable-nodes X_1 and X_2 and two identical function-nodes, F_{12} and F_{21} , each connected to both variable-nodes, Max-sum is guaranteed to converge to the optimal solution after the first iteration.*

Proof: We prove by induction on the number of iterations. If the smallest cost c in the cost table held by both function-nodes⁷ is in entry i, j representing the cost when the value assignments selected are the i 'th value in the domain of X_1 and the j 'th value in the domain of X_2 , then in the first iteration of the algorithm, each function-node will send to X_1 a vector where the i 'th cost in it is c , and the messages sent to X_2 will include c in the j 'th cost of the vector. All other costs in these vector must be larger than c . Thus, following the first iteration values i and j are selected. The induction assumption is that in the k 'th iteration⁸, $k > 1$, the i 'th cost in the messages sent to X_1 and the j 'th cost in the messages sent to X_2 will be $\frac{k+1}{2}c$, while all other costs in these vectors will be larger. In the next iteration ($k+1$), X_1 and X_2 will send forward the message received from each function-node, to the other function-node. In iteration $k+2$ in the vector sent in each message to X_1 the i 'th cost will include a sum of the smallest cost in the vector received, which according to the induction assumption

⁶ Although Lemmas 1 and 2 can be implied from Lemma 3, for simplicity of presentation we enclose all three, provide complete proof for Lemma 1, and intuitive explanations how to generalize the proof so it will apply to Lemmas 2 and 3.

⁷ Recall that we assumed in Sect. 3.1 that there are no ties, so such a cost is unique.

⁸ Without loss of generality, we assume that k is odd. If it was even, then the assumption was that the cost is $\frac{k}{2}c$.

is $\frac{k+1}{2}c$, and the smallest cost in the cost table, c . Thus, the resulting cost, which must still be smallest in the vector, is $\frac{k+1}{2}c + c = \frac{k+3}{2}c$. For similar reasons, the smallest cost in the vectors sent to X_2 in iteration $k + 2$ are the j 'th costs and they are equal to $\frac{k+3}{2}c$. \square

Lemma 2. *On a factor graph with two variable-nodes X_1 and X_2 and two function-nodes, F_{12} and F_{21} , each connected to both variable-nodes. If for any real number m , the cost tables held by the two function-nodes maintain the relation $C_{F_{12}} = mC_{F_{21}}$ then Max-sum is guaranteed to converge to the optimal solution after the first iteration.*

The main difference is that in the first iteration, X_1 is sent one message with the i 'th cost equal to c and another where it is equal to mc and this is true for the j 'th costs in the messages sent to X_2 . In iteration k the i 'th cost in a message sent to X_1 and the j 'th cost in the messages sent to X_2 will include alternating summations of c and mc , while all other costs in the vectors will include corresponding summations of other (larger) numbers.

Lemma 3. *On a factor graph with two variable-nodes X_1 and X_2 and two function-nodes, F_{12} and F_{21} , each connected to both variable-nodes. If for any real number m , the cost tables held by the two function-nodes maintain the relation $C_{F_{12}} = mC_{F_{21}}$ then DMS is guaranteed to converge to the optimal solution after the first iteration, regardless of the damping factor being used.*

In each iteration the costs calculated are multiplied by $1 - \lambda$ and added to the previous message sent, multiplied by λ . Thus, in the first iteration the costs we mention in the proofs for the lemmas above will be multiplied by $1 - \lambda$, but the smallest entries will remain i for X_1 and j for X_2 . The same will be true for the $k + 2$ iteration in the induction step. The smallest entry will not change as a result of multiplying all messages by the same factor.

Proposition 1. *If DMS is applied to a constant SCFG generated from a factor graph including two variable-nodes and a single function-node representing the constraint among them, it will converge after the first iteration, regardless of the damping factor used.*

Proof: Immediate from Lemma 3.

It is important to notice that previous works on the behavior of Belief propagation on single cycle graphs, only prove the optimality of the solutions obtained when the algorithm converges [24]. Moreover, when damping is used (as in DMS), the algorithm might converge to sub-optimal solutions on single cycle graphs. Thus, the fact that on constant split cycles both Max-sum and DMS are guaranteed to converge after a single cycle to the optimal solution is novel and significant.

In contrast, when Max-sum is applied to an SCFG generated by a random split of a single constraint, function-nodes might choose in their calculations minimal costs, which do not correspond with identical value assignments, and further

calculations may be based on inconsistent assignment choices for the same variable, producing impossible belief costs for the variable-nodes. Hence, Max-sum does not necessarily converge to the optimal solution. We empirically observed this behavior in experiments on 20,000 randomly generated, single cycle factor graphs, in which Max-sum did not converge to the optimal solution in many problem instances. Interestingly, DMS always converged, but not necessarily to the optimal solution. For example, when applied to the *SCFG* portrayed in Fig. 1, standard Max-sum alternates endlessly between solutions of costs 120, 190 and 360. The optimal solution incurs a cost of 120. DMS, however, converges after 18 iterations to the suboptimal solution of cost 190.

Nevertheless, as demonstrated in Sect. 6, in *SCFGs* based on DCOPs containing multiple cycles, this pathology can induce exploration, which can be adjusted by determining the level of asymmetry of split constraints, and exploited by using a high damping factor.⁹

6 Experimental Evaluation

In order to investigate the advantages of the use of *SCFGs* when applying Max-sum to DCOPs, we present a set of experiments comparing standard Max-sum and DMS, both when applied to different *SCFGs* and two versions that guarantee convergence: Bounded Max-sum and Max-sum_ADVP (for detailed descriptions of these algorithms see [17, 30]). We also include in our experiments the results of the well known DSA algorithm (we use type C with $p = 0.7$ [28]), in order to give an insight on the quality of the results, in comparison with local search DCOP algorithms.

We evaluated the algorithms on random uniform minimization DCOPs and on structured and realistic problems, i.e., graph coloring, meeting scheduling and scale-free nets (see details below). At each experiment we randomly generated 50 different problem instances and ran the algorithms for 2,000 iterations on each of them. The results presented in the graphs are an average of those 50 runs. For each iteration we present the sum of costs of the constraints involved in the assignment that would have been selected by each algorithm at that iteration. The statistical significance of the results was verified using paired t-tests with significance level of $p = 0.05$. In order to maximize the benefit of the algorithms exploration property we implemented all algorithms within the anytime framework proposed in [29]. This allowed us to report for each of the algorithms the best result it traverses within 2,000 iterations. Also, in all versions of Max-Sum, we used personal value preferences, selected randomly for the purpose of tie breaking, as was suggested in [3].

All problems were formulated as minimization problems. The uniform random problems were generated by adding in each problem a constraint for each pair of agents (variables) with probability p_1 . For each constrained pair we set a

⁹ further insights on the relation between the success of our empirical results and the properties presented in this section are detailed in Sect. 6.2.

cost for each combination of value assignments, selected uniformly between 100 and 200.¹⁰ Each problem included 50 variables with 10 values in each domain.

Graph coloring problems include random constraint graph topologies and all constraints $R_{ij} \in \mathcal{R}$ are “not-equal” cost functions where an equal assignment of neighbors in the graph incurs a cost of 50 and non equal value assignments incur 0 cost. Following the literature, we used $p_1 = 0.05$ and three values (i.e., colors) in each domain to generate these problems, which included 50 agents [4, 28, 29].

Scale-free network problems were generated using the Barabási–Albert (BA) model with an initial set of 7 connected agents, and additional 43 agents, which 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 rest of the problem parameters were identical to the random uniform problems.

Meeting scheduling problems included ninety agents, which scheduled 20 meetings into 20 time slots. Each agent was a participant in two randomly chosen meetings. For each pair of meetings, a travel time was chosen uniformly at random between 6 and 10. When the difference between the time slots of two meetings is less than the travel time between them, participants in both meetings are overbooked, and a cost equal to the number of overbooked agents is incurred.

Space limitations do not allow us to present all results obtained in our comprehensive experimental study, therefore we focus on the most significant results. We present results when applying Max-sum and DMS to two constant SCFGs, one splitting the costs evenly among the two function-nodes (0.5 version) and one in which the split had 95% of the cost in one function-node and 5% on the other (0.95 version). For random SCFGs we specify the range of costs from which the cost for the first function-node was selected, i.e., version 0.4–0.6 includes SCFGs where for each entry including cost c in the original constraint cost table, the cost for the corresponding entry of the first function-node was selected randomly between $0.4c$ and $0.6c$. Following [2] we present results of DMS with $\lambda = 0.9$. Our experiments with other λ values (0.5 and 0.7) validated that this version indeed performs best.

Figure 2 presents the costs per iteration and the costs of the anytime solution per iteration, when applying Max-sum, DSA, Bounded Max-sum and Max-sum_ADVP to SCFGs. For constant SCFGs and random SCFGs where costs were selected from a small range, the solutions found have lower cost than standard Max-sum. However, the results are of much lower quality (higher costs) than DSA. In order to avoid density we only depict the best anytime results among all versions of the algorithm presented in this graph, which were obtained when applying Max-sum to constant SCFGs 0.95. They are significantly better than the costs per iteration when applying Max-sum to any of the SCFGs but are far worse than the results produced by DSA and Max-sum_ADVP. The results

¹⁰ This range was selected so that the numbers do not become too small and due to precision, generate distorted SCFGs. Obviously, if the input costs are between 0 and 100, adding 100 to each cost can be the first step of the splitting method.

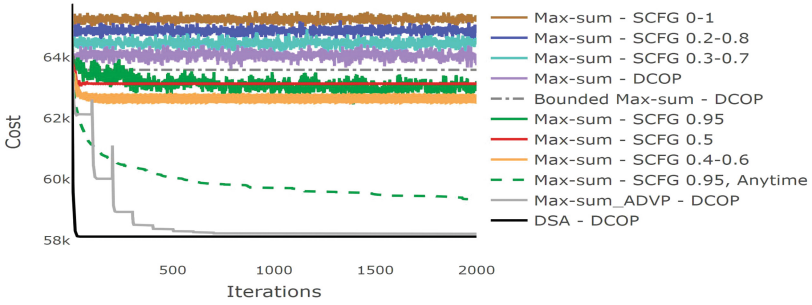


Fig. 2. Solution costs for Max-sum solving SCFGs of random uniform problems.

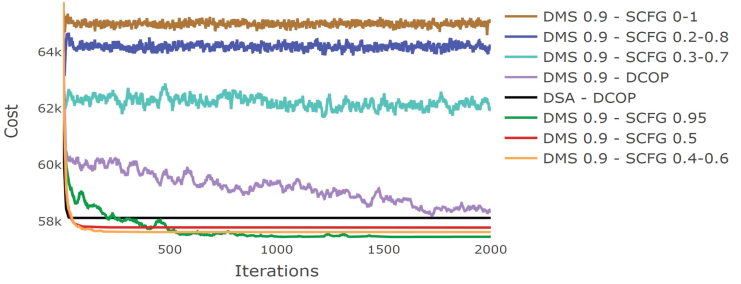


Fig. 3. Solutions costs for DMS solving SCFGs of random uniform problems.

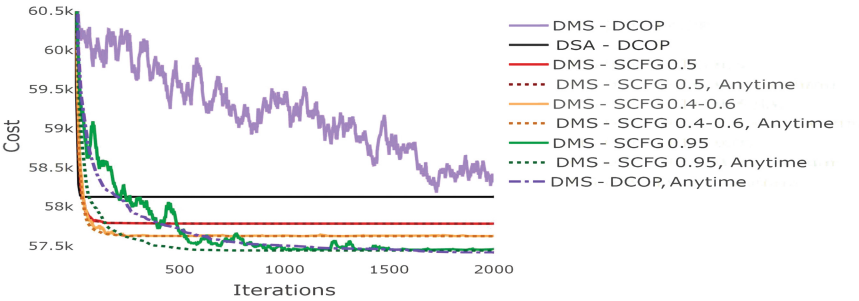


Fig. 4. Solution and anytime costs for DMS solving SCFGs of random uniform problems.

Max-sum produces on random SCFGs 0–1, are consistent with the results in [31], where only random splits were considered and damping was not used.

Figure 3 presents results for DMS applied to the same SCFGs. While the trends are the same, and the best results are achieved using constant SCFGs and random SCFGs with a small range for selecting the first cost, here the best results significantly outperform DSA. When using SCFG 0.5 and SCFG 0.4–0.6 the algorithm finds high quality solutions in a small number of iterations. When using SCFG 0.95, the algorithm performs more exploration, and after approximately 500 iterations it averagely finds solutions of higher quality than when the 0.5 and the 0.4–0.6 SCFGs are used. Notice that all results presented in Fig. 3 are cost per iteration and not the anytime costs. Figure 4 provides a closer look on the differences of the most successful versions of the algorithm and their anytime results. The anytime results of DMS on standard factor graphs are similar to the results per iteration of DMS on the SCFG 0.95 version. However, the anytime results of the 0.95 version converge much faster. When applied to SCFG 0.5 the algorithm converges very fast, yet the costs of the solutions are higher. When applied to random SCFGs 0.4–0.6, solutions with lower costs are also reached very fast. The small level of additional exploration gives an advantage in this case.

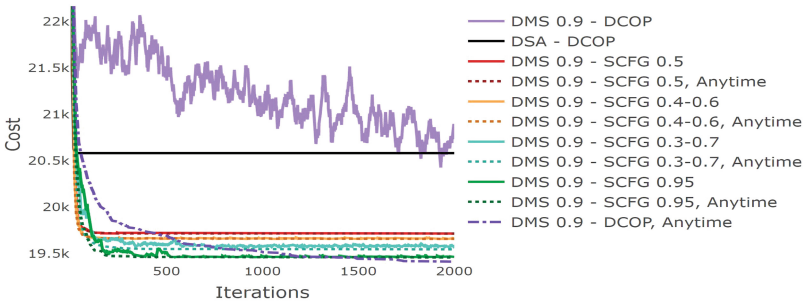


Fig. 5. Costs per iteration and anytime costs for DMS solving SCFGs of scale free nets.

The general trends were similar for denser uniform random problems with $p_1 = 0.7$ and for all other problem types, therefore, only selective graphs that give a closer view on the results of the most successful versions of DMS for these problems are presented. On scale free nets (Fig. 5), both the cost per iteration and the anytime costs of solutions found by DMS when applied to constant SCFGs and random SCFGs 0.4–0.6, converge much faster than the anytime result of DMS solving standard factor graphs. In addition, the costs per iteration and the anytime costs when applying DMS to random SCFGs 0.3–0.7 of scale free nets were lower than for constant SCFGs 0.5 and random SCFGs 0.4–0.6, thus, we depict them as well. On constant SCFGs 0.95, DMS found high quality solutions very fast.

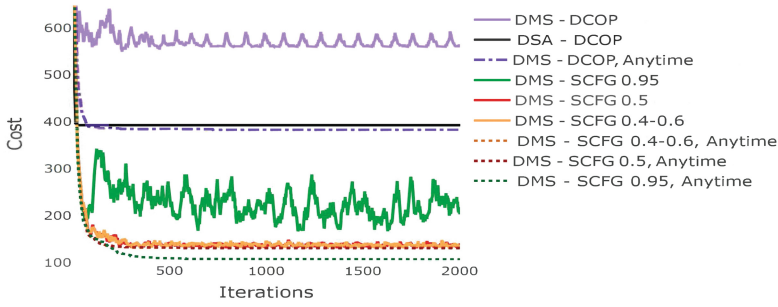


Fig. 6. Costs per iteration and anytime costs for DMS solving SCFGs of graph coloring problems.

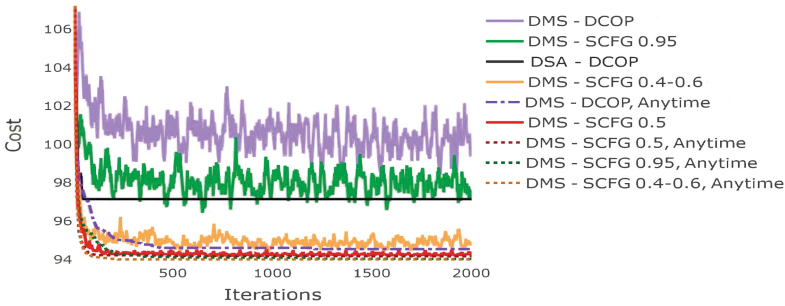


Fig. 7. Costs per iteration and anytime costs for DMS solving SCFGs of meeting scheduling problems.

For graph coloring problems (Fig. 6) DMS on standard factor graphs produces solutions with relatively high costs, and its anytime results are similar to the results of DSA. On the other hand the solutions found by DMS when applied to constant SCFGs 0.5 and to random SCFGs 0.4–0.6 are of significantly lower costs per iteration and anytime costs. The 0.95 version performs more exploration and the resulting anytime costs are lowest. The results for the meeting scheduling problems (Fig. 7) show a similar trend. When DMS is applied to constant SCFGs 0.95 it performs more exploration than when applied to SCFGs 0.5 and 0.4–0.6.

Figure 8 presents the time for convergence for each of the 50 runs of each of the algorithms and the percentage of problems on which the algorithm converged among the 50 runs on each problem type. The random SCFGs with larger ranges than 0.3–0.7 do not appear because they never converge. It is clear that constant SCFGs and the 0.4–0.6 random version have higher convergence rate than when DMS is applied to standard factor graphs. Moreover, it is also apparent that the constant 0.5 version converges very fast. Both the constant SCFG 0.95 and the random SCFG 0.4–0.6 versions converge slower and with slightly lower rates than the SCFG 0.5 version. Thus, they have more opportunities to perform

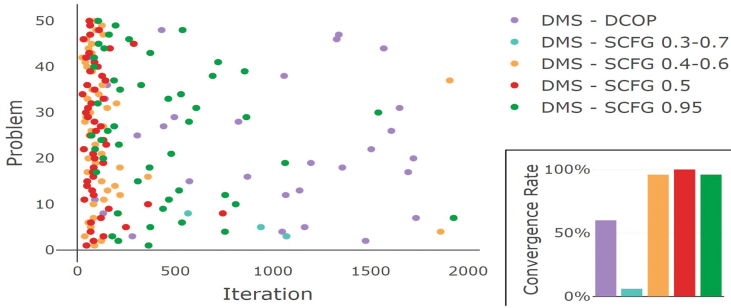


Fig. 8. Number of iterations for convergence and convergence rate on relatively sparse random uniform problems $p_1 = 0.2$.

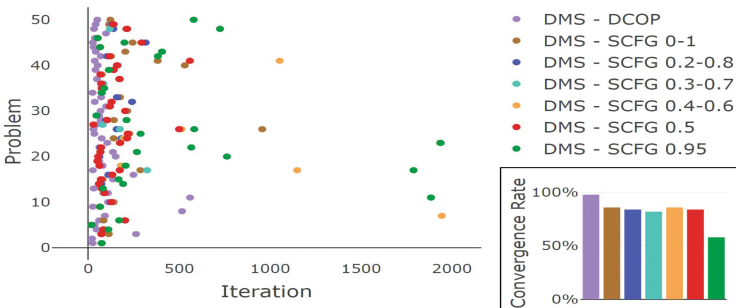


Fig. 9. Number of iterations for convergence and convergence rate on graph coloring problems.

exploration, which can explain the advantage they have in solution quality, as can be seen in Fig. 4.

Figure 9 presents the time for convergence and the convergence rate for graph coloring problems. Here, it is clear that fast convergence prevents the algorithm from finding solutions with low cost, when applied to standard factor graphs. On the other hand, when applied to SCFGs, DMS performs balanced exploration, which results in solutions with low costs. The constant SCFG 0.95 version triggers more exploration that results in lower anytime costs.¹¹

6.1 Runtime Overhead

If we consider each node in the factor graph as a separate agent, the overhead in runtime caused by splitting function-nodes is only for the variable-nodes, since

¹¹ For lack of space we do not present convergence graphs for the other problems. As expected the meeting scheduling convergence results were similar to graph coloring while the results for the other problem types were similar to the convergence results of the sparse uniform random problems.

they need to produce and send a double amount of messages. On the other hand, splitting function-nodes does not create a delay in the computation of function-nodes, since they are performed concurrently. However, commonly it is assumed that the role of the nodes in the factor-graphs are performed by the original (“real”) DCOP agents. Thus, when adding function-nodes to the graph we increment the runtime of each iteration. We note that a factor 2 increase in runtime can be easily achieved by having each agent that performed the role of a function-node in the original factor graph to perform the role of the two function-nodes resulting from its split. Our results indicate that for Symmetric SCFGs and SCFGs with limited randomness, the convergence of DMS is orders of magnitude faster than when using standard factor graphs and that is still true when each iteration takes a double amount of time.

6.2 Discussion

Our results indicate success in combining damping and asymmetry for balancing exploration and exploitation. Damping alone is enough to trigger Max-sum to explore solutions with low cost, however, it does not converge to high quality solutions within two thousand iterations. However, when using constant SCFGs split evenly (0.5), convergence is achieved within a few tens of iterations. The use of an uneven split of a constant SCFG (0.95 version) or random SCFGs with small ranges (0.4–0.6), allows limited exploration that results in solutions with both per iteration and anytime lower costs.

In order to explain this success one needs to look back at the results presented in Sect. 5. In problems with two variables and a single constraint, when splits are symmetric both Max-sum and DMS converge after a single iteration. Thus, in the general case, the difference between a single function-node representing a constraint and its symmetric split to two function-nodes is the ratio between the costs sent from the variable-nodes to the function-nodes and the costs in the function-nodes’ tables. Consider a variable-node v and its neighboring function-node f in factor graph G , which is symmetrically split to f' and f'' in factor graph G' (for simplicity assume that other function-nodes are not split). Let vc be the vector that holds the sum of costs v has received from function-node neighbors, which are not f , in iteration i . In G , in iteration $i + 1$, vc will be sent to f . In G' , vc will be sent to both function-nodes f' and f'' . However, each cost in the table of f is cut by half in the tables of f' and f'' . Thus, f' and f'' will make different calculations than f giving more consideration to the differences between costs in vc .

This phenomenon allows DMS to converge faster. Damping allows Max-sum to explore high quality solutions because it reduces the effect of multiple counting of information as observed by Pearl [2, 11]. On the other hand, it slows the aggregation of costs in the propagated cost vectors, and thus the differences between costs in the vectors are less pronounced in the beginning of the run. In contrast, the function-node cost tables are fixed throughout the run. Thus, damping delays assignment replacements, which are required for convergence. Symmetric SCFGs reduce the differences in the cost tables by half but do not

reduce the costs in the propagated vectors and thus, allow the required changes to take place faster.

On the other hands, as demonstrated in Sect. 5, random splits might generate oscillations. When such oscillations occur in multiple cycles in the graph, the beliefs propagated are inconsistent and prevent convergence.

7 Conclusion

We introduced a novel degree of freedom for balancing exploration and exploitation when using Max-sum for solving DCOPs, the level of asymmetry in the factor graph. To this end, we proposed to shift standard factor graphs representing a DCOP to equivalent split constraint factor graphs (SCFGs), in which each constraint is represented by two function-nodes. The level of asymmetry in SCFGs is determined by the similarity between table costs of function-nodes representing the same constraint.

We proved that Max-sum is guaranteed to converge to the optimal solution on cycles generated as a result of a constant split of a single constraint factor graph, regardless of the constant fraction and the damping factor used. This is in contrast to the general case where Max-sum is not guaranteed to converge on single cycle factor-graphs, and DMS might converge to a sub-optimal solution. Empirical results indicate that by tuning the two degrees of freedom, the damping factor and the level of asymmetry, Max-sum can produce solutions of high quality within a small number of iterations, even when an anytime framework cannot be used. When the level of exploration is too high, e.g., without damping, the algorithm fails to find solutions of high quality. On the other hand, limited exploration results in solutions of higher quality than immediate convergence.

References

1. Chli, M., Winsper, M.: Using the max-sum algorithm for supply chain emergence in dynamic multiunit environments. *IEEE Trans. Syst. Man Cybern.* **45**(3), 422–435 (2015)
2. Cohen, L., Zivan, R.: Max-sum revisited: the real power of damping. In: Sukthankar, G., Rodriguez-Aguilar, J.A. (eds.) *AAMAS 2017. LNCS (LNAI)*, vol. 10643, pp. 111–124. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71679-4_8
3. Farinelli, A., Rogers, A., Petcu, A., Jennings, N.: Decentralised coordination of low-power embedded devices using the max-sum algorithm. In: *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, vol. 2, pp. 639–646. International Foundation for Autonomous Agents and Multiagent Systems (2008)
4. Farinelli, A., Rogers, A., Petcu, A., Jennings, N.R.: Decentralized coordination of low-power embedded devices using the max-sum algorithm. In: *AAMAS*, pp. 639–646 (2008)
5. Farinelli, A., Rogers, A., Jennings, N.R.: Agent-based decentralised coordination for sensor networks using the max-sum algorithm. *Auton. Agents Multi-Agent Syst.* **28**(3), 337–380 (2014)

6. Gershman, A., Meisels, A., Zivan, R.: Asynchronous forward bounding. *J. Artif. Intell. Res.* **34**, 25–46 (2009)
7. Kschischang, F.R., Frey, B.J., Loeliger, H.A.: Factor graphs and the sum-product algorithm. *IEEE Trans. Inf. Theory* **47**(2), 181–208 (2001)
8. Lazic, N., Frey, B., Aarabi, P.: Solving the uncapacitated facility location problem using message passing algorithms. In: *International Conference on Artificial Intelligence and Statistics*, pp. 429–436 (2010)
9. Macarthur, K.S., Stranders, R., Ramchurn, S.D., Jennings, N.R.: A distributed anytime algorithm for dynamic task allocation in multi-agent systems. In: *AAAI* (2011)
10. Modi, P.J., Shen, W., Tambe, M., Yokoo, M.: Adopt: asynchronous distributed constraints optimization with quality guarantees. *Artif. Intell.* **161**(1–2), 149–180 (2005)
11. Pearl, J.: *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco (1988)
12. Petcu, A., Faltings, B.: A scalable method for multiagent constraint optimization. In: *IJCAI*, pp. 266–271 (2005)
13. Petcu, A., Faltings, B.: Approximations in distributed optimization. In: van Beek, P. (ed.) *CP 2005*. LNCS, vol. 3709, pp. 802–806. Springer, Heidelberg (2005). https://doi.org/10.1007/11564751_68
14. Pretti, M.: A message-passing algorithm with damping. *J. Stat. Mech. Theory Exp.* **11**, P11008 (2005)
15. Pujol-Gonzalez, M., Cerquides, J., Meseguer, P., Rodríguez-Aguilar, J.A., Tambe, M.: Engineering the decentralized coordination of UAVs with limited communication range. In: Bielza, C., et al. (eds.) *CAEPIA 2013*. LNCS (LNAI), vol. 8109, pp. 199–208. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40643-0_21
16. Ramchurn, S.D., Farinelli, A., Macarthur, K.S., Jennings, N.R.: Decentralized coordination in robocup rescue. *Comput. J.* **53**(9), 1447–1461 (2010)
17. Rogers, A., Farinelli, A., Stranders, R., Jennings, N.R.: Bounded approximate decentralized coordination via the max-sum algorithm. *Artif. Intell.* **175**(2), 730–759 (2011)
18. Ruozzi, N., Tatikonda, S.: Message-passing algorithms: reparameterizations and splittings. *IEEE Trans. Inf. Theory* **59**(9), 5860–5881 (2013)
19. Som, P., Chockalingam, A.: Damped belief propagation based near-optimal equalization of severely delay-spread UWB MIMO-ISI channels. In: *2010 IEEE International Conference on Communications (ICC)*, pp. 1–5. IEEE (2010)
20. Stranders, R., Farinelli, A., Rogers, A., Jennings, N.R.: Decentralised coordination of mobile sensors using the max-sum algorithm. In: *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, 11–17 July 2009*, pp. 299–304 (2009)
21. Tarlow, D., Givoni, I., Zemel, R., Frey, B.: Graph cuts is a max-product algorithm. In: *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence* (2011)
22. Tarlow, D., Givoni, I., Zemel, R.: Hop-map: efficient message passing with high order potentials. In: *AISTATS*, vol. 9, pp. 812–819 (2010)
23. Teacy, W.T.L., Farinelli, A., Grabham, N.J., Padhy, P., Rogers, A., Jennings, N.R.: Max-sum decentralized coordination for sensor systems. In: *AAMAS*, pp. 1697–1698 (2008)
24. Weiss, Y.: Correctness of local probability propagation in graphical models with loops. *Neural Comput.* **12**(1), 1–41 (2000)

25. Yanover, C., Meltzer, T., Weiss, Y.: Linear programming relaxations and belief propagation - an empirical study. *J. Mach. Learn. Res.* **7**, 1887–1907 (2006)
26. Yedidsion, H., Zivan, R., Farinelli, A.: Explorative max-sum for teams of mobile sensing agents. In: International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS 2014, Paris, France, 5–9 May 2014, pp. 549–556 (2014)
27. Yeoh, W., Felner, A., Koenig, S.: Bnb-ADOPT: an asynchronous branch-and-bound DCOP algorithm. *Artif. Intell. Res. (JAIR)* **38**, 85–133 (2010)
28. Zhang, W., Xing, Z., Wang, G., Wittenburg, L.: Distributed stochastic search and distributed breakout: properties, comparison and applications to constraints optimization problems in sensor networks. *Artif. Intell.* **161**(1–2), 55–88 (2005)
29. Zivan, R., Okamoto, S., Peled, H.: Explorative anytime local search for distributed constraint optimization. *Artif. Intell.* **211**, 1–21 (2014)
30. Zivan, R., Parash, T., Cohen, L., Peled, H., Okamoto, S.: Balancing exploration and exploitation in incomplete min/max-sum inference for distributed constraint optimization. *Auton. Agents Multi-Agent Syst.* **31**(5), 1165–1207 (2017)
31. Zivan, R., Parash, T., Naveh, Y.: 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, 25–31 July 2015, pp. 432–439 (2015)



A Large Neighboring Search Schema for Multi-agent Optimization

Khoi D. Hoang¹, Ferdinando Fioretto²(✉), William Yeoh¹, Enrico Pontelli³,
and Roie Zivan⁴

¹ Washington University in St. Louis, St. Louis, USA
{khai.hoang, wyeoh}@wustl.edu

² University of Michigan, Ann Arbor, USA
fioretto@umich.edu

³ New Mexico State University, Las Cruces, USA
epontell@cs.nmsu.edu

⁴ Ben Gurion University of the Negev, Beersheba, Israel
zivavr@cs.bgu.ac.il

Abstract. The Distributed Constraint Optimization Problem (DCOP) is an elegant paradigm for modeling and solving multi-agent problems which are distributed in nature, and where agents cooperate to optimize a global objective within the confines of localized communication. Since solving DCOPs optimally is NP-hard, recent effort in the development of DCOP algorithms has focused on incomplete methods. Unfortunately, many of such proposals do not provide quality guarantees or provide a loose quality assessment. Thus, this paper proposes the *Distributed Large Neighborhood Search (DLNS)*, a novel iterative local search framework to solve DCOPs, which provides guarantees on solution quality refining lower and upper bounds in an iterative process. Our experimental analysis of DCOP benchmarks on several important classes of graphs illustrates the effectiveness of DLNS in finding good solutions and tight upper bounds in both problems with and without hard constraints.

Keywords: Multiagent Systems
Distributed Constraint Optimization · Large Neighborhood Search

1 Introduction

In a cooperative *Multi-Agent System (MAS)*, multiple autonomous agents interact to pursue personal interests and to achieve common objectives. *Distributed Constraint Optimization Problems (DCOPs)* [8, 24, 30] have emerged as a prominent agent model to govern the agents' behavior in cooperative MAS. In this context, agents control variables of a weighted constrained problem and coordinate their value assignments to maximize the overall sum of resulting constraint utilities. DCOPs are suitable to model problems that are distributed in nature and where a collection of agents attempts to optimize a global objective within the confines of localized communication. They have been employed to model distributed versions of meeting scheduling problems [22, 39], allocation of targets

to sensors in a network [5], channel selection in wireless networks [41], coordination of multi-robot teams [43], optimization in smart grids [13, 19, 23], generation of coalition structures [37], and device scheduling in smart homes [12, 18, 34].

DCOP algorithms are classified as either *complete* or *incomplete*. Complete DCOP algorithms find optimal solutions at the cost of large runtimes, while incomplete approaches trade optimality for faster runtimes. Since finding optimal DCOP solutions is NP-hard [24], incomplete algorithms are often necessary to solve larger problems' instances. Unfortunately, several local search algorithms (e.g., DSA [42] and MGM [21]) and local inference algorithms (e.g., Max-Sum [5]) do not provide guarantees on the quality of the solutions found. More recent developments, such as region-optimal algorithms [17, 28, 38], sampling-based algorithms [10, 25, 27] and (Improved) Bounded Max-Sum [32, 33] alleviate this limitation. Region-optimal algorithms allow the specification of regions with a maximum size of k agents or t hops from each agent, and they optimally solve the subproblem within each region. Solution quality bounds are provided as a function of k [28], t [17], or a combination of both [38]. Sampling-based algorithms such as DUCT [27] and D-Gibbs [10, 25] extend the centralized UCT [1] and Gibbs [14] sampling algorithms, respectively. They are able to bound the quality of solutions found as a function of the number of samples used by the algorithms. Bounded Max-Sum [32] extends Max-Sum by solving an acyclic version of the DCOP graph and bounding its solution quality as a function of the edges removed from the graph. Improved Bounded Max-Sum [33] further provides tighter upper bounds. Although good quality assessments are essential for sub-optimal solutions, many incomplete DCOP approaches provide poor quality assessments and are unable to exploit domain-dependent knowledge and/or hard constraints present in problems.

We address these limitations by introducing the *Distributed Large Neighborhood Search (DLNS)* framework.¹ DLNS solves DCOPs by building on the strengths of LNS [35], a *centralized* meta-heuristic algorithm that iteratively explores complex neighborhoods of the search space to find better candidate solutions. LNS has been shown to be very effective in solving a number of optimization problems [15]. While typical LNS approaches focus on iteratively refining lower bounds of a solution, we propose a method that refines both lower and upper bounds, imposing no restriction on the objective and constraints.

Contributions: This paper makes the following contributions: **(1)** We provide a novel distributed local search framework for DCOPs, which provides quality guarantees by refining both lower and upper bounds of the solution found during the iterative process; **(2)** We introduce a novel distributed search algorithm called *Tree-based DLNS (T-DLNS)*, which is built within the DLNS framework and characterized by the ability to exploit the problem structure—T-DLNS provides also a low computational complexity per agent; and **(3)** Evaluations against state-of-the-art incomplete DCOP algorithms that also return bounded solutions show that T-DLNS converges to better solutions providing tighter quality bounds.

¹ An extended abstract of this work [6] appeared at AAMAS 2015.

2 Background

DCOP: A *Distributed Constraint Optimization Problem (DCOP)* is a tuple $\langle \mathcal{X}, \mathcal{D}, \mathcal{F}, \mathcal{A}, \alpha \rangle$, where: $\mathcal{X} = \{x_1, \dots, x_n\}$ is a set of *variables*; $\mathcal{D} = \{D_1, \dots, D_n\}$ is a set of finite *domains* (i.e., $x_i \in D_i$); $\mathcal{F} = \{f_1, \dots, f_e\}$ is a set of *utility functions* (also called *constraints*), where $f_i : \prod_{x_j \in \mathbf{x}^{f_i}} D_i \rightarrow \mathbb{R}_+ \cup \{-\infty\}$ and $\mathbf{x}^{f_i} \subseteq \mathcal{X}$ is the set of the variables (also called the *scope*) relevant to f_i ; $\mathcal{A} = \{a_1, \dots, a_p\}$ is a set of *agents*; and $\alpha : \mathcal{X} \rightarrow \mathcal{A}$ is a function that maps each variable to one agent. f_i specifies the utility of each combination of values assigned to the variables in \mathbf{x}^{f_i} . To ease readability, in the following, we assume all constraints are binary, and all agents control exactly one variable. Thus, we will use the terms “variable” and “agent” interchangeably and assume that $\alpha(x_i) = a_i$. The extensions to the n-ary constraint and multi-variable agents are straightforward [11].

A *partial assignment* σ is a value assignment to a set of variables $X_\sigma \subseteq \mathcal{X}$ that is consistent with the variables’ domains. The utility $\mathcal{F}(\sigma) = \sum_{f \in \mathcal{F}, \mathbf{x}^f \subseteq X_\sigma} f(\sigma)$ is the sum of the utilities of all the applicable utility functions in σ . A *solution* is a partial assignment σ for all the variables of the problem, i.e., with $X_\sigma = \mathcal{X}$. We will denote with \mathbf{x} a solution, while x_i is the value of x_i in \mathbf{x} . The goal is to find an optimal solution $\mathbf{x}^* = \operatorname{argmax}_{\mathbf{x}} \mathcal{F}(\mathbf{x})$.

Given a DCOP P , $G = (\mathcal{X}, E)$ is the **constraint graph** of P , where $(x, y) \in E$ iff $\exists f_i \in \mathcal{F}$ s.t. $\{x, y\} = \mathbf{x}^{f_i}$. A **DFS pseudo-tree** arrangement for G is a *spanning tree* $T = \langle \mathcal{X}, E_T \rangle$ of G s.t. if $f_i \in \mathcal{F}$ and $\{x, y\} \subseteq \mathbf{x}^{f_i}$, then x and y appear in the same branch of T . Edges of G that are *in* (resp. *out* of) E_T are called *tree edges* (resp. *backedges*). Tree edges connect a node with its parent and its children, while backedges connect a node with its *pseudo-parents* and its *pseudo-children*. We use $N(a_i) = \{a_j \in \mathcal{A} \mid (x_i, x_j) \in E\}$ to denote the neighbors of the agent a_i . We denote with $G^k = \langle X^k, E^k \rangle$, the subgraph of G used in the execution of our iterative algorithms, where $X^k \subseteq \mathcal{X}$ and $E^k \subseteq E$.

Figure 1 depicts: (a) the constraint graph of a DCOP with agents a_1, \dots, a_4 , each controlling a variable with domain $\{0,1\}$, (b) a pseudo-tree (solid lines identify tree edges, dotted lines refer to backedges), and (c) the DCOP constraints.

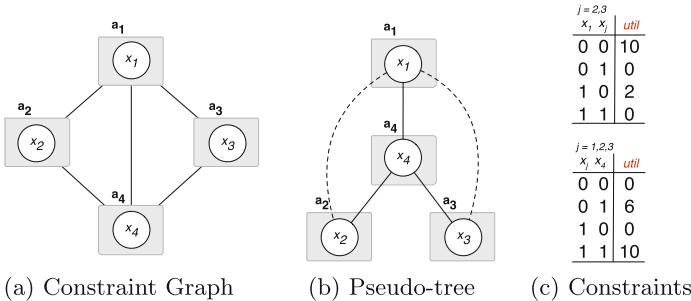


Fig. 1. Example DCOP.

LNS: In (*centralized*) *Large Neighborhood Search (LNS)* [35], an initial solution is iteratively improved by being repeatedly *destroyed* and *repaired*. Destroying a solution means selecting a subset of variables whose current values will be discarded. The set of such variables is the *large neighborhood (LN)*. Repairing a solution means finding a new value assignment for the LN variables, given that the non-destroyed variables maintain their values from the previous iteration. The peculiarity of LNS, compared to other local search techniques, is the (larger) size of the neighborhood to explore at each step. It relies on the intuition that searching over a larger neighborhood allows the process to escape local optima and find better candidate solutions.

3 The DLNS Framework

In this section, we introduce DLNS, a general distributed LNS framework to solve DCOPs. It takes into account the restriction that each agent is only aware of its local subproblem (i.e., its neighbors and constraints) which makes centralized LNS techniques unsuitable and infeasible for solving DCOPs.

Algorithm 1 shows the general structure of DLNS, as executed by each agent $a_i \in \mathcal{A}$. After initializing its iteration counter k (line 1), its current value assignment \mathbf{x}_i^0 (as a random choice, solving a relaxed problem, or by exploiting domain knowledge, when available), and its current lower and upper bounds LB_i^0 and UB_i^0 of the optimal utility (line 2), the agent, like in LNS, iterates through the destroy and repair phases (lines 3–7). Next, the agent executes a *bound* phase (line 8) which updates the current lower and upper bounds. If the solution is not satisfiable (i.e., if it has a negative infinite lower bound utility), the agent restores its value assignment to that of the previous iteration (line 9). The process repeats until a termination condition occurs (line 3). Possible termination conditions include reaching a maximum value of k , a timeout limit, or a confidence threshold on the error of the reported best solution.

Algorithm 1. DLNS

```

1  $k \leftarrow 0$ ;
2  $\langle \mathbf{x}_i^0, LB_i^0, UB_i^0 \rangle \leftarrow \text{VALUE-INITIALIZATION}()$ ;
3 while termination condition is not met do
4    $k \leftarrow k + 1$ ;
5    $z_i^k \leftarrow \text{DESTROY-ALGORITHM}()$ ;
6   if  $z_i^k = \circ$  then  $\mathbf{x}_i^k \leftarrow \text{NULL}$ ; else  $\mathbf{x}_i^k \leftarrow \mathbf{x}_i^{k-1}$  ;
7    $\mathbf{x}_i^k \leftarrow \text{REPAIR-ALGORITHM}(z_i^k)$ ;
8    $\langle LB_i^k, UB_i^k \rangle \leftarrow \text{BOUND-ALGORITHM}(\mathbf{x}_i^k)$ ;
9   if  $LB_i^k = -\infty$  then  $\mathbf{x}_i^k \leftarrow \mathbf{x}_i^{k-1}$  ;

```

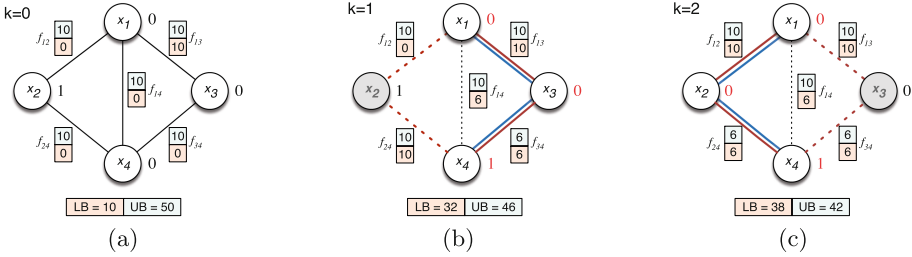


Fig. 2. DLNS with T-DBR example trace.

3.1 Destroy Phase

The result of this phase is the generation of a LN, which we refer to as LN^k as the subset of variables in \mathcal{X} that will need to be *repaired* in each iteration k . This step is executed in a distributed fashion, having each agent a_i calling a DESTROY-ALGORITHM to determine if its local variable x_i should be **destroyed** (\circ) or **preserved** (\star), as indicated by the flag z_i^k (line 5). We say that destroyed (resp. preserved) variables are (resp. are not) in LN^k . In a destroy process, such decisions can be either random or made by exploiting domain knowledge. DLNS allows the agents to use any destroy schema to achieve the desired outcome. Once the destroyed variables are determined, the agents reset their values and keep the values of the preserved variables from the previous iteration (line 6).

Example 1. Figure 2 illustrates the execution of the DLNS algorithm (whose details will be discussed later) over the first 3 iterations. The value of each variable is shown on its right. Gray shaded nodes denote variables that have been preserved, while white nodes denote those that have been destroyed, and thus are in the LN of that iteration. The values for the variable x_2 in iteration 1 and x_3 in iteration 2 are preserved to their values in iterations 0 and 1, respectively.

3.2 Repair Phase

In the repair phase, the DLNS agents seek to find a new value assignment for the destroyed variables by calling the REPAIR-ALGORITHM function (line 7). This process is carried out exclusively by the destroyed agents with the goal of finding an improved solution by searching over the large neighborhood. The DLNS framework imposes no restriction on the choice of algorithms by agents. In each iteration k , the agents coordinate the resolution of two problems: \hat{P}^k and \hat{P}^k , which we call *relaxations* of the original DCOP problem P . They are used to compute, respectively, a lower and an upper bound on the optimal utility for P , and are defined as follows. Let $E_{LN}^k = \{(x, y) \mid (x, y) \in E; x, y \in LN^k\}$ be the set of constraints involving exclusively destroyed variables (i.e., those in LN^k),

- $\hat{G}^k = \langle LN^k, \hat{E}^k \rangle$ is the **relaxation graph** of \hat{P} in iteration k , where $\hat{E}^k \subseteq E_{LN}^k$, is any subset of E_{LN}^k . The decision which edges to include in \hat{E}^k characterizes the subproblem to solve in iteration k .

- $\check{G}^k = \langle LN^k, \check{E}^k \rangle$ is the **relaxation graph** of \check{P} , where $\check{E}^k = \hat{E}^k \cup \{(x, y) \mid (x, y) \in E; x \in LN^k, y \notin LN^k\}$. It is the union of \hat{E}^k and the set of constraints whose scope has at least one destroyed variable.

Selecting \hat{E}^k and \check{E}^k is algorithmically dependent, and it is the factor that affects, in general, the algorithm’s complexity—we show later a simple choice for \hat{E}^k which allows our agents to solve each iteration in polynomial time.

In the problem \check{P}^k , we wish to find a partial assignment:

$$\check{\mathbf{x}}^k = \operatorname{argmax}_{\mathbf{x}} \left[\sum_{f \in \hat{E}^k} f(\mathbf{x}_i, \mathbf{x}_j) + \sum_{\substack{f \in \mathcal{F}, \mathbf{x}^f = \{x_i, x_j\} \\ x_i \in LN^k, x_j \notin LN^k}} f(\mathbf{x}_i, \check{\mathbf{x}}_j^{k-1}) \right]$$

where $\check{\mathbf{x}}_j^{k-1}$ is the value assigned to the preserved variable x_j for problem \check{P}^{k-1} in the previous iteration. The first summation is over all functions in \hat{E}^k , while the second is over all functions between a destroyed and a preserved variable. Thus, solving \check{P}^k means optimizing over all the destroyed variables given that the preserved ones take on their previous value, and ignoring the set of edges $E \setminus \check{E}^k$ that are not part of the relaxation graph. This partial assignment is used to compute lower bounds during the *bounding phase*.

In the problem \hat{P}^k , we wish to find a partial assignment:

$$\hat{\mathbf{x}}^k = \operatorname{argmax}_{\mathbf{x}} \sum_{f \in \hat{E}^k} f(\mathbf{x}_i, \mathbf{x}_j)$$

Thus, solving \hat{P}^k means optimizing over all the destroyed variables considering exclusively the set of edges \hat{E}^k that are part of the relaxation graph. This partial assignment is used to compute upper bounds during the *bounding phase*. Note that the partial assignments returned while solving these two relaxed problems involve exclusively the variables in LN^k .

Example 2. Consider again the example of Fig. 2. The relaxation graphs \check{G}^1 (in red), \hat{G}^1 (in blue) and \check{G}^2, \hat{G}^2 are illustrated in subfigures (b) and (c), respectively, with the nodes colored white and edges represented by bold solid lines. All other constraints (of the original problem P) are represented by black dotted lines. In more detail, $LN^1 = \{x_1, x_3, x_4\}$, $\hat{E}^1 = \{f_{13}, f_{34}\}$, $\check{E}^1 = \{f_{13}, f_{34}, f_{12}, f_{24}\}$, and $LN^2 = \{x_1, x_2, x_4\}$, $\hat{E}^2 = \{f_{12}, f_{24}\}$, $\check{E}^2 = \{f_{12}, f_{24}, f_{13}, f_{34}\}$. At each step, the resolution of the relaxed problems involves the functions represented by bold lines— \hat{P} is solved optimizing over the blue colored functions, and \check{P} over the red ones. Recall that while solving \hat{P} focuses solely on the functions in G^k , solving \check{P} further accounts for the functions that involve a destroyed and a preserved variable.

3.3 Bounding Phase

Once the relaxed problems are solved, *all* agents start the bounding phase, which results in computing the lower and upper bounds based on the partial assignments $\check{\mathbf{x}}^k$ and $\hat{\mathbf{x}}^k$. To do so, both solutions to the problems \check{P}^k and \hat{P}^k are extended to a solution $\check{\mathbf{x}}^k$ and $\hat{\mathbf{x}}^k$, respectively, for P , where the preserved variables $x_j \notin LN^k$ are assigned the values \check{x}_j^{k-1} from the previous iteration. The lower bound is computed by evaluating $\mathcal{F}(\check{\mathbf{x}}^k)$. The upper bound is computed by *combining* the optimal solution costs of two relaxed problems \hat{P}^k and \hat{P}^ℓ , solved in two different iterations. We focus on the case where $\ell < k$ is the iteration with the smallest upper bound found so far. Notice that $\sum_{f \in \hat{E}^k} f(\mathbf{x}_i, \mathbf{x}_j) \geq \sum_{f \in \hat{E}^k} f(\mathbf{x}_i^*, \mathbf{x}_j^*)$, where \mathbf{x}^* is the optimal solution of P ; therefore, reporting the optimal solutions found in two iterations will result in a larger utility, which is guaranteed to be an upper bound, albeit a conservative estimate. Thus, if a constraint is optimized in both iterations, we sum up the two solution qualities and subtract the minimum utility of that constraint. That will make the upper bound $\hat{F}^k(\hat{\mathbf{x}}^k) = \sum_{f \in \mathcal{F}} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k)$ smaller while preserving the correctness of the bound where:

$$\hat{f}^k(\mathbf{x}_i, \mathbf{x}_j) = \begin{cases} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k), & \text{if } f \in \hat{E}^k \setminus \hat{E}^\ell \\ \hat{f}^\ell(\hat{\mathbf{x}}_i^\ell, \hat{\mathbf{x}}_j^\ell), & \text{if } f \in \hat{E}^\ell \setminus \hat{E}^k \\ \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) + \hat{f}^\ell(\hat{\mathbf{x}}_i^\ell, \hat{\mathbf{x}}_j^\ell) - \min_{d_i \in D_i, d_j \in D_j} f(d_i, d_j), & \text{if } f \in \hat{E}^k \cap \hat{E}^\ell \\ \max_{d_i \in D_i, d_j \in D_j} f(d_i, d_j), & \text{otherwise.} \end{cases}$$

In other words, the utility of $\hat{F}^k(\hat{\mathbf{x}}^k)$ is composed of four parts. The first part involves the constraints considered while solving \hat{P}^k at iteration k , excluding those involved at iteration ℓ . The second part includes the constraints considered at iteration ℓ , excluding those involved at current iteration k . The third part involves the constraints adopted in both problems' iterations ℓ and k , and the fourth part involves the remaining constraints which were excluded when constructing both problems \hat{P}^ℓ and \hat{P}^k at iterations ℓ and k , respectively. We illustrate this process with the following example.

Example 3. Consider our example in Fig. 2. When $k = 0$, in subfigure (a), each agent randomly assigns a value to its variable, which results in a solution with utility $\mathcal{F}(\check{\mathbf{x}}^0) = f(\check{\mathbf{x}}_1^0, \check{\mathbf{x}}_2^0) + f(\check{\mathbf{x}}_1^0, \check{\mathbf{x}}_3^0) + f(\check{\mathbf{x}}_1^0, \check{\mathbf{x}}_4^0) + f(\check{\mathbf{x}}_2^0, \check{\mathbf{x}}_4^0) + f(\check{\mathbf{x}}_3^0, \check{\mathbf{x}}_4^0) = 0 + 10 + 0 + 0 + 0 = 10$ to get the lower bound. Moreover, \hat{P}^0 chooses the maximum utility of every constraint at iteration 0 and yields an upper bound as $\hat{F}^0(\hat{\mathbf{x}}^0) = \max \hat{f}^0(\hat{\mathbf{x}}_1^0, \hat{\mathbf{x}}_2^0) + \max \hat{f}^0(\hat{\mathbf{x}}_1^0, \hat{\mathbf{x}}_3^0) + \max \hat{f}^0(\hat{\mathbf{x}}_1^0, \hat{\mathbf{x}}_4^0) + \max \hat{f}^0(\hat{\mathbf{x}}_2^0, \hat{\mathbf{x}}_4^0) + \max \hat{f}^0(\hat{\mathbf{x}}_3^0, \hat{\mathbf{x}}_4^0) = 10 + 10 + 10 + 10 + 10 = 50$.

In the first iteration ($k = 1$), the destroy phase preserves x_2 , thus $\check{x}_2^1 = \check{x}_2^0 = 1$. In this example, the chosen algorithm builds the spanning tree with the remaining variables choosing f_{13} and f_{34} as tree edges, so $E^1 = \{f_{13}, f_{34}\} \subset E_{LN}^1 = \{f_{13}, f_{34}, f_{14}\}$. Thus the relaxation graph for \hat{P}^1 involves the edges $\{f_{13}, f_{34}, f_{12}, f_{24}\}$ (in red), and the relaxation graph for \hat{P}^1 involves the edges

$\{f_{13}, f_{34}\}$ (in blue). Solving \tilde{P}^1 yields a partial assignment $\tilde{\mathbf{x}}^1$ with utility $\tilde{F}^1(\tilde{\mathbf{x}}^1) = f(\tilde{\mathbf{x}}_1^1, \tilde{\mathbf{x}}_3^1) + f(\tilde{\mathbf{x}}_3^1, \tilde{\mathbf{x}}_4^1) + f(\tilde{\mathbf{x}}_1^1, \tilde{\mathbf{x}}_2^1) + f(\tilde{\mathbf{x}}_2^1, \tilde{\mathbf{x}}_4^1) = 10 + 6 + 0 + 10 = 26$, which results in a lower bound $\mathcal{F}(\tilde{\mathbf{x}}^1) = \tilde{F}^1(\tilde{\mathbf{x}}^1) + f(\tilde{\mathbf{x}}_1^1, \tilde{\mathbf{x}}_4^1) = 26 + 6 = 32$. Solving \hat{P}^1 yields a solution $\hat{\mathbf{x}}^1$ with utility $\hat{F}^1(\hat{\mathbf{x}}^1) = \hat{f}^1(\hat{\mathbf{x}}_1^1, \hat{\mathbf{x}}_3^1) + \hat{f}^1(\hat{\mathbf{x}}_3^1, \hat{\mathbf{x}}_4^1) + \max \hat{f}^1(\hat{\mathbf{x}}_1^1, \hat{\mathbf{x}}_2^1) + \max \hat{f}^1(\hat{\mathbf{x}}_1^1, \hat{\mathbf{x}}_4^1) + \max \hat{f}^1(\hat{\mathbf{x}}_2^1, \hat{\mathbf{x}}_4^1) = 10 + 6 + 10 + 10 + 10 = 46$, which is the current upper bound. After the first iteration, we have $\ell = 1$ as $\hat{F}^1(\hat{\mathbf{x}}^1) < \hat{F}^0(\hat{\mathbf{x}}^0)$.

Finally, in the second iteration ($k=2$), the destroy phase retains x_3 's value in the previous iteration $\tilde{\mathbf{x}}_3^2 = \tilde{\mathbf{x}}_3^1 = 0$, and the repair phase builds the new spanning tree with the remaining variables choosing f_{12} and f_{24} as tree edges with $E^2 = \{f_{12}, f_{24}\}$. Thus the relaxation graph for \tilde{P}^2 involves the edges $\{f_{12}, f_{24}, f_{13}, f_{34}\}$, and the relaxation graph for \hat{P}^2 involves the edges $\{f_{12}, f_{24}\}$. Solving \tilde{P}^2 and \hat{P}^2 yields partial assignments $\tilde{\mathbf{x}}^2$ and $\hat{\mathbf{x}}^2$, respectively, with utilities $\tilde{F}^2(\tilde{\mathbf{x}}^2) = 10 + 6 + 10 + 6 = 32$, which results in a lower bound $\mathcal{F}(\tilde{\mathbf{x}}^2) = 32 + 6 = 38$, and an upper bound $\hat{F}^2(\hat{\mathbf{x}}^2) = \hat{f}^2(\hat{\mathbf{x}}_1^2, \hat{\mathbf{x}}_2^2) + \hat{f}^1(\hat{\mathbf{x}}_2^2, \hat{\mathbf{x}}_4^2) + \hat{f}^1(\hat{\mathbf{x}}_1^2, \hat{\mathbf{x}}_3^1) + \hat{f}^1(\hat{\mathbf{x}}_3^1, \hat{\mathbf{x}}_4^1) + \max \hat{f}^1(\hat{\mathbf{x}}_1^2, \hat{\mathbf{x}}_4^1) = 10 + 6 + 10 + 6 + 10 = 42$. After this iteration, $\ell = 2$.

Crucially, this framework enables DLNS to iteratively refine both lower and upper bounds of the solution, without imposing any restrictions on the form of the objective function and of the constraints adopted.²

4 Tree-Based DLNS (T-DLNS)

Having discussed the general DLNS framework, we now introduce an efficient (polynomial-time) DLNS algorithm by specifying the construction of the problem relaxation graphs.

Tree-based DLNS (T-DLNS) defines the relaxed DCOPs \tilde{P}^k and \hat{P}^k using a spanning tree $T^k = \langle LN^k, E_{T^k} \rangle$, computed from G and LN^k and ignoring back edges. Solving the problem \tilde{P}^k means optimizing over T^k and considering edges connecting destroyed and preserved variables. Thus, $\tilde{G}^k = \langle LN^k, \tilde{E}^k \rangle$ where $\tilde{E}^k = E_{T^k} \cup \{(x, y) \mid (x, y) \in E; x \in LN^k, y \notin LN^k\}$. Solving the problem \hat{P}^k means optimizing over the spanning tree $\hat{G}^k = T^k$.

T-DLNS uses a complete inference-based algorithm composed of two phases operating on a tree-structured network [30]. This algorithm is complete on tree networks. Thus, while it will solve optimally and efficiently our relaxations, it will not guarantee to find an optimal solution for the original DCOP problem:

- In the *utility propagation* phase, each agent, starting from the leaves of the pseudo-tree, projects out its variable and sends its projected utilities to its parent. These utilities are propagated up the tree induced from E_{T^k} until they reach the root. The hard constraints of the problem are handled in this phase by pruning all inconsistent values before sending a message to its parent.

² However, it does not imply that the lower and upper bounds will converge to the same value.

- Once the root agent receives the utilities from all its children, it starts the *value propagation* phase by selecting the value that maximizes its utility and sends it to its children, which repeat the same process. The problem is solved as soon as the values reach the leaves.

The solving schema of T-DLNS is similar to that of DPOP [30] in that it uses utility and value propagation phases; however, the different underlying relaxation graph adopted imposes several important differences. Algorithm 2

Algorithm 2. T-DLNS(z_i^k)

```

10  $\mathbf{T}_i^k \leftarrow \text{RELAXATION}(z_i^k)$ 
11  $\text{UTIL-PROPAGATION}(\mathbf{T}_i^k)$ 
12  $\langle \tilde{\chi}_i^k, \hat{\chi}_i^k \rangle \leftarrow \text{VALUE-PROPAGATION}(\mathbf{T}_i^k)$ 
13  $\langle LB_i^k, UB_i^k \rangle \leftarrow \text{BOUND-PROPAGATION}(\tilde{\chi}_i^k, \hat{\chi}_i^k)$ 
14 return  $\langle \tilde{\mathbf{x}}_i^k, LB_i^k, UB_i^k \rangle$ 

```

Procedure. UTIL-Propagation(\mathbf{T}_i^k)

```

15 receive  $\text{UTIL}_{a_c}(\tilde{U}_c, \hat{U}_c)$  from each  $a_c \in C_i^k$ 
16 forall values  $\mathbf{x}_i, \mathbf{x}_{P_i^k}$  do
17    $\tilde{U}_i(\mathbf{x}_i, \mathbf{x}_{P_i^k}) \leftarrow f(\mathbf{x}_i, \mathbf{x}_{P_i^k}) + \sum_{a_c \in C_i^k} \tilde{U}_c(\mathbf{x}_i) + \sum_{x_j \notin LN^k} f(\mathbf{x}_i, \tilde{\mathbf{x}}_j^{k-1})$ 
18    $\hat{U}_i(\mathbf{x}_i, \mathbf{x}_{P_i^k}) \leftarrow f(\mathbf{x}_i, \mathbf{x}_{P_i^k}) + \sum_{a_c \in C_i^k} \hat{U}_c(\mathbf{x}_i)$ 
19 forall values  $\mathbf{x}_{P_i^k}$  do
20    $\langle \tilde{U}'_i(\mathbf{x}_{P_i^k}), \hat{U}'_i(\mathbf{x}_{P_i^k}) \rangle \leftarrow \langle \max_{\mathbf{x}_i} \tilde{U}_i(\mathbf{x}_i, \mathbf{x}_{P_i^k}), \max_{\mathbf{x}_i} \hat{U}_i(\mathbf{x}_i, \mathbf{x}_{P_i^k}) \rangle$ 
21 send  $\text{UTIL}_{a_i}(\tilde{U}'_i, \hat{U}'_i)$  msg to  $P_i^k$ 

```

Function. VALUE-Propagation(\mathbf{T}_i^k)

```

22 if  $P_i^k = \text{NULL}$  then
23    $\langle \tilde{\mathbf{x}}_i^k, \hat{\mathbf{x}}_i^k \rangle \leftarrow \langle \text{argmax}_{\mathbf{x}_i} \tilde{U}_i(\mathbf{x}_i), \text{argmax}_{\mathbf{x}_i} \hat{U}_i(\mathbf{x}_i) \rangle$ 
24   send  $\text{VALUE}_{a_i}(\tilde{\mathbf{x}}_i^k, \hat{\mathbf{x}}_i^k)$  msg to each  $a_j \in N(a_i)$ 
25   forall  $a_j \in N(a_i)$  do
26     receive  $\text{VALUE}_{a_j}(\tilde{\mathbf{x}}_j^k, \hat{\mathbf{x}}_j^k)$  msg from  $a_j$ 
27     Update  $x_j$  in  $\langle \tilde{\chi}_i^k, \hat{\chi}_i^k \rangle$  with  $\langle \tilde{\mathbf{x}}_j^k, \hat{\mathbf{x}}_j^k \rangle$ 
28 else
29   forall  $a_j \in N(a_i)$  do
30     receive  $\text{VALUE}_{a_j}(\tilde{\mathbf{x}}_j^k, \hat{\mathbf{x}}_j^k)$  msg from  $a_j$ 
31     Update  $x_j$  in  $\langle \tilde{\chi}_i^k, \hat{\chi}_i^k \rangle$  with  $\langle \tilde{\mathbf{x}}_j^k, \hat{\mathbf{x}}_j^k \rangle$ 
32     if  $a_j = P_i^k$  then
33        $\langle \tilde{\mathbf{x}}_i^k, \hat{\mathbf{x}}_i^k \rangle \leftarrow \langle \text{argmax}_{\mathbf{x}_i} \tilde{U}_i(\mathbf{x}_i), \text{argmax}_{\mathbf{x}_i} \hat{U}_i(\mathbf{x}_i) \rangle$ 
34       send  $\text{VALUE}_{a_i}(\tilde{\mathbf{x}}_i^k, \hat{\mathbf{x}}_i^k)$  msg to each  $a_j \in N(a_i)$ 
35 return  $\langle \tilde{\chi}_i^k, \hat{\chi}_i^k \rangle$ 

```

Procedure. *BOUND-Propagation*($\check{\chi}_i^k, \hat{\chi}_i^k$)

```

36 receive BOUNDSac(LBck, UBck) msg from each ac ∈ Ci
37 LBik ← f( $\check{\mathbf{x}}_i^k, \check{\mathbf{x}}_{P_i}^k$ ) + ∑aj ∈ PPik f( $\check{\mathbf{x}}_i^k, \check{\mathbf{x}}_j^k$ ) + ∑ac ∈ Ci LBck
38 UBik ←  $\hat{f}^k(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}_{P_i})$  + ∑aj ∈ PPik  $\hat{f}^k(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}_j)$  + ∑ac ∈ Ci UBck
39 send BOUNDSai(LBik, UBik) msg to Pi

```

shows the pseudocode of T-DLNS. We use the following notations: P_i^k , C_i^k , PP_i^k denote the parent, the set of children, and pseudo-parents of the agent a_i , at iteration k . The set of these items is referred to as \mathbf{T}_i^k , which is a_i 's local view of the pseudo-tree T^k . $\check{\chi}_i$ and $\hat{\chi}_i$ denote a_i 's context (i.e., the values for each $x_j \in N(a_i)$) with respect to problems \check{P} and \hat{P} , respectively. We assume that by the end of the destroy phase (line 6) each agent knows its current context as well as which of its neighboring agents has been destroyed or preserved.

In each iteration k , T-DLNS executes these phases:

Repair Phase. It constructs a pseudo-tree T^k (line 10), which ignores, from G , the preserved variables as well as the functions involving these variables in their scopes. The construction prioritizes tree-edges that have not been chosen in previous pseudo-trees over the others. The T-DLNS solving phase is composed of two phases operating on the relaxed pseudo-tree T^k , and executed synchronously:

1. *Utility Propagation:* After the pseudo-tree T^k is constructed (line 11), each leaf agent computes the optimal sum of utilities in its subtree considering exclusively tree edges (i.e., edges in E_{T^k}) and edges with destroyed variables. Each leaf agent computes the utilities $\check{U}_i(\mathbf{x}_i, \mathbf{x}_{P_i^k})$ and $\hat{U}_i(\mathbf{x}_i, \mathbf{x}_{P_i^k})$ for each pair of values of its variable \mathbf{x}_i and its parent's variable $\mathbf{x}_{P_i^k}$ (lines 16–18), in preparation for retrieving the solutions of \check{P} and \hat{P} , used during the bounding phase. The agent projects itself out (lines 19–20) and sends the projected utilities to its parent in a UTIL message (line 21). Each agent, upon receiving the UTIL message from each child, performs the same operations. Thus, these utilities will propagate up the pseudo-tree until they reach the root agent.
2. *Value Propagation:* Once the utility propagation is completed (line 12) the root agent computes its optimal values $\check{\mathbf{x}}_i^k$ and $\hat{\mathbf{x}}_i^k$ for the relaxed DCOPs \check{P} and \hat{P} , respectively (line 23). Then, it sends its values to all its neighbors in a VALUE message (line 24). When any of its children receive this message, they also compute their optimal values and sends them to all their neighbors (lines 32–34). Thus, these values propagate down the pseudo-tree until they reach the leaves, at which point every agent has chosen its respective values. In this phase, in preparation for the bounding phase, when each agent receives a VALUE message from its neighbor, it will also update the corresponding value in its contexts $\check{\chi}_i^k$ and $\hat{\chi}_i^k$ (lines 25–27 and 30–31).

Bounding Phase. Once the relaxed DCOPs \check{P} and \hat{P} have been solved, the algorithm starts the bound propagation phase (line 13). Each leaf agent

of the pseudo-tree T computes the lower and upper bounds LB_i^k and UB_i^k (lines 37–38). These bounds are sent to the agent’s parent in T (line 39). When its parent receives this message from all its children (line 36), it performs the same operations. The lower and upper bounds of the whole problem are determined when the bounds reach the root agent.

5 Theoretical Properties

Theorem 1. For each LN^k , $\mathcal{F}(\check{\mathbf{x}}^k) \leq \mathcal{F}(\mathbf{x}^*) \leq \hat{F}^k(\hat{\mathbf{x}}^k)$.

Proof. The result $\mathcal{F}(\check{\mathbf{x}}^k) \leq \mathcal{F}(\mathbf{x}^*)$ follows from that $\check{\mathbf{x}}^k$ is an optimal solution of the relaxed problem \check{P} whose functions are a subset of \mathcal{F} .

By definition of $\hat{F}^k(\mathbf{x})$, it follows that:

$$\begin{aligned}
 \hat{F}^k(\hat{\mathbf{x}}^k) &= \sum_{f \in \mathcal{F}} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) \\
 &= \sum_{f \in \hat{E}^k \setminus \hat{E}^\ell} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) + \sum_{f \in \hat{E}^\ell \setminus \hat{E}^k} \hat{f}^\ell(\hat{\mathbf{x}}_i^\ell, \hat{\mathbf{x}}_j^\ell) + \sum_{f \notin \hat{E}^k \cup \hat{E}^\ell} \max_{d_i \in D_i, d_j \in D_j} f(d_i, d_j) \\
 &\quad + \sum_{f \in \hat{E}^k \cap \hat{E}^\ell} \left(\hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) + \hat{f}^\ell(\hat{\mathbf{x}}_i^\ell, \hat{\mathbf{x}}_j^\ell) - \min_{d_i \in D_i, d_j \in D_j} f(d_i, d_j) \right) \\
 &= \sum_{f \in \hat{E}^k} \hat{f}^k(\hat{\mathbf{x}}_i^k, \hat{\mathbf{x}}_j^k) + \sum_{f \in \hat{E}^\ell} \hat{f}^\ell(\hat{\mathbf{x}}_i^\ell, \hat{\mathbf{x}}_j^\ell) \\
 &\quad + \sum_{f \notin \hat{E}^k \cup \hat{E}^\ell} \max_{d_i \in D_i, d_j \in D_j} f(d_i, d_j) - \sum_{f \in \hat{E}^k \cap \hat{E}^\ell} \min_{d_i \in D_i, d_j \in D_j} f(d_i, d_j) \\
 &\geq \sum_{f \in \hat{E}^k} f(\hat{\mathbf{x}}_i^*, \hat{\mathbf{x}}_j^*) + \sum_{f \in \hat{E}^\ell} f(\hat{\mathbf{x}}_i^*, \hat{\mathbf{x}}_j^*) + \sum_{f \notin \hat{E}^k \cup \hat{E}^\ell} f(\hat{\mathbf{x}}_i^*, \hat{\mathbf{x}}_j^*) - \sum_{f \in \hat{E}^k \cap \hat{E}^\ell} f(\hat{\mathbf{x}}_i^*, \hat{\mathbf{x}}_j^*) \\
 &\geq \sum_{f \in \hat{E}^k \cup \hat{E}^\ell} f(\hat{\mathbf{x}}_i^*, \hat{\mathbf{x}}_j^*) + \sum_{f \notin \hat{E}^k \cup \hat{E}^\ell} f(\hat{\mathbf{x}}_i^*, \hat{\mathbf{x}}_j^*) \\
 &\geq \sum_{f \in \mathcal{F}} f(\hat{\mathbf{x}}_i^*, \hat{\mathbf{x}}_j^*) \\
 &\geq \mathcal{F}(\mathbf{x}^*)
 \end{aligned}$$

and, thus, $\mathcal{F}(\check{\mathbf{x}}^k) \leq \mathcal{F}(\mathbf{x}^*) \leq \hat{F}^k(\hat{\mathbf{x}}^k)$ for each LN^k . ■

Corollary 1. An approximation ratio for the problem is

$$\rho = \frac{\min_k \hat{F}^k(\hat{\mathbf{x}}^k)}{\max_k \mathcal{F}(\check{\mathbf{x}}^k)} \geq \frac{\mathcal{F}(\mathbf{x}^*)}{\max_k \mathcal{F}(\check{\mathbf{x}}^k)}.$$

Theorem 2. In each iteration, T -DLNS requires $O(|\mathcal{F}|)$ number of messages of size in $O(d)$, where $d = \max_{a_i \in \mathcal{A}} |D_i|$.

Proof. In the Value Propagation Phase of Algorithm 2, each agent sends a message to its neighbors (lines 24 and 34). Thus, the overall amount of messages

sent in this phase by the agents is $2\|\mathcal{F}\|$. All other phases use up to $|\mathcal{A}|$ messages (which are reticulated from the leaves to the root of the pseudo-tree and vice-versa). Therefore, T-DLNS requires $O(|\mathcal{F}|)$ messages in each iteration. The largest messages are sent during the *Utility Propagation Phase*, where each agent (excluding the root agent) sends a message containing a value for each element of its domain (line 21). Thus, the size of the DLNS messages is in $O(d)$. ■

Theorem 3. *In each iteration, the number of constraint checks of each T-DLNS agent is in $O(d^2)$, where $d = \max_{a_i \in \mathcal{A}} |D_i|$.*

Proof. The largest amount of constraint checks per iteration is performed during the Util-Propagation Phase. In this phase, each agent (except the root agent) computes the lower and upper bound utilities for each value of its variable \mathbf{x}_i and its parent’s variable $\mathbf{x}_{P_i^k}$ (lines 17–18). Therefore, the number of constraint checks per iteration of each agent is in $O(d^2)$. ■

6 Related Work

In addition to the algorithms described in the introduction, several extensions to complete search-based algorithms that trade solution quality for faster runtimes have been proposed [24, 40]. Another line of work has investigated non-iterative versions of inference-based incomplete DCOP algorithms, such as ADPOP [29] and p-OPT [26]. These algorithms operate on relaxations of the original DCOP. ADPOP is an incomplete version of DPOP that bounds the maximal message size transmitted over the network, trading off message size for better runtimes. p-OPT ignores some edges of the induced chordal graph of the DCOP and solves exactly the problem over such subgraphs to generate an approximate solution. Both algorithms are different from DLNS in that they operate in a single iteration only and, thus, do not refine the solution found. The algorithm that shares most similarities with DLNS is LS-DPOP [31]. LS-DPOP runs several local searches on pseudo-trees. However, unlike DLNS, it operates in a single iteration, does not change its neighborhood, and does not provide quality guarantees.

7 Experimental Results

We evaluate the DLNS framework against representative state-of-the-art incomplete DCOP algorithms, with and without quality guarantees. We choose T-DLNS as a representative algorithm of the DLNS framework. We select DSA as a representative incomplete *search*-based DCOP algorithm; Max-Sum (MS) and Bounded MS (BMS) as representative *inference*-based DCOP algorithms; and k -optimal algorithms (KOPT2 and KOPT3) as representative *region optimal*-based DCOP methods. All algorithms are selected based on their performance and popularity. We use the FRODO framework [20] to run MS and DSA,³ the

³ We use DSA-B and set $p = 0.6$.

authors’ code of BMS [32], and the DALO framework [17] for KOPT. We also force T-DLNS first large neighboring exploration to use the same tree as that used by BMS. We experimentally observed that using this option improves the effectiveness of T-DLNS in finding high quality solutions.

Random DCOPs. First, we evaluate the algorithms on random DCOPs over *random*, *grid*, and *scale-free* topologies. The instances for each topology are generated as follows: For random networks, we create an n -node network, whose density p_1 produces $\lfloor n(n-1)p_1 \rfloor$ edges in total. We do not bound the tree-width, which is based on the underlying graph. For grid networks, we create an n -node network arranged in a rectangular grid, where internal nodes are connected to four neighboring nodes and nodes on the edges (resp. corners) are connected to two (resp. three) neighbors. Finally, for scale-free networks, we create an n -node network based on the Barabasi-Albert model [2]. Starting from a connected 2-node network, we repeatedly add a new node, randomly connecting it to two existing nodes. In turn, these two nodes are selected with probabilities that are proportional to the numbers of their connected edges. The total number of edges is $2(n-2) + 1$.

We generate 50 instances for each topology, ensuring that the underlying graph is connected. The utility functions are generated using random utilities in $[0, 100]$. We set as default parameters, $|D_i| = 10$ for all variables, $p_1 = 0.5$, and for instances with hard constraints, $p_2 = 0.5$. We use a random destroy strategy for the T-DLNS algorithms, in which each agent destroys a variable with probability $p = 0.5$. The runtime of all the algorithms is measured using the *simulated runtime* metric [36], and averaged over all instances. The experiments are performed on an Intel i7 Quadcore 3.4 GHz machine with 16GB of RAM.

Tables 1, 2 and 3 report the approximation ratio ρ , simulated runtime t , the normalized lower bound (LB), and the normalized upper bound (UB) values. The LB (UB) value of each algorithm is normalized over the LBs (UBs) reported by all algorithms. A normalized lower (upper) bound of 0 means that it is the worst lower (upper) bound among all lower (upper) bounds. Similarly, a normalized lower (upper) bound of 1 means that it is the best lower (upper) bound. The best approximation ratios, normalized lower (upper) bounds, and runtimes are shown in bold. All tables report results for problem with and without hard constraints (HC).

Table 1. Experimental results on *random* networks. Times are in ms.

A HC	T-DLNS				BMS				KOPT2				KOPT3				MS		DSA		
	ρ	LB	UB	t	ρ	LB	UB	t	ρ	LB	UB	t	ρ	LB	UB	t	LB	t	LB	t	
25	✓	1.36	0.95	0.97	377	1.75	0.78	0.92	310	9.14	0.88	0.15	293	7.10	0.89	0.20	1204	0.88	253	0.93	84
		1.86	0.94	0.97	341	2.70	0.65	0.97	321	9.14	0.82	0.22	298	7.10	0.85	0.28	1319	0.84	235	0.91	81
64	✓	2.46	0.92	0.97	1703	3.27	0.69	0.97	2936	21.88	0.87	0.07	3051	16.66	0.76	0.11	4935	0.86	2471	0.96	241
		2.46	0.92	0.97	1703	3.27	0.69	0.97	2936	21.88	0.77	0.13	3124	16.66	0.64	0.20	4943	0.78	2188	0.95	235
100	✓	1.64	0.94	0.97	7580	1.95	0.84	0.92	6312	33.64	0.82	0.05	5222	25.48	0.80	0.07	8193	0.86	6368	0.97	432
	✓	2.70	0.92	0.97	5089	3.49	0.71	0.97	7324	33.64	0.68	0.10	5166	25.48	0.64	0.14	7884	0.76	6852	0.96	540
144	✓	1.70	0.94	0.97	26156	1.97	0.86	0.92	19372	48.02	0.82	0.03	14223	36.26	0.83	0.05	17239	0.88	6578	0.97	722
	✓	2.88	0.91	0.97	16562	3.58	0.73	0.97	20313	48.02	0.68	0.07	15421	36.26	0.68	0.10	16342	0.78	7161	0.96	1517

Table 2. Experimental results on *grid* networks. Times are in ms.

A HC	T-DLNS				BMS				KOPT2				KOPT3				MS		DSA	
	ρ	LB	UB	t	ρ	LB	UB	t	ρ	LB	UB	t	ρ	LB	UB	t	LB	t	LB	t
25	1.06	0.96	0.94	270	1.26	0.83	0.92	17	9.14	0.91	0.11	85	7.10	0.93	0.14	108	0.93	31	0.90	38
	✓	1.16	0.94	0.92	279	1.42	0.75	0.90	19	9.14	0.85	0.12	82	7.10	0.88	0.15	107	0.87	31	0.82
64	1.08	0.96	0.97	759	1.29	0.83	0.94	79	21.88	0.91	0.05	199	16.66	0.92	0.06	266	0.82	51	0.91	52
	✓	1.21	0.95	0.93	716	1.49	0.77	0.92	64	21.88	0.87	0.05	200	16.66	0.88	0.07	264	0.75	55	0.85
100	1.09	0.96	0.97	1422	1.30	0.83	0.94	173	33.64	0.91	0.03	319	25.48	0.92	0.04	415	0.84	58	0.91	64
	✓	1.23	0.96	0.96	1253	1.51	0.77	0.94	151	33.64	0.87	0.03	319	25.48	0.89	0.04	422	0.75	58	0.87
144	1.10	0.96	0.97	2900	1.31	0.83	0.94	249	48.02	0.92	0.02	833	36.26	0.93	0.03	1082	0.83	69	0.92	185
	✓	1.24	0.95	0.96	2489	1.52	0.76	0.95	227	48.02	0.86	0.02	460	36.24	0.89	0.03	655	0.69	44	0.87

Table 3. Experimental results on *scale-free* networks. Times are in ms.

A HC	T-DLNS				BMS				KOPT2				KOPT3				MS		DSA	
	ρ	LB	UB	t	ρ	LB	UB	t	ρ	LB	UB	t	ρ	LB	UB	t	LB	t	LB	t
25	1.22	0.95	0.94	523	1.58	0.78	0.90	137	9.14	0.89	0.13	594	7.10	0.89	0.17	1498	0.85	166	0.91	111
	✓	1.54	0.93	0.83	299	2.17	0.65	0.82	142	9.14	0.82	0.15	160	7.10	0.81	0.20	535	0.83	162	0.85
25	1.28	0.95	0.96	1099	1.62	0.79	0.91	393	21.88	0.90	0.05	641	16.66	0.84	0.08	1901	0.85	414	0.93	150
	✓	1.64	0.94	0.96	880	2.23	0.69	0.95	423	21.88	0.84	0.08	544	16.66	0.75	0.11	1256	0.78	414	0.89
25	1.30	0.96	0.97	1922	1.61	0.80	0.93	712	33.64	0.91	0.03	935	25.48	0.84	0.05	3199	0.85	532	0.94	180
	✓	1.70	0.95	0.95	1540	2.26	0.71	0.94	832	33.64	0.86	0.05	947	25.48	0.75	0.08	1914	0.75	584	0.93
25	1.31	0.96	0.97	4322	1.61	0.82	0.93	1585	48.02	0.92	0.02	1651	36.26	0.84	0.04	3614	0.84	626	0.95	447
	✓	1.74	0.94	0.97	3294	2.27	0.71	0.96	1681	48.02	0.87	0.03	1714	36.26	0.74	0.05	3211	0.72	647	0.93

Table 1 tabulates the results for random networks. Among all algorithms that do provide upper bounds (i.e., T-DLNS, BMS, KOPT2, and KOPT3), T-DLNS find the highest values. Additionally, T-DLNS also provides the best approximation ratios among all such algorithms. However, this comes at a cost of increased runtimes compared to the other algorithms. The quality of the solutions reported by DSA exceeds, albeit slightly, that of T-DLNS. However, DSA provides no quality guarantees. In general, all algorithms that do provide upper bounds have larger runtimes than those that do not provide these bounds. This behavior is not surprising since these algorithms require additional computation to compute and provide these bounds.

Table 2 tabulates the results for grid networks. These results are similar to the ones on random networks. Additionally, T-DLNS also outperforms DSA in finding solutions of high qualities (i.e., large LBs). However, this dominance comes at a price: T-DLNS has the largest runtime, on average, among all algorithms. Also, whereas DSA is the fastest algorithm for solving random networks, MS is the fastest for solving grid networks. This is due to that the computational time for MS is exponential in the arity of each variable and in grid networks each variable has significantly fewer neighbors than in random networks.

Finally, Table 3 tabulates the results for scale-free networks. The trend in this topology is similar to those in random and grid networks: T-DLNS provides better lower and upper bounds and, consequently, better approximation ratios compared to all other algorithms.

While T-DLNS does have larger runtimes than its competitors, it consistently outperforms state-of-the-art incomplete DCOP algorithms that do provide error

bounds: It finds both higher solutions’ qualities and tighter upper bounds. In the majority of the cases, it also outperforms state-of-the-art incomplete DCOP algorithms that do not provide error bounds.

Distributed Meeting Scheduling. Next, we evaluate the ability of T-DLNS to exploit the domain knowledge over *distributed meeting scheduling problems*. In such problems, one wishes to schedule a set of events within a time range. We use the *event as variable* formulation [21], where events are modeled as decision variables. Meeting participants can attend different meetings and have time preferences that are taken into account in the problem formulation. Each variable can take on a value from the interval $[0, 100]$. The problem requires that no meetings sharing some participants can overlap. We generate the underlying constraint network using the random network model described earlier. Our analysis focuses on compare T-DLNS using a *random* (RN) destroy and a *domain-knowledge* (DK) destroy strategies. The former randomly selects a set of variables to destroy, while the latter destroys the set of variables that are in overlapping meetings. Table 4 reports the percentage of satisfied instances reported (% SAT) and the time needed to find the first satisfiable solution (TF), averaged over 50 runs. The *domain-knowledge* destroy has a clear advantage over the *random* one, being able to effectively exploit domain knowledge. All other local search algorithms tested failed to report satisfiable solutions for any of the problems—only KOPT3 was able to find some solutions for problem instances with 20 meetings.

Thus, our experiments suggest that DLNS can bring a decisive advantage on both general and domain-specific problems, where exploiting structure can be done explicitly within the destroy phase.

Table 4. Experimental results on *meeting scheduling*.

Meetings	20		50		100	
	% SAT	t (ms)	% SAT	t (ms)	% SAT	t (ms)
DK destroy	80.05	78	54.11	342	31.20	718
RN destroy	12.45	648	1.00	52207	0.00	–
KOPT3	4.30	110367	0.00	–	–	–

8 Conclusions

In this paper, we proposed a *Distributed Large Neighborhood Search* (DLNS) framework to find quality-bounded solutions in DCOPs. DLNS is composed of a destroy phase, which selects a neighborhood to search, and a repair phase, which performs the search over such neighborhood. Within DLNS, we proposed a novel distributed algorithm, T-DLNS, characterized by low network usage and low computational complexity per agent. Our experimental results showed that T-DLNS finds better solutions compared to representative search-, inference-, and region-optimal-based incomplete DCOP algorithms. The proposed results

are significant—the anytime property and the ability to refine online quality guarantees makes DLNS-based algorithms good candidates to solve a wide class of DCOP problems. We strongly believe that this framework has the potential to solve very large distributed constraint optimization problems, with thousands of agents, variables, and constraints. In the future, we plan to investigate other schemes to incorporate into the repair phase of DLNS, including constraint propagation techniques [3, 7, 16] to better prune the search space, techniques that actively exploit the bounds reported during the iterative procedure, as well as the use of general purpose graphics processing units to parallelize the search for better speedups [4, 9], especially when agents have large local subproblems.

Acknowledgments. The research at the Washington University in St. Louis was supported by the National Science Foundation (NSF) under grant numbers 1550662 and 1540168. The research at New Mexico State University was supported by the NSF under grant numbers 1458595 and 1345232. The views and conclusions contained in this document are those of the authors only.

References

1. Auer, P.: Using confidence bounds for exploitation-exploration trade-offs. *J. Mach. Learn. Res.* **3**, 397–422 (2002)
2. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. *Science* **286**(5439), 509–512 (1999)
3. Bessiere, C., Gutierrez, P., Meseguer, P.: Including soft global constraints in DCOPs. In: Milano, M. (ed.) *CP 2012*. LNCS, pp. 175–190. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33558-7_15
4. Campeotto, F., Dovier, A., Fioretto, F., Pontelli, E.: A GPU implementation of large neighborhood search for solving constraint optimization problems. In: *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pp. 189–194 (2014)
5. Farinelli, A., Rogers, A., Petcu, A., Jennings, N.: Decentralised coordination of low-power embedded devices using the max-sum algorithm. In: *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 639–646 (2008)
6. Fioretto, F., Campeotto, F., Dovier, A., Pontelli, E., Yeoh, W.: Large neighborhood search with quality guarantees for distributed constraint optimization problems. In: *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1835–1836 (2015)
7. Fioretto, F., Le, T., Yeoh, W., Pontelli, E., Son, T.C.: Improving DPOP with branch consistency for solving distributed constraint optimization problems. In: O’Sullivan, B. (ed.) *CP 2014*. LNCS, vol. 8656, pp. 307–323. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_24
8. Fioretto, F., Pontelli, E., Yeoh, W.: Distributed constraint optimization problems and applications: a survey. *J. Artif. Intell. Res.* **61**, 623–698 (2018)
9. Fioretto, F., Pontelli, E., Yeoh, W., Dechter, R.: Accelerating exact and approximate inference for (distributed) discrete optimization with GPUs. *Constraints* **23**(1), 1–43 (2018)

10. Fioretto, F., Yeoh, W., Pontelli, E.: A dynamic programming-based MCMC framework for solving DCOPs with GPUs. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP), pp. 813–831 (2016)
11. Fioretto, F., Yeoh, W., Pontelli, E.: Multi-variable agents decomposition for DCOPs. In: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 2480–2486 (2016)
12. Fioretto, F., Yeoh, W., Pontelli, E.: A multiagent system approach to scheduling devices in smart homes. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 981–989 (2017)
13. Fioretto, F., Yeoh, W., Pontelli, E., Ma, Y., Ranade, S.: A DCOP approach to the economic dispatch with demand response. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 999–1007 (2017)
14. Geman, S., Geman, D.: Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intell.* **6**(6), 721–741 (1984)
15. Godard, D., Laborie, P., Nuijten, W.: Randomized large neighborhood search for cumulative scheduling. In: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), vol. 5, pp. 81–89 (2005)
16. Gutierrez, P., Lee, J.H.M., Lei, K.M., Mak, T.W.K., Meseguer, P.: Maintaining Soft Arc Consistencies in BnB-ADOPT⁺ during Search. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP), pp. 365–380 (2013)
17. Kiekintveld, C., Yin, Z., Kumar, A., Tambe, M.: Asynchronous algorithms for approximate distributed constraint optimization with quality bounds. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 133–140 (2010)
18. Kluegel, W., Iqbal, M.A., Fioretto, F., Yeoh, W., Pontelli, E.: A realistic dataset for the smart home device scheduling problem for DCOPs. In: Sukthankar, G., Rodriguez-Aguilar, J.A. (eds.) AAMAS 2017. LNCS (LNAI), vol. 10643, pp. 125–142. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71679-4_9
19. Kumar, A., Faltings, B., Petcu, A.: Distributed constraint optimization with structured resource constraints. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 923–930 (2009)
20. Léauté, T., Ottens, B., Szymanek, R.: FRODO 2.0: an open-source framework for distributed constraint optimization. In: International Workshop on Distributed Constraint Reasoning (DCR), pp. 160–164 (2009)
21. Maheswaran, R., Pearce, J., Tambe, M.: Distributed algorithms for DCOP: a graphical game-based approach. In: Proceedings of the Conference on Parallel and Distributed Computing Systems (PDCS), pp. 432–439 (2004)
22. Maheswaran, R., Tambe, M., Bowring, E., Pearce, J., Varakantham, P.: Taking DCOP to the real world: efficient complete solutions for distributed event scheduling. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 310–317 (2004)
23. Miller, S., Ramchurn, S., Rogers, A.: Optimal decentralised dispatch of embedded generation in the smart grid. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 281–288 (2012)
24. Modi, P., Shen, W.M., Tambe, M., Yokoo, M.: ADOPT: asynchronous distributed constraint optimization with quality guarantees. *Artif. Intell.* **161**(1–2), 149–180 (2005)

25. Nguyen, D.T., Yeoh, W., Lau, H.C.: Distributed Gibbs: a memory-bounded sampling-based DCOP algorithm. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 167–174 (2013)
26. Okimoto, T., Joe, Y., Iwasaki, A., Yokoo, M., Faltings, B.: Pseudo-tree-based incomplete algorithm for distributed constraint optimization with quality bounds. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 660–674. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23786-7_50
27. Ottens, B., Dimitrakakis, C., Faltings, B.: DUCT: an upper confidence bound approach to distributed constraint optimization problems. In: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 528–534 (2012)
28. Pearce, J., Tambe, M.: Quality guarantees on k-optimal solutions for distributed constraint optimization problems. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 1446–1451 (2007)
29. Petcu, A., Faltings, B.: Approximations in distributed optimization. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 802–806. Springer, Heidelberg (2005). https://doi.org/10.1007/11564751_68
30. Petcu, A., Faltings, B.: A scalable method for multiagent constraint optimization. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 1413–1420 (2005)
31. Petcu, A., Faltings, B.: A hybrid of inference and local search for distributed combinatorial optimization. In: Proceedings of the International Conference on Intelligent Agent Technology (IAT), pp. 342–348 (2007)
32. Rogers, A., Farinelli, A., Stranders, R., Jennings, N.: Bounded approximate decentralised coordination via the max-sum algorithm. *Artif. Intell.* **175**(2), 730–759 (2011)
33. Rollon, E., Larrosa, J.: Improved bounded max-sum for distributed constraint optimization. In: Milano, M. (ed.) CP 2012. LNCS, pp. 624–632. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33558-7_45
34. Rust, P., Picard, G., Ramparany, F.: Using message-passing dcop algorithms to solve energy-efficient smart environment configuration problems. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 468–474 (2016)
35. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M., Puget, J.-F. (eds.) CP 1998. LNCS, vol. 1520, pp. 417–431. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49481-2_30
36. Sultanik, E., Modi, P.J., Regli, W.C.: On modeling multiagent task scheduling as a distributed constraint optimization problem. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 1531–1536 (2007)
37. Ueda, S., Iwasaki, A., Yokoo, M.: Coalition structure generation based on distributed constraint optimization. In: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), pp. 197–203 (2010)
38. Vinyals, M., et al.: Quality guarantees for region optimal DCOP algorithms. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 133–140 (2011)
39. Yeoh, W., Felner, A., Koenig, S.: BnB-ADOPT: an asynchronous branch-and-bound DCOP algorithm. *J. Artif. Intell. Res.* **38**, 85–133 (2010)
40. Yeoh, W., Sun, X., Koenig, S.: Trading off solution quality for faster computation in DCOP search algorithms. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 354–360 (2009)
41. Yeoh, W., Yokoo, M.: Distributed problem solving. *AI Mag.* **33**(3), 53–65 (2012)

42. Zhang, W., Wang, G., Xing, Z., Wittenberg, L.: Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. *Artif. Intell.* **161**(1–2), 55–87 (2005)
43. Zivan, R., Yedidsion, H., Okamoto, S., Grinton, R., Sycara, K.: Distributed constraint optimization for teams of mobile sensing agents. *J. Auton. Agents Multi Agent Syst.* **29**(3), 495–536 (2015)



Distributed Constrained Search by Selfish Agents for Efficient Equilibria

Vadim Levit and Amnon Meisels(✉)

Department of Computer Science, Ben-Gurion University, Beersheba, Israel
am@cs.bgu.ac.il

Abstract. Search for stable solutions in games is a hard problem that includes two families of constraints. The global stability constraint and multiple soft constraints that express preferences for socially, or otherwise, preferred solutions. To find stable solutions (e.g., pure Nash equilibria - PNEs) of high efficiency, the multiple agents of the game perform a distributed search on an asymmetric distributed constraints optimization problem (ADCOP). Approximate (local) distributed search on ADCOPs does not necessarily guarantee convergence to an outcome that satisfies the stability constraints, as well as optimizes the soft constraints. The present paper proposes a distributed search algorithm that uses transfer of funds among selfish agents. The final outcome of the algorithm can be stabilized by transfer of funds among the agents, where the transfer function is contracted among the agents during search. It is shown that the proposed algorithm - Iterative Nash Efficiency enhancement Algorithm (INEA) - guarantees improved efficiency for any initial outcome.

The proposed distributed search algorithm can be looked at as an extension to best response dynamics, that uses transfer functions to guarantee convergence and enforce stability in games. The best-response-like nature of INEA establishes its correct behavior for selfish agents in a multi-agents game environment. Most important, unlike best response, the proposed INEA converges to efficient and stable outcomes even in games that are not potential games.

1 Introduction

A common solution concept for games played by multiple agents is the Nash equilibrium, i.e, a stable state in which no participant can gain by a unilateral change of strategy (cf. [1]). Finding a PNE in a general game and in particular an efficient one forms a constrained search problem, where stability is a global hard constraint and efficiency is composed of soft constraints for which a good solution (e.g., of higher social welfare) is sought (cf. [2]). The problem of finding a Nash equilibrium has been shown to be computationally intensive [1, 3] and heuristic search algorithms for this constrained problem have been recently proposed [4, 5]. Additionally, it is known that equilibria states are not necessarily efficient with respect to a global objective [3, 6] or even with respect to the personal payoffs of the agents [7].

The distributed nature of games, played by multiple agents, requires distributed search methods to find solutions. Several distributed complete search algorithms for finding stable states in games were proposed [8–10]. However, for games among a large number of agents the problem of finding a PNE becomes computationally intractable [1] and one needs to rely on approximate search which can find a PNE in reasonable time (cf. [2]). Complete distributed search assumes cooperation among the participating agents [11], but this assumption is not always acceptable for selfish agents. One simple way of avoiding the assumption of cooperation among the searching agents is to apply specially designed local search. However, the major problem of approximate (local) search is its ability to converge, as well as guarantee that the resulting solution will be stable.

On a different theme, approximate search methods were shown to converge to a stable state in *Potential games* [12], where best-response dynamics converge to a PNE. When an approximate search method like best response converges to a stable solution, the efficiency of the solution can be arbitrarily low. In contrast, it is well known that outcomes of high efficiency are not necessarily stable (e.g. “the prisoners’ dilemma”). This tradeoff between efficiency and stability motivates the use of an incentive mechanism to promote stability in efficient states. Jackson and Wilkie [13] have studied the efficiency of equilibria that are achieved by using *side payments*. The side payments mechanism enables to incentivize unsatisfied agents to agree on some preferred outcomes. Side payments allow transfers of funds (payoffs) between agents, such that agents who gain from some outcome may want to pay others that are unsatisfied by it.

The present paper proposes an incomplete decentralized search algorithm for selfish agents, to find a stable state of higher efficiency. One can view the proposed Iterative Nash Efficiency-enhancement Algorithm (INEA) as an enhancement to the best-response dynamics (cf. [12]). The innovation of INEA is that it enables participating agents to sacrifice part of their payoff at a certain outcome in order to convince other agents to play a certain strategy. In this sense, the inter-agent transfers of funds are used in order to ensure the convergence of the algorithm to outcomes which are both stable and efficient. The outcome resulting from the run of the INEA algorithm is ensured to be at least as efficient as the original outcome. Most importantly, it is guaranteed that the resulting outcome can be transformed into a stable state by the use of the transfers of funds that were contracted by the agents during the run of the algorithm.

The proposed INEA algorithm uses a fixed order of all the agents. Each agent in its turn exchanges messages with its neighbors and decides on its selected strategy. The messages may include proposed transfers of funds among the interacting agents. The game is assumed to have interactions among a limited number of agents at each step, which is equivalent to assuming some underlying graph or social network. This compact general structure for multi-agents games has been termed graphical games in the past [14] and they are known to have a compact representation. For clarity, the term multi-agents games (MAGs) will be used through the rest of the paper. Since the distributed search algorithm proposed

by the present paper uses strategic reasoning by the agents that compute their transfers of payoffs, transfer functions can be thought of as *side payments* [13]. Side payments that are contracted by the agents during the run of the algorithm, to be transferred for its final outcome [13, 15, 16].

An extensive experimental evaluation of the INEA algorithm demonstrates two main results. First, that for multi-agents games that are also potential games, the proposed algorithm produces stable states that are more efficient than best response and in fact that the resulting stable states are close to optimal. Second, that for randomly generated MAGs the fairness of the efficient stable states that are produced by the INEA algorithm is highly affected by the nature of the transfer function used.

Section **Preliminaries** introduces the needed concepts of games and their stable states, pure Nash equilibria (PNEs). It emphasizes the equivalence of multi-agents games with distributed constraints search. **Preliminaries** includes the definitions of transfers of funds among agents as well as the definition of outcomes which can be stabilized by the use of side payments. The proposed Iterative Nash Efficiency enhancement Algorithm is described in detail in section **Finding Efficient Equilibria** together with proofs for the main guarantees of the algorithm. A detailed example of the run of the proposed algorithm on an example problem is also provided. The following section presents the experimental evaluation of INEA. First, the INEA algorithm is compared to best-response on problems that are potential games. Next, two forms of transfer functions are compared on randomly generated problems that are not necessarily potential games.

2 Preliminaries

2.1 Multi-agents Games

Multi-agents games (MAGs) consist of a set of agents $A = \{1, \dots, n\}$, where each agent i has a set of strategies V_i from which it can select a strategy $v_i \in V_i$ to play. In the terminology of distributed constraints problems, strategies are *assignments* [8] and the games among agents are equivalent to asymmetric constraints [9, 11]. An *outcome* of the multi-agents game, $v = (v_1, \dots, v_n) \in V_1 \times \dots \times V_n$, is a collection of strategies (e.g., assignments), one for each agent. Each agent i interacts (plays) with some set of other agents, which we term N_i (e.g., its neighborhood in graphical games terms [14]) and can be a small subset of A . Agent i 's payoff function is denoted $u_i : \times_{j \in N_i \cup i} V_j \rightarrow \mathbb{R}$, where N_i is usually limited to a small subset of A .

Multi-agents games that have agents interacting with a small numbers of “neighboring” agents are commonly termed graphical games and have a compact representation [14]. The agents can be thought of as if they are connected by some underlying graph (e.g., social network) $G = \{N, E\}$. Each vertex in N represents an agent and edges in E represent the interaction structure of the game. As stated above, game-like interactions are asymmetric constraints. Given an agent $i \in N$, the set of i 's neighbors N_i are the agents whose actions impact

i 's payoff [17]. Note that the classical Normal-form game can be represented as a MAG having a complete underlying graph. The assumption of a small size of N_i is used in order to improve the run-time of the computation of the transfers of payoffs and to simplify the presentation of the proposed distributed search algorithm.

The *pure Nash Equilibrium (PNE)* is a central concept in game theory (cf. [1,3]). We say that an outcome is a PNE, if every agent does not prefer to change its strategy (assignment), given the strategies (assignments) of all other agents in this outcome. Formally, an outcome v is a PNE if the following holds:

$$\forall i \in N, \nexists v'_i \in V_i \text{ s.t., } u_i(v_i, v_{-i}) < u_i(v'_i, v_{-i}) \quad (1)$$

where v_{-i} is the standard notation for the combined strategies (assignments) of all agents except i .

Finding a PNE in the above description of a MAG is equivalent to solving an Asymmetric Constraints Optimization Problem (ADCOP) [11], where the agents' domains of values are the palettes of strategies of agents in the MAG and the constraints among neighbors of an ADCOP are represented by the values in the normal form game matrix (cf. [8,16,18]). Stability is represented by the above global constraint of the ADCOP. Once an ADCOP is constructed, the finding of an efficient PNE can be solved by an appropriate algorithm (cf. [8,19]). The present paper focuses on local search algorithms that can be run by rational agents, emphasizing a distributed search protocol that bears similarities to the game theoretic best-response dynamics [12].

2.2 Transfer Functions and Side Payments

Transfer functions endow agents with the possibility of sacrificing part of their payoff in order to convince other agents to play a certain strategy. Given a multi-agents game, transfers of payoff are defined by a function $\tau : N \times N \times V_1 \times \dots \times V_n \rightarrow \mathbb{R}^+$, where $\tau_{i,j}(v)$ denotes the payment being transferred from agent i to agent j in outcome v . In order to take transfer functions into consideration while deciding on the action to take, an agent's utility must reflect the change in its payoff. The *net loss* that is incurred on agent i at outcome v when using the transfer function τ is defined by Eq. 2.

$$\tau_i(v) := \sum_{j \in N_i} (\tau_{i,j}(v) - \tau_{j,i}(v)) \quad (2)$$

We restrict our attention to transfer functions such that if $j \notin N_i$ then $\forall v : \tau_{i,j}(v) = 0$. In words, side payments take place between neighbors in the game. Note again that neighbors are connected by a two-players game which is an asymmetric binary constraint. Given the definition of net loss (Eq. 2), one can update the definition of the utility that agent i obtains from outcome v as follows:

$$u_i^\tau(v) := u_i(v) - \tau_i(v) \quad (3)$$

Transfer functions have been termed *Side payments* by Jackson and Wilkie [13] when their values are decided upon by the agents that strategically reason about their values. Side payments enable games to be transformed from the inside and are defined as transfers of payoffs between agents, where each agent may pay (or receive payment from) each one of its neighbors. Since the distributed search algorithm proposed by the present paper uses strategic reasoning by the agents that compute their offered and accepted transfers of payoffs, we will use the terms transfer functions and side payments interchangeably.

The use of transfer functions in the present study is mainly motivated by the goal to obtain a stable state (PNE) with certain properties. Let us start by differentiating between outcomes that can be transformed into a PNE and those that cannot.

Definition 1. *An outcome v in a multi-agents game G is side payments enforceable (SPE) if there exists a transfer function τ , such that:*

$$\forall i \in N, \nexists v'_i \in V_i \text{ s.t., } u_i^\tau(v_i, v_{-i}) < u_i^\tau(v'_i, v_{-i})$$

Definition 1 ensures that the updated agents utilities (according to the transfer function τ) result in a PNE at outcome v , satisfying Eq. 1.

The intuitive argument for how transfers of payoffs have the ability to obtain stable outcomes goes as follows: An agent can offer a neighboring agent compensation which is a function of the second agent’s action, such that the compensation effectively reflects any utility loss that the second agent’s action incurs on the first agent. Take for example the following case. The utility loss of agent i is x if agent j takes action v'_j rather than its current action v_j , while the benefit of agent j of taking action v'_j rather than v_j is only y where $x > y$. Agent i in this example can offer to agent j a monetary compensation of z , where $x \geq z \geq y$, if agent j will not deviate from its current strategy v_j . Note that any z such that $x \geq z \geq y$ will provide a sufficient incentive for agent j .

3 Finding Efficient Equilibria

The proposed distributed search algorithm, Iterative Nash Efficiency enhancement Algorithm (INEA), relies on inter-agent transfers of payoffs which is a key element in the proposed procedure. The INEA search algorithm is iterative, where each iteration goes over all agents in a predefined fixed order. Each agent in its turn proposes its selected strategy by sending messages to its neighbors and receiving from them response messages if they wish to transfer funds to the proposing agent in order to convince it to avoid taking the proposed strategy. The algorithm is proven to converge to an SPE global state with improved (or at least the same) global efficiency. In addition, this distributed method may be complemented by existing heuristics that find initial outcomes of high efficiency. The resulting combination (which is not pursued in the present study) has the potential to produce in its end result PNEs of even higher efficiency.

3.1 The INEA Algorithm

The INEA algorithm is composed of two consecutive stages. Starting from an initial outcome v , the main stage iterates over all agents (in a predefined fixed order) until it converges to an outcome v^* . The efficiency of outcome v^* is proven to be at least as high as the efficiency of the initial outcome v . Outcome v^* is not necessarily a stable one, but is guaranteed to be side payments enforceable (SPE). A set of transfers that can be used as side payments among the agents to enforce a PNE is computed by the agents during the iterations and is guaranteed to not worsen the utility of all agents. It therefore can be thought of as a binding contract among the agents during the iterative search, computed by the neighboring (e.g., interacting) agents during each step. In that sense one can call them side payments [13] and think of the proposed INEA algorithm as an iterative method for searching for both an efficient outcome and the side payments that can guarantee its stability. Each iteration ends with an outcome and side payments that were computed during the iteration. The final outcome is reached when for a complete iteration over all agents no agent wishes to change its strategy. Exchanging messages with neighbors on the graph of the game makes the INEA algorithm a distributed local search algorithm (cf. [20–22]). It is important to understand that the major problem of standard local (approximate) search in DCOPs and ADCOPs is its ability to converge, as well as guarantee that the resulting solution will be stable [20, 21].

In the second stage of the run of the proposed INEA algorithm the transfers of payoffs among the agents are applied according to the computed contracts so that the outcome v^* is transformed into a stable state (i.e., a PNE). One can think of the proposed INEA algorithm as an iterative computation of the final outcome and the side payments that guarantee its stability, where all computations are performed by the agents in a distributed manner.

Main Stage

One can view the INEA algorithm (Algorithm 1) as an extended version of the best-response dynamics, extended with payoff transfers (contracts) among selfish agents. During the iterative run of the algorithm the participating agents decide on their transfer contracts to the other agents as well as the outcome which can be stabilized by applying these contracts.

All agents perform the main stage in a predefined fixed order, and each agent in turn executes the function `onStrategySelect()`. We will term the agent performing function `onStrategySelect()` the *current agent*. During the execution of Algorithm 1 the agents agree on contracts with their neighbors which are the incentive to the agents to remain with their choices (i.e., not change their strategy to the proposed one). Therefore, in order to decide regarding the choice of action, the current agent must take into consideration not only its own utility but also the compensation (contracts) from its neighbors as can be observed from lines 1–4 of Algorithm 1. Contracts are dissolved only due to the decision of an agent to change its strategy. Consequently, if an agent notifies its neighbors regarding the desire to change its strategy (line 5), all contracts

concluded between its neighbors and itself are considered to be nullified. As a response to the message from the current agent, proposing to change its strategy, each neighbor decides upon the payoff transfer it wishes to sacrifice in order to convince the current agent to remain with its choice. These choices are sent back to the current agent and considered as new contracts only in case the agent remains with its choice. When the current agent receives responses from all of its neighbors, the agent will not choose the strategy that improves its utility only in the case that the agent’s neighbors can secure for it a larger payoff (lines 7–8). This is a completely selfish (e.g., rational) decision when taking into consideration transfer functions.

Algorithm 1. Iterative Nash efficiency enhancement algorithm

```

onStrategySelect()
1: let  $\mathbb{T} \leftarrow \sum_{j \in N_i} T_{j,i}$ 
2: if  $\nexists v'_i$  s.t.  $u_i(v'_i, v_{-i}) > u_i(v_i, v_{-i}) + \mathbb{T}$  then
3:   return
4: select  $v'_i$  s.t.  $u_i(v'_i, v_{-i}) > u_i(v_i, v_{-i}) + \mathbb{T}$ 
5: send(choice,  $v'_i$ ) to all  $j \in N_i$ 
6: let  $\mathbb{T}'$  be the sum of replies from all  $j \in N_i$ 
7: if  $u_i(v'_i, v_{-i}) > u_i(v_i, v_{-i}) + \mathbb{T}'$  then
8:   update  $v \leftarrow (v'_i, v_{-i})$ 

when received(choice,  $v'_j$ )
9: chose  $T_{i,j}$ 
10: reply  $T_{i,j}$ 

```

Algorithm 1 defines the communication protocol among agents which enables them to decide about the transfers of payoff that can be used to stabilize the final outcome. Nevertheless, INEA does not explicitly specify how the decision about the exact monetary transfer is made (line 9). Such a decision can have a great impact on the ability of the contracts to transform the outcome into a stable one. We will restrict our attention only to *admissible contracts* (defined below).

Definition 2. *Given an outcome v and a beneficial deviation v'_i of the current agent i , the contracts $T_{j,i}$ are admissible if the following conditions hold:*

1. $\forall j \in N_i, 0 \leq T_{j,i} \leq u_j(v'_i, v_{-i}) - u_j(v_i, v_{-i})$
2. *if* $\sum_{j \in N_i, u_j(v_i, v_{-i}) > u_j(v'_i, v_{-i})} u_j(v_i, v_{-i}) - u_j(v'_i, v_{-i}) \geq u_i(v'_i, v_{-i}) - u_i(v_i, v_{-i})$ *then* $\sum_{j \in N_i} T_{j,i} \geq u_i(v'_i, v_{-i}) - u_i(v_i, v_{-i})$

Condition 1 restricts the proposed compensations to be “acceptable” by rational agents. It will be greater than zero only in the case that i ’s proposed change of strategy will incur a negative utility change for the responding neighbor.

Additionally, an agent will not propose a transfer of payoff that exceeds its loss of utility. Condition 2 ensures that if the sum of the negative utility changes to neighbors exceeds the benefit of the current agent from its proposed unilateral deviation, so should the compensations. In other words, the proposed side-payments reflect the loss to the neighbors (from the proposed change of strategy of the current agent) and is therefore large enough to compensate for them.

Note that the contracts among agents do not affect the social welfare of outcomes. Therefore, one can omit the payoff transfers (i.e., take into consideration only the “original” utilities of the agents) when computing the social welfare. However, it is necessary to take the payoff transfers into consideration when reasoning about the stability of an outcome.

3.2 Correctness

Proposition 1. *Every update of an agent’s strategy, performed by following Algorithm 1, improves the social welfare of the outcome v .*

Proof. According to Algorithm 1 only a current agent i which can improve its payoff by deviating from strategy v_i to v'_i can update its strategy. Such an update occurs only in the case where the sum of transfers offered by the responses of i ’s neighbors is smaller than the improvement in i ’s utility resulting from the strategy change (lines 7–8).

Suppose by contradiction that the current agent i updated its strategy from v_i to v'_i (running Algorithm 1) which improves i ’s utility but decreases the global social welfare. Clearly, all transfers do not change the social welfare because they represent only movement of funds from one agent to another. Consequently, the change in social welfare for the above case can be defined irrespective of the transfers as follows:

$$\sum_{j \in N} u_j(v'_i, v_{-i}) - \sum_{j \in N} u_j(v_i, v_{-i}) \tag{4}$$

and will affect only the utilities of agent i and its neighbors. Therefore, Eq. 4 can be simplified to:

$$(u_i(v'_i, v_{-i}) - u_i(v_i, v_{-i})) + \sum_{j \in N_i} (u_j(v'_i, v_{-i}) - u_j(v_i, v_{-i})) \tag{5}$$

Since the social welfare was decreased by deviation from v_i to v'_i , Eq. 6 must hold.

$$\begin{aligned} u_i(v'_i, v_{-i}) - u_i(v_i, v_{-i}) &< \sum_{j \in N_i} (u_j(v_i, v_{-i}) - u_j(v'_i, v_{-i})) \\ &\leq \sum_{j \in N_i, u_j(v_i, v_{-i}) > u_j(v'_i, v_{-i})} (u_j(v_i, v_{-i}) - u_j(v'_i, v_{-i})) \end{aligned} \tag{6}$$

Since only admissible contracts are considered, condition 2 of the admissible contracts requires that $\sum_{j \in N_i} T_{j,i} \geq u_i(v'_i, v_{-i}) - u_i(v_i, v_{-i})$. Consequently,

$u_i(v'_i, v_{-i}) \leq u_i(v_i, v_{-i}) + \sum_{j \in N_i} T_{j,i}$ holds. As a result, the decrement in social welfare results in a contradiction to the condition of line 7 of Algorithm 1 and cannot be true.

Corollary 2. *Algorithm 1 converges in a finite number of steps.*

The correctness of Corollary 2 follows directly from Proposition 1. Since each update of an agent’s strategy according to Algorithm 1 improves the Social welfare of the outcome v and the Social welfare is bounded, the proposed algorithm terminates in a finite number of steps.

Corollary 3. *For every outcome $v \in V$, the social welfare of v^* obtained by running Algorithm 1 is greater than or equal to the social welfare of v .*

Since Proposition 1 ensures that the social welfare of outcome v is non-decreasing (at each step), then so is the social welfare of the final outcome v^* , which is at least as high as the social welfare of v .

Observation 4. *A graphical game updated by admissible contracts is an ordinal potential game where the potential function is the social welfare.*

The correctness of Proposition 1 directly leads to Observation 4. Namely, if an agent can change its strategy so as to increase its utility then the social welfare will be improved. Note that the opposite direction does not necessarily hold, i.e., a strategy change which decreases the agent’s payoff does not necessarily result in a decrease of the overall social welfare.

In order to get some intuition about the differences between best-response dynamics and Algorithm 1, consider the following example:

Example 1. *Three agents are connected by edges, each representing a two-players game (e.g, an asymmetric constraint) on the graph in Fig. 1. Every agent has exactly two strategies, a or b. The utilities of agents are the sum of payoffs of the two-player games (constraints) in Fig. 1.*

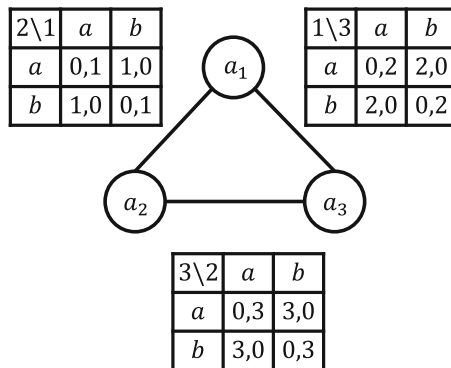


Fig. 1. An example game with three agents

Table 1. The utilities of agents a_1 , a_2 and a_3 in the example game.

v_1	v_2	v_3	u_1	u_2	u_3
a	a	a	1	3	2
a	a	b	3	0	3
a	b	a	0	1	5
a	b	b	2	4	0
b	a	a	2	4	0
b	a	b	0	1	5
b	b	a	3	0	3
b	b	b	1	3	2

Consider an outcome $v = (v_1 = a, v_2 = a, v_3 = a)$, the payoff of agent a_1 is 1 in the game with a_2 and 0 in the game with a_3 , leading to a total utility of 1 for agent a_1 in outcome v . The utility of agent a_2 is 0 in the game/constraint with a_1 and 3 against a_3 , summing up to a total utility of 3 for a_2 in v . Finally, the utility of a_3 in outcome v is 2 since its payoff is 2 in the two-player game/constraint against a_1 and 0 in the game against a_2 .

Let us start by following the run of the best response algorithm, where agents are ordered according to their indexes and the initial outcome is $v = (v_1 = a, v_2 = a, v_3 = a)$. To simplify the following of the agents reasoning in both algorithms, Table 1 presents the utilities of every agent in all 8 possible outcomes of the example game. Best response for agent a_1 is the unilateral deviation to strategy b which will increase its utility to 2. In the resulting outcome agent a_2 has no strategy which can improve its utility. Agent a_3 can change its strategy to b which will result in outcome $(v_1 = b, v_2 = a, v_3 = b)$. Now a_1 can improve its utility by changing to $v_1 = a$ which will be followed by agent a_2 's move to strategy $v_2 = b$ and then agent a_3 will switch its strategy to a. Having arrived at the outcome $(v_1 = a, v_2 = b, v_3 = a)$, agent a_1 will benefit from deviating to $v_1 = b$. In the resulting outcome a_2 whose turn is next will change its strategy to a. It is easy to see that for our example game and outcome best-response dynamics has just closed a loop of states and does not converge.

Let us now follow the run of Algorithm 1 on the same game and outcome $v = (v_1 = a, v_2 = a, v_3 = a)$ and in the same order. The use of admissible contracts is assumed. Initially there are no transfers contracted and for all agents, $T_{i,j} = 0$.

As before, agent a_1 can improve its utility by deviating from a to b. Executing function **onStrategySelect()**, it will send message $\langle \text{choice}, v_1 = b \rangle$ to its neighbors a_2 and a_3 (line 5 of Algorithm 1). The benefit to a_1 from this change of strategy is 1 (i.e., $u_1(v_1 = a, v_{-1}) = 1$ and $u_1(v_1 = b, v_{-1}) = 2$) but it will reduce the utility of agent a_3 from 2 to 0. For the admissibility of the contracts, agent a_3 will select a transfer of $1 \leq T_{3,1} \leq 2$ as incentive to agent a_1 to remain in its current strategy (line 7). The payoff of a_2 increases from the proposed change of a_1 , so it does not respond at all. Since the outcome remains, a_2 has no strategy

which improves its utility and it does not propose any (line 2 and 3). Agent a_3 will benefit from unilateral deviation from $v_3 = a$ to $v_3 = b$. However, a_2 will propose a sufficient incentive ($2 \leq T_{2,3} \leq 3$) for a_3 to remain with its current choice. This will suffice for a_3 . No agent changed its strategy during this complete round and Algorithm 1 terminates with the stability enforceable outcome $v^* = (v_1 = a, v_2 = a, v_3 = a)$.

It is easy to see that the final outcome of Example 1 is not a stable state. Agent a_1 , for example, can change its strategy to b unilaterally and improve its gain from 1 to 2. However, this final state can be stabilized by using the transfers of funds that were contracted during the run of the algorithm as described above. In other words, it is guaranteed to be side payments enforceable. For example, agent a_3 can pay 1 to agent a_1 to secure its strategy and retain a stable state (a PNE).

Second Stage

When Algorithm 1 terminates there may be agents which have an incentive to unilaterally deviate from their strategy and improve their utility (similarly to Example 1). To stabilize outcome v^* the contracted transfers during the execution of Algorithm 1 need to be applied. The payoff transfers computed during the run of Algorithm 1 are as follows:

$$\tau_{j,i}(v) := \begin{cases} T_{j,i}, & \text{if } v_i = v_i^* \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

Proposition 5. *For every outcome $v \in V$, the outcome v^* of Algorithm 1 is side payments enforceable.*

Proof. The correctness of Proposition 5 follows directly from the termination condition of Algorithm 1. When Algorithm 1 terminates, the only agents which must be incentivized in order to stabilize outcome v^* are those which can gain from unilateral deviation.

Formally, given outcome v an agent i has an incentive to unilaterally deviate if there exists a choice v'_i for it, such that $u_i(v'_i, v_{-i}) > u_i(v_i, v_{-i})$. The termination of Algorithm 1 implies that for each agent i it holds that

$$\nexists v'_i, u_i(v'_i, v_{-i}) > u_i(v_i, v_{-i}) + \sum_{j \in N_i} T_{j,i}$$

which implies that

$$\exists v'_i, u_i(v'_i, v_{-i}) - \sum_{j \in N_i} T_{i,j} > u_i(v_i, v_{-i}) + \sum_{j \in N_i} (T_{j,i} - T_{i,j})$$

and consequently, for each agent i the following holds:

$$\exists v'_i, u_i^T(v'_i, v_{-i}) > u_i^T(v_i, v_{-i})$$

Observation 6. *An outcome v that maximizes social welfare is side payments enforceable.*

By Corollary 3, applying Algorithm 1 to an outcome v that maximizes social welfare will return the exact same outcome $v^* \equiv v$. By Proposition 5, this outcome is SPE.

The payoff transfer contracts defined by Algorithm 1 are sufficient in order to convince unsatisfied agents to stay in outcome v^* (Proposition 5). However, the INEA algorithm does not specify explicitly the actual exact transfer of payoff. It only assumes the admissibility of the contracts. Let us view two distinct admissible contract schemes which can be efficiently computed during the run of the INEA algorithm.

The first one does not make any assumption regarding the knowledge of agents about the utilities and the contracts of other agents. The agents also do not perform any negotiation with other agents in order to decide about the amount of payoffs.

Consider the maximal payoff transfer that agent j is willing to secure for its neighbor i in outcome v in order to prevent i 's change of strategy to v'_i :

$$\delta_{j,i}(v, v'_i) := \begin{cases} u_j(v_i, v_{-i}) - u_j(v'_i, v_{-i}), & \text{if } u_j(v_i, v_{-i}) > u_i(v'_i, v_{-i}) \\ 0, & \text{otherwise} \end{cases} \quad (8)$$

Using Eq. 8 admissible contracts are:

$$T_{j,i} = \delta_{j,i}(v, v'_i) \quad (9)$$

These contracts are termed *maximal payoff contracts* and their computation does not need interaction among agents.

Observation 7. *Maximal payoff contracts are admissible.*

The admissibility of the maximal payoff contracts arises directly from the definition. Agent j will sacrifice a payoff transfer of exactly $T_{j,i} = u_j(v_i, v_{-i}) - u_j(v'_i, v_{-i})$ to agent i only if it equals the decrement in its utility due to the unilateral deviation of agent i from v_i to v'_i .

Another contracts scheme - *cooperative contracts* - assumes full knowledge of the game and cooperation among the agents in a neighborhood and the contracts are defined as follows:

$$T_{j,i} := \begin{cases} 0, & \text{if } \sum_{l \in N_i} \delta_{l,i}(v, v'_i) < u_i(v'_i, v_{-i}) - u_i(v_i, v_{-i}) \\ \delta_{j,i}(v, v'_i) \cdot \frac{u_i(v'_i, v_{-i}) - u_i(v_i, v_{-i})}{\sum_{l \in N_i} \delta_{l,i}(v, v'_i)}, & \text{otherwise} \end{cases} \quad (10)$$

The size of *cooperative transfers* is proportional to the loss in the agent's utility, which is analogous to Shapley's value [23].

Observation 8. *Cooperative contracts are admissible.*

This follows immediately from the definition in Eq. 10. Non-zero transfers are only proposed if their sum compensates the deviating agent and no neighboring agent transfers more than its loss.

4 Experimental Evaluation

To empirically demonstrate the effectiveness of the INEA algorithm (Algorithm 1), two families of experiments were run. One family compares the performance of the INEA algorithm to best-response dynamics. Both algorithms converge to a stable point – best-response to a PNE and Algorithm 1 to an SPE. The other family of experiments studies the behavior of different admissible transfer functions. All the games that are used in the experimental evaluation fall under the description of *graphical games* [9,14], where interaction among players/agents is limited to a small fraction (i.e., a neighborhood) of the overall large number of players.

4.1 Problem Generation

The first set of experiments uses “best-shot” public goods games [17,24], which were proven to be potential games [18]. In “best-shot” public goods games each agent chooses an action $v_i \in \{0, 1\}$. For example, the action might be buying a book or some other product that is easily lent from one agent to another. Taking the action 1 is costly and an agent prefers a neighbor $j \in N_i$ taking the action than having to take it (N_i is i ’s neighborhood) But, taking the action and paying the cost is better than having nobody take the action. More formally, agent i ’s utility at outcome v is

$$u_i(v) := \begin{cases} 1 - c, & \text{if } v_i = 1 \\ 1, & \text{if } v_i = 0 \wedge \exists j \in N_i, v_j = 1 \\ 0, & \text{if } v_i = 0 \wedge \forall j \in N_i, v_j = 0 \end{cases} \quad (11)$$

where $0 < c < 1$ is the cost of taking the action (in all experiments $c = \frac{1}{2}$).

For the second set of experiments random games are used, that are not necessarily potential games. These random games were generated with 10 possible strategies for each agent and the utilities in the game matrices of each neighborhood are randomly chosen from a uniform distribution in the range $[0,1)$.

4.2 Experimental Results

The performance of the best response algorithm and of Algorithm 1 is measured by the social welfare of the final outcome. Social welfare is simply the sum of all utilities.

$$SW(v) = \sum_{i \in N} u_i(v)$$

The first set of experiments uses relatively small games of up to 20 agents in order to compare the performance of both best-response and INEA to the optimal efficiency of stable outcomes which have to be found by exhaustive search. Given our theoretical guarantees on the efficiency of the outcomes of INEA (Corollary 3), the gap from the most efficient outcomes can in principle be unbounded.

The average efficiency of the outcomes of INEA is higher than those obtained by best-response (Fig. 2). Moreover, the efficiency of the outcomes of INEA is similar and sometimes better than that of *PNEs of maximal social welfare*. In contrast, best-response converges to outcomes with social welfare that is only slightly better than that of PNEs which minimize the social welfare.

The second set of experiments is performed on large games that are not necessarily potential games and are randomly generated. These games are used to study the dependency of the outcomes of INEA on the transfer function used. An interesting measure is the *fairness* of the outcome (degree of inequality), measured by the *Gini Index*

$$GI(v) = \frac{\sum_{i \in N} \sum_{j \in N} |u_i(v) - u_j(v)|}{2 \sum_{i \in N} \sum_{j \in N} u_j(v)}$$

The fairness of the outcome is highly affected by the type of contracts that are used during search (Fig. 3). Outcomes obtained by maximal payoff contracts are less fair than those that use cooperative contracts. When using maximal payoff, multiple agents may secure a payoff transfer to a single change-proposing player which may result in a compensation for the player that is much higher

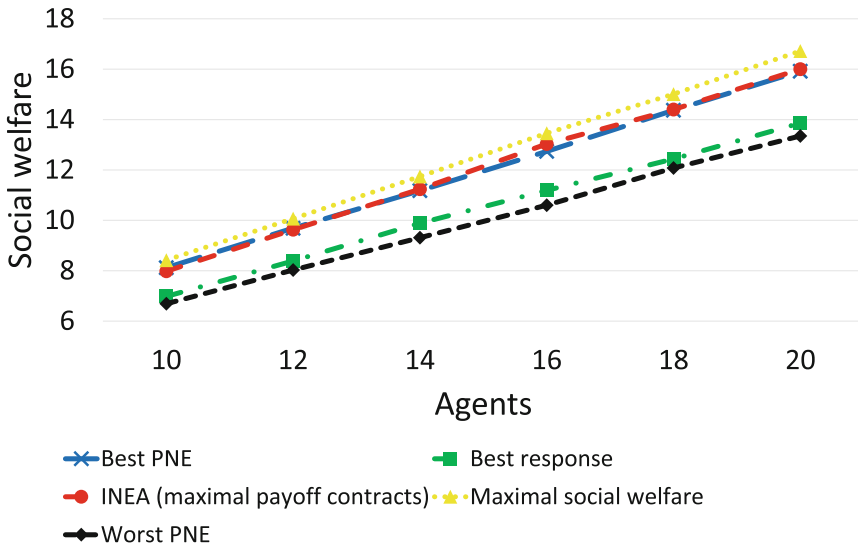


Fig. 2. Social welfare of “best-shot” public goods games

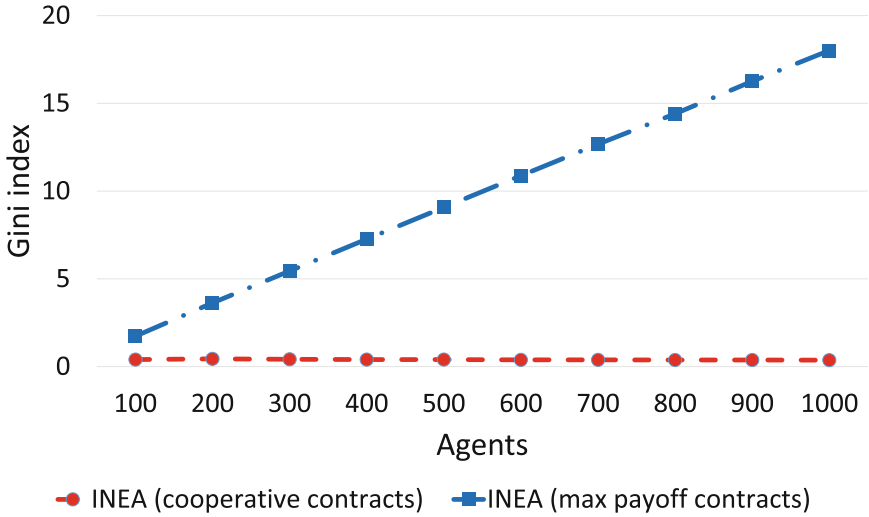


Fig. 3. Fairness of stable outcomes on random networks

than its benefit from unilateral deviation. This can lead to a deterioration in the fairness of the solution, as is evident by the increase of the Gini index with the size of the problem, in Fig. 3.

It is interesting to see that cooperative contracts are also able to provide outcomes of greater efficiency than maximal payoff contracts (Fig. 4). When using cooperative contracts the neighbors of a deviating agent may secure less payoff in order to convince it to stay with its current choice. Consequently, a greater improvement in social welfare is possible but needs more improvement steps. The percentage of improvement of social welfare over that of the initial state

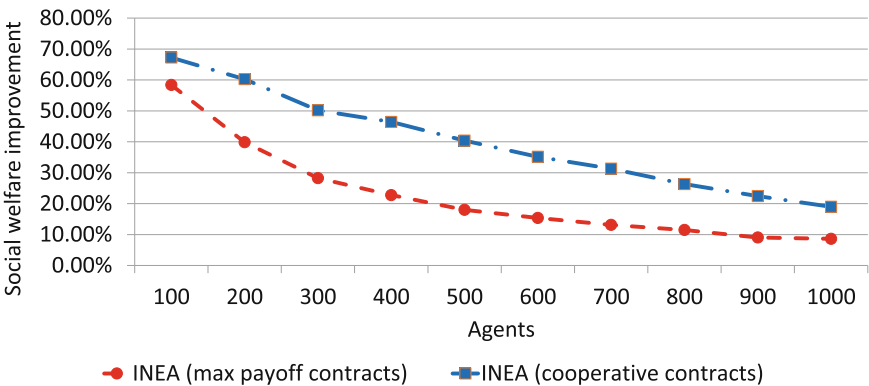


Fig. 4. Social welfare of stable outcomes on random networks

decreases with problem size. Inspecting Fig. 4 one can see that for games with 500 agents the improvement is still sizeable - 20% for maximal payoff contracts and 40% for cooperative contracts.

5 Conclusions

An iterative distributed search algorithm for finding a stable outcome of improved efficiency in multi-agents games (MAG) is proposed – the Iterative Nash efficiency Enhancement Algorithm (INEA). The algorithm searches for a solution to the global constraint of stability and performs heuristic distributed optimization search on the soft constraints of efficiency for a good solution (e.g., of higher social welfare).

The INEA algorithm iterates among all agents in the game in a fixed order and each agent in its turn exchanges messages with the agents that interact with it (e.g., its neighbors). Messages propose the selected change of strategy of the agent and in response other agents that are affected by it, may offer transfer of funds in order to compensate the proposer for not changing its strategy and secure a desired outcome.

The INEA algorithm is guaranteed to converge to a state that is of higher efficiency than its initial state and that can be stabilized by the use of transfers of funds among the agents who run the algorithm. A transfer function that can achieve stability is computed by the agents running the algorithm. Since the transfers that secure stability form a binding contract among agents, computed during the search, it is natural to think of them as side payments [13]. The agents playing the game can start from any initial outcome and by running INEA are guaranteed to arrive at a state of higher efficiency that can be stabilized by the use of side payments.

One can think of the proposed INEA algorithm as an extension to the well known best response mechanism (cf. [25, 26]). The proposed method uses transfer of payoffs to extend the standard best-response. The use of this extension guarantees convergence to a stable state in general MAGs, whereas best response is only guaranteed to converge to a PNE for games that are potential games [12]. More importantly, the proposed INEA procedure is guaranteed to converge to a stable state of improved efficiency. This is in contrast to standard best response that does not address efficiency at all.

References

1. Osborne, M., Rubinstein, A.: A Course in Game Theory. The MIT Press, London (1994)
2. DallAsta, L., Pin, P., Ramezanzpour, A.: Optimal equilibria of the best shot game. *J. Public Econ. Theory* **13**(6), 885–901 (2011)
3. Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V.V.: Algorithmic Game Theory. Cambridge University Press, Cambridge (2007)

4. Nguyen, T.-V.-A., Lallouet, A.: A complete solver for constraint games. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 58–74. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_8
5. Palmieri, A., Lallouet, A.: Constraint games revisited. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017), Melbourne, AU, pp. 729–735 (2017)
6. Roughgarden, T.: *Selfish Routing and the Price of Anarchy*. The MIT Press, Cambridge (2005)
7. Monderer, D., Tennenholtz, M.: Strong mediated equilibrium. *Artif. Intell.* **173**(1), 180–195 (2009)
8. Grubshtein, A., Meisels, A.: Finding a nash equilibrium by asynchronous backtracking. In: Milano, M. (ed.) CP 2012. LNCS, pp. 925–940. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33558-7_66
9. Litov, O., Meisels, A.: Distributed search for pure nash equilibria in graphical games. In: Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems (AAMAS-2016), pp. 1279–1280 (2016)
10. Wahbi, M., Brown, K.N.: A distributed asynchronous solver for nash equilibria in hypergraphical games. In: ECAI, pp. 1291–1299 (2016)
11. Grinshpoun, T., Grubshtein, A., Zivan, R., Netzer, A., Meisels, A.: Asymmetric distributed constraint optimization problems. *JAIR* **47**, 613–647 (2013)
12. Monderer, D., Shapley, L.S.: Potential games. *Games Econ. Behav.* **14**, 124–143 (1996)
13. Jackson, M.O., Wilkie, S.: Endogenous games and mechanisms: side payments among players. *Rev. Econ. Stud.* **72**(2), 543–566 (2005)
14. Kearns, M.J.: Graphical games. In: Vazirani, V.V., Nisan, N., Roughgarden, T., Tardos, É. (eds.) *Algorithmic Game Theory*, pp. 159–178. Cambridge University Press (2007)
15. Jackson, M.O.: *Social and Economic Networks*, vol. 3. Princeton University Press, Princeton (2008)
16. Levit, V., Komarovskiy, Z., Grinshpoun, T., Meisels, A.: Tradeoffs between incentive mechanisms in Boolean games. *IJCA I*, 68–74 (2015)
17. Jackson, M.O., Zenou, Y.: Games on networks. In: Young, P., Zamir, S. (eds.) *Handbook of Game Theory*, vol. 4, pp. 102–191 (2014)
18. Komarovskiy, Z., Levit, V., Grinshpoun, T., Meisels, A.: Efficient equilibria in a public goods game. In: WI-IAT, pp. 214–219 (2015)
19. Grubshtein, A., Meisels, A.: A distributed cooperative approach for optimizing a family of network games. In: Brazier, F.M.T., Nieuwenhuis, K., Pavlin, G., Warnier, M., Badica, C. (eds.) IDC 2011. *Studies in Computational Intelligence*, vol. 382, pp. 49–62. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-24013-3_6
20. Grubshtein, A., Zivan, R., Grinshpoun, T., Meisels, A.: Local search for distributed asymmetric optimization. In: AAMAS, pp. 1015–1022 (2010)
21. Zivan, R., Okamoto, S., Peled, H.: Explorative anytime local search for distributed constraint optimization. *Artif. Intell.* **212**, 1–26 (2014)
22. Zivan, R., Parash, T., Cohen, L., Peled, H., Okamoto, S.: Balancing exploration and exploitation in incomplete min/max-sum inference for distributed constraint optimization. *Auton. Agents Multi Agent Syst.* **31**, 1165–1207 (2017)
23. Shapley, L.: A value for n-person games. In: Kunh, H.W., Tucker, A.W. (eds.) *Contributions to the Theory of Games*. *Annals of Mathematical Studies*, vol. 28, pp. 307–317 (1953)

24. Hirshleifer, J.: From weakest-link to best-shot: the voluntary provision of public goods. *Public Choice* **41**(3), 371–386 (1983)
25. Nisan, N., Schapira, M., Valiant, G., Zohar, A.: Best-response mechanisms. In: *Innovations in Computer Science - ICS*, Beijing, China, pp. 155–165 (2011)
26. Nisan, N., Schapira, M., Valiant, G., Zohar, A.: When is it best to best-respond? *SIGecom Exch.* **10**, 16–18 (2011)

Testing and Verification Track



Metamorphic Testing of Constraint Solvers

Özgür Akgün^(✉), Ian P. Gent, Christopher Jefferson, Ian Miguel,
and Peter Nightingale

School of Computer Science, University of St Andrews, St Andrews, UK
{ozgur.akgun, ian.gent, caj21, ijm, pwn1}@st-andrews.ac.uk

Abstract. Constraint solvers are complex pieces of software and are notoriously difficult to debug. In large part this is due to the difficulty of pinpointing the source of an error in the vast searches these solvers perform, since the effect of an error may only come to light long after the error is made. In addition, an error does not necessarily lead to the wrong result, further complicating the debugging process. A major source of errors in a constraint solver is the complex constraint propagation algorithms that provide the inference that controls and directs the search. In this paper we show that metamorphic testing is a principled way to test constraint solvers by comparing two different implementations of the same constraint. Specifically, specialised propagators for the constraint are tested against the general purpose table constraint propagator. We report on metamorphic testing of the constraint solver MINION. We demonstrate that the metamorphic testing method is very effective for finding artificial bugs introduced by random code mutation.

1 Introduction

Metamorphic testing [24] involves generating new test cases from existing ones, where the expected result of a new test case can be generated from the result of an existing test via a *metamorphic relation*. By comparing the results of the original test with the new one we can identify cases where the metamorphic relations are broken, indicating the presence of errors in the implementation. As an illustrative example in the context of a constraint satisfaction problem (CSP), given any unsolvable CSP, adding a constraint or removing domain values from a variable should result in another unsolvable CSP.¹

Metamorphic testing cannot be used completely in isolation — a solver that immediately returns that a problem is unsolvable would pass all of the metamorphic tests given in this paper. However, the big advantage of metamorphic testing of a constraint solver is that it transforms the relatively hard problem of validating the behaviour of a constraint solver against an independent oracle into comparing a solver against its own behaviour on a different problem.

¹ The authors have experienced both of these conditions being violated in both their own, and other, solvers.

One major area of constraints research is the creation of new propagation algorithms for constraints. Here metamorphic testing shines — we can check the correctness and propagation level of a new propagation algorithm for a constraint by comparing it with a previously existing algorithm. Assuming the two algorithms do not contain exactly the same bug (which is unlikely, if the two algorithms were independently designed and implemented), comparing one against the other provides us with a high level of confidence that both are correct.

The central argument of this paper is that metamorphic testing is a good fit for constraint solving. While it can be difficult to solve a constraint problem, it is easy to generate one randomly, and there are simple transformations that can take one constraint problem and produce another with the same set of solutions.

2 The Constraint Solver Minion

Throughout this paper, we use the MINION constraint solver to illustrate metamorphic testing in constraint solvers. We begin with an overview of the most important features of MINION's input language, for a more extensive discussion of MINION's features see [10].

Variable Types: MINION provides four implementations of integer variables.

BOOL (Initial domain must be $\{0, 1\}$), DISCRETE and BOUND (Initial domain must be a range $\{x..y\}$) and SPARSEBOUND. BOUND and SPARSEBOUND only support changing the bounds of a variable during search.

Constraints: The current release of MINION features 72 constraints, including arithmetic and logical constraints, element [8], alldiff [11] and gcc [19].

Constraints can also be combined, either disjunctively or conjunctively [14].

Reification: Every constraint in MINION can be reified or reified implied [14] (also called half reification [6]): these combine a constraint C and a Boolean b into the new constraint $b \iff C$ and $b \implies C$ respectively.

Search Orders: MINION provides ten variable orderings, including static orderings, smallest domain first, conflict ordering [18] and weighted degree [2].

Global Propagation Level: MINION can perform higher levels of consistency, including Singleton Arc Consistency [5].

Some of MINION's features make testing particularly challenging. Several constraints have different implementations for the different types of variables. Also, MINION allows propagators to dynamically change the set of variables whose changes they are informed of. Further, these changes can either automatically revert when search backtracks, or remain in their new location [8]. This means testing must verify how a propagator behaves over an entire search.

3 Instance-Based Testing in Minion

The original method of testing MINION involved a set of small test instances, for which the following information was derived by hand:

SOLCOUNT *n* The number of solutions to the problem is *n*.
NODECOUNT *n* The number of search nodes until the first solution under MINION's default search strategy is *n*.
CHECKONESOL *sol* The first solution with MINION's default search strategy is *sol* (given as a list).

SOLCOUNT tests are used to test variable and value ordering heuristics, while **NODECOUNT** is used to check that a constraint propagator achieves an expected level of consistency. At the time of writing, this test suite includes 312 instances. New tests are added whenever a bug is discovered in MINION.

For illustration, the bugs found in the inequality constraint ($\sum c_i \times x_i \leq y$ for constants c_i and variables x_i and y) before metamorphic testing was introduced include: Summing a list of Boolean variables to ≥ -1 ; summing variables to greater than 65,536 (which overflowed the `unsigned short` type in C); crashing if any $c_i = 0$; and sums where all variables were assigned at the root node.

This test-based approach is effective at preventing the reintroduction of previously identified bugs. However, we still found users discovering a large number of errors. This motivates the adoption of the more proactive metamorphic approach. Instance-based testing remains in use to complement the metamorphic tester described in the following section, and remains the primary means of testing variable and value ordering heuristics.

4 Metamorphic Testing in Minion

In this section we explain how MINION uses metamorphic testing for propagators. Given a constraint propagator to test, testing begins by generating a problem instance consisting of a single occurrence of that constraint with a random domain for each variable in its scope. As an example, consider `difference(x,y,z)`, which implements the constraint $|x - y| = z$. The leftmost instance in Fig. 1 shows a corresponding instance. The tester adds several random additional constraints, as presented in the middle instance of Fig. 1. These extra constraints are necessary because many bugs only occur when multiple propagators interact. Optionally, the tester may at this stage create an optimisation problem by **MAXIMISING** or **MINIMISING** a randomly chosen variable. Finally, the tester transforms this instance into another instance with the same set of solutions by replacing the constraint being tested with an equivalent **table** constraint, as presented in the rightmost instance of Fig. 1. The tester then compares the searches of the middle and rightmost instances.

We test the reified version of each constraint propagator separately. The production of metamorphic instances proceeds similarly, but differs in the construction of the table constraint. The scope of the table constraint includes that of the reified constraint, and the reification variable.

The rationale for using a table constraint is that it can represent any other constraint and many propagators for the table constraint achieve **GAC** [15]. Any errors in MINION's implementation of the table constraint propagator would likely be found while testing it against all other constraints. Also,

```

MINION 3                               MINION 3                               MINION 3
**VARIABLES**                          **VARIABLES**                          **VARIABLES**
DISCRETE x1 {-3..0}                    DISCRETE x1 {-3..0}                    DISCRETE x1 {-3..1}
DISCRETE x2 {0..4}                     DISCRETE x2 {0..4}                     DISCRETE x2 {-0..4}
DISCRETE x3 {0..1}                     DISCRETE x3 {0..1}                     DISCRETE x3 {0..1}
**CONSTRAINTS**                        **CONSTRAINTS**                        **TUPLELIST**
difference(x1,x2,x3)                   difference(x1,x2,x3)                    tab 3 3
**EOF**                                 diseq(x1,x3)                             -1 0 1 0 0 0 1 1
                                         eq(x2,x3)                                **CONSTRAINTS**
**EOF**                                 **EOF**                                  table([x1,x2,x3],tab)
                                                                                 diseq(x1,x3)
                                                                                 eq(x2,x3)
                                         **EOF**

```

Fig. 1. Stages of production of metamorphic instances.

MINION includes several implementations of table constraints [15,20], which are compared.

The testing process is automated in Python. For each propagator, there is a Python function that, given a list of domains for each variable, returns the list of allowed tuples. MINION is run on the original and transformed instances with the default static variable ordering. The search trees of both instances are compared according to the metamorphic relations described in the following section.

5 Tree Comparison

The tester compares the search trees explored by MINION in solving two different instances, which differ only in the expression of the constraint being tested. If the tested propagator achieves GAC, MINION will explore identical search trees for both instances. If the propagator achieves less than GAC then the search tree may include extra nodes and values but will still find the same set of solutions.

We begin with a formal definition of a two-way branching search tree, which is commonly employed in modern constraint solvers, including MINION:

Definition 1. Consider a CSP with a list of variables V and a function D which gives the initial domain of every variable. A search node contains a function N which maps each $v \in V$ to a subset of $D(v)$ and one of the following labels:

Fail: A **fail** node has no children and represents no solutions. In this case $\forall v \in V. |N(v)| = 0$ (some solvers do not empty all domains at a failure node. We require this to make node comparison easier).

Solution: In a **solution** node, $|N(v)| = 1$ for all $v \in V$. N then represents a single assignment to every variable, which is a solution to the CSP.

Branch: In a **branch** node, there is a branching literal $\langle v, x \rangle$, where $v \in V$ and $x \in D(v)$ and two child nodes, where in the left v is assigned x , and in the right x is removed from the domain of v . $|D(v)|$ must be > 1 .

This definition is purposely loose — we could easily place (and check) extra requirements on search trees, in particular between a search node and its children in the case of branching nodes but this has not appeared necessary thus far.

When the tester compares two propagators which achieves GAC, comparing search trees is easy – check the search trees are identical, including equality of the variable domains at each node. For weaker propagators we need a more complex method of comparing trees, which is given in Definition 2.

Definition 2. *Consider a set of variables V with initial domains D and search nodes N_1, N_2 from two search trees of CSPs with variables V with initial domains D (but possibly different constraints). The tree rooted on N_1 is a subtree of the tree rooted on N_2 if $\forall v \in V. N_1(v) \subseteq N_2(v)$ and the following are true:*

- If N_2 is a **solution** node, so is N_1 , similarly if N_2 is a **fail** node, so is N_1 .
- If N_2 is a **branch** node with branch literal $l = \langle v, x \rangle$, then one of the following cases is true:
 - N_1 is also a **branch** node, N_1 also branches on l , and N_1 's left child is a subtree of N_2 's left child (and similarly for right child).
 - $x \notin N_1(v)$, there are no solutions in the subtree under N_2 's left child and N_2 's right child is a subtree of N_1 .

The most complex part of Definition 2 is the final branching case. In this case, we branch on a literal in N_2 that does not occur in N_1 . All we know about the left branch of this node is that it will contain no solutions. We will show later this is equivalent to being a supertree of a fail node.

We also need to define propagators, and how they change search states. The property the tester uses to compare search trees is given in Lemma 1 below. Intuitively, Lemma 1 shows that if we run propagators on a search state, then either adding more literals to the search state, or replacing one or more propagators with weaker propagators, never leads propagation removing more literals. This lemma uses the fact that a GAC propagator removes every literal which can correctly be removed and that propagators are inflationary (they never add literals to the search state). This lemma applies to a wide range of propagators, including non-monotonic propagators or propagators which use randomised algorithms [23]. This paper does not contain a full discussion of propagators: for a general overview see [1, 22].

Lemma 1. *Consider two search states N and M on the same set of variables V , where $\forall v \in V. N(v) \subseteq M(v)$ (from here denoted by $N \subseteq_V M$), and two lists of inflationary propagators $P = \langle p_1, \dots, p_n \rangle$ and $Q = \langle q_1, \dots, q_n \rangle$ where for all i , p_i and q_i are both propagators for a constraint c_i and the p_i achieves GAC. If N_P is a result of applying elements of P to N until a fixed point is reached, and similarly for M_Q , then $N_P \subseteq_V M_Q$.*

Proof. We proceed by induction. Consider a search state S where $N_P \subseteq_V S \subseteq_V M$, and a propagator $q_i \in Q$. Then the result S_Q of applying q_i to S must be contained in M as q_i is inflationary, and must satisfy $N_P \subseteq_V S_Q$, because $N_P \subseteq S$, and N_P is a fixed point for P and therefore all of the p_i .

Given Lemma 1, we can now prove our main result, which is that the definition of subtrees given in Definition 2 is correct, and the tester uses this in metamorphic testing to check the correctness of propagators.

Lemma 2. *Consider a set of variables V with initial domains D , and two lists of propagators $P = \langle p_1, \dots, p_n \rangle$ and $Q = \langle q_1, \dots, q_n \rangle$, where for all i p_i and q_i are both propagators for a constraint c_i , and the p_i achieves GAC. Given a static variable and value ordering, a search tree generated by P is a subtree of a search tree generated by Q .*

Proof. We will proceed inductively. To begin, apply P and Q to D to get the root nodes of the search states. By Lemma 1 we know that $\forall v \in V.P(D) \subseteq Q(D)$, and as correct propagators never remove solutions, the set of assignments to $P(D)$ and $Q(D)$ which are solutions will be the same. Now consider any pair of search states N_P and N_Q where $\forall v \in V.N_P(v) \subseteq N_Q(v)$ and the assignments to N_P and N_Q which are solutions are the same. We then continue our induction by considering the possible types of N_Q .

If N_Q is a **branch** node with branching literal $l = \langle v, x \rangle$, there are two possibilities: either $x \in N_P(v)$ or $x \notin N_P(v)$. If $x \notin N_P(v)$, then the left child of N_Q is created from N_Q by reducing the domain of v to $\{x\}$ and then applying Q . None of the assignments to this search state will be solutions, as $x \notin N_P(v)$ and the set of assignments to N_P and N_Q which are solutions are the same. Therefore this child and all of its children are not **solution** nodes.

If $x \in N_P(v)$, then the branching literal we pick for N_P must also be l , as the value and variable order is static. The left child of N_P is created by reducing the domain of v to $\{x\}$ and then running P to a fixed point. The left child of N_Q is created similarly, and by Lemma 1 these children will satisfy our inductive hypothesis. Similarly, the right children also satisfy the inductive hypothesis.

If N_Q is a **fail** node, then as $\forall v \in V.N_P(v) \subseteq N_Q(v)$, N_P is also a **fail** node. Similarly, if N_Q is a **solution** node, then as the assignments to N_Q and N_P which are solutions are the same, then N_P must also be a **solution** node.

6 Practical Experience

Metamorphic testing was introduced into MINION in 2007. It is impossible to measure reliably the number of bugs it has discovered over time, as it has been used during the development of every constraint [9, 11–15, 19, 20] added to MINION since 2007 and many bugs will be found at this stage. We can say no bug has been reported in any propagator which was published after development in MINION, except for one which was not added to the metamorphic tester.

6.1 Mutation Testing

In order to test the robustness of metamorphic testing, we performed mutation testing [4, 16]. We tested three constraints: strict (**lexless**) and non-strict (**lexleq**) lexicographic ordering, **difference**, and the reified and reified/implified

variants of these three constraints. `difference` is an example of a propagator with complex arithmetic, while `lexless` and `lexleq` are global constraints. These constraints were still simple enough to check which mutations actually introduced bugs. We ran the tester for the `difference` constraints for 1000 tests, and the `lex` constraints for 100 tests (the metamorphic tester tests global constraints less as tests take much longer to run). We ran the tester 10 times on each mutation. After filtering out non-compiling mutations, for each constraint we have 20 mutations generated by changing a random Boolean operator (an ROR mutation [17]) and 10 mutations generated by removing a random line of code (an SDL mutation [17]).

The tester found each bug in at least one out of the ten test runs. The non-buggy mutations fall into three groups. Five mutations introduced inefficiencies without changing propagation. Three mutations changed the propagation level of a non-GAC propagator, which is not currently be detected. Finally, one mutation improved performance by removing unnecessary code!

Three mutations were only detected in some runs of the tester. Two *lex* bugs passed more than half of the time, each passing six out of ten test runs. These bugs only affected the first invocation of the constraint, and required several variables already to be assigned by other constraints. This demonstrates the importance of introducing extra constraints to the models. One *difference* instance passed eight out of ten times. This bug introduced a problem in reifying the constraint, by acting as if the domain of the first variable had no holes in it. This problem only very occasionally resulted in an incorrect solution.

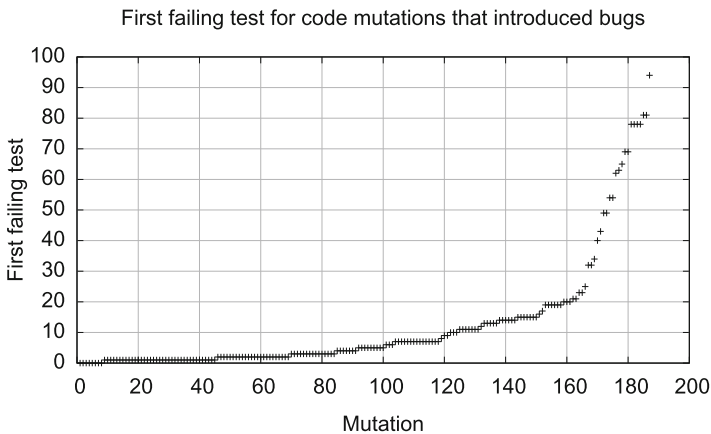


Fig. 2. Number of tests before detecting a bug introduced by a code mutation

Figure 2 shows the median number of tests that were needed before detecting a bug that was introduced by a source code mutation, sorted by first failing test. In many cases bugs are detected with less than 25 test instances.

6.2 Limitations of Metamorphic Testing

There have been three wrong-answer bugs found in released versions of Minion since the introduction of metamorphic testing. One involved a constraint which was not added to the metamorphic tester. The other two involved large domains – the first involved domain values larger than 2^{30} , the second searches with more than 2^{31} search nodes. Another source of occasional bugs has been bad behaviour on inputs which were supposed to be rejected – the tester only tests valid constraint problems.

6.3 Related Work

Many large A.I. systems have developed similar test frameworks, where many problems, sometimes randomly generated, are tested for correctness. We discuss a few of the most relevant here.

Testing of the Gecode solver [7] has evolved similarly to MINION's. Gecode has an extensive list of fixed tests and a tester which uses random problems. Gecode's random tests creating a random search state which contains a known solution. Literals not in the known solution are removed one by one and the propagator is run, checking no literals from the known solution are ever removed. Other properties of propagators (such as if reaching a fixed point) are also checked. This approach has some limitations compared to MINION, such as not testing multi-solution problems, optimisation problems, or if the propagator works when backtracking. Also, search-tree comparison metamorphic testing should be easier to add to other solvers, as it only requires solvers to output their search tree.

Brummayer and Biere [3] generate random inputs for SMT solvers. They compare the results of multiple solvers looking for inconsistencies. This was very successful, but not useful for solvers like Minion with a unique input format. Also, their technique could not be used to detect the level of propagation achieved.

Reger, Suda and Voronkov discuss the testing of the Vampire theorem prover [21]. They use a fixed set of benchmarks, testing the solver's many configuration options against each other, and also validating the generated proof.

7 Conclusions

Overall, metamorphic testing has been a great success for MINION. It is impossible to measure the number of bugs it has discovered, as it is used by developers when creating new propagators, when large numbers of bugs are found and fixed during development.

While metamorphic testing has been very useful in MINION, it is not immediately applicable to all solvers. One major limitation is learning solvers, where new constraints are added during search. Significant metamorphic tests could still be performed in such solvers by checking that the solver produces the same set of solutions, and optimisation problems achieve the same optimal solution. It would be interesting to investigate if more subtle forms of metamorphic testing would be beneficial in such cases.

Acknowledgements. We thank EPSRC for funding this work via the grants EP/P015638/1 and EP/P026842/1. Dr Jefferson holds a Royal Society University Research Fellowship.

References

1. Apt, K.: Principles of Constraint Programming. Cambridge University Press, New York (2003)
2. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI 2004, pp. 482–486 (2004)
3. Brummayer, R., Biere, A.: Fuzzing and delta-debugging SMT solvers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories, SMT 2009, pp. 1–5. ACM, New York (2009)
4. Budd, T.A.: Mutation analysis of program test data. Ph.D. thesis, New Haven, CT, USA (1980)
5. Debruyne, R., Bessière, C.: Some practicable filtering techniques for the constraint satisfaction problem. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 1997), pp. 412–417 (1997)
6. Feydy, T., Somogyi, Z., Stuckey, P.J.: Half reification and flattening. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 286–301. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23786-7_23
7. Gecode Team: Gecode: Generic constraint development environment (2006). <http://www.gecode.org>
8. Gent, I.P., Jefferson, C., Miguel, I.: Watched literals for constraint propagation in minion. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 182–197. Springer, Heidelberg (2006). https://doi.org/10.1007/11889205_15
9. Gent, I.P., Jefferson, C., Linton, S., Miguel, I., Nightingale, P.: Generating custom propagators for arbitrary constraints. *Artif. Intell.* **211**, 1–33 (2014). <http://www.sciencedirect.com/science/article/pii/S000437021400023X>
10. Gent, I.P., Jefferson, C., Miguel, I.: Minion: a fast scalable constraint solver. In: ECAI, vol. 141, pp. 98–102 (2006)
11. Gent, I.P., Miguel, I., Nightingale, P.: Generalised arc consistency for the alldifferent constraint: an empirical survey. *Artif. Intell.* **172**(18), 1973–2000 (2008)
12. Gent, I., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalised arc consistency for extensional constraints. In: AAAI (CPPOD-19-2006-A), pp. 191–197 (2007). <http://www.aaai.org/Papers/AAAI/2007/AAAI07-029.pdf>
13. Jefferson, C., Kadioglu, S., Petrie, K.E., Sellmann, M., Živný, S.: Same-relation constraints. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 470–485. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_38
14. Jefferson, C., Moore, N.C.A., Nightingale, P., Petrie, K.E.: Implementing logical connectives in constraint programming. *Artif. Intell.* **174**(16–17), 1407–1429 (2010)
15. Jefferson, C., Nightingale, P.: Extending simple tabular reduction with short supports. In: IJCAI (2013)
16. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* **37**(5), 649–678 (2011)
17. King, K.N., Offutt, A.J.: A fortran language system for mutation-based software testing. *Softw. Pract. Exp.* **21**(7), 685–718 (1991). <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380210704>

18. Lecoutre, C., Saïs, L., Tabary, S., Vidal, V.: Reasoning from last conflict(s) in constraint programming. *Artif. Intell.* **173**(18), 1592–1614 (2009). <http://www.sciencedirect.com/science/article/pii/S0004370209001040>
19. Nightingale, P.: The extended global cardinality constraint: an empirical survey. *Artif. Intell.* **175**(2), 586–614 (2011)
20. Nightingale, P., Gent, I.P., Jefferson, C., Miguel, I.: Short and long supports for constraint propagation. *J. Artif. Int. Res.* **46**(1), 1–45 (2013). <http://dl.acm.org/citation.cfm?id=2512538.2512539>
21. Reger, G., Suda, M., Voronkov, A.: Testing a saturation-based theorem prover: experiences and challenges. In: Gabmeyer, S., Johnsen, E.B. (eds.) TAP 2017. LNCS, vol. 10375, pp. 152–161. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61467-0_10
22. Rossi, F., van Beek, P., Walsh, T. (eds.): *Handbook of Constraint Programming*. Elsevier, New York (2006)
23. Schulte, C., Tack, G.: Weakly monotonic propagators. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 723–730. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_56. <http://www.gecode.org/paper.html?id=SchulteTack:CP:2009>
24. Chen, T.Y., Cheung, S.C., Yiu, S.M.: Metamorphic testing: a new approach for generating next test cases. Technical report HKUST-CS98-01 (1998)



Algebraic Fault Attack on SHA Hash Functions Using Programmatic SAT Solvers

Saeed Nejadi¹(✉), Jan Horáček², Catherine Gebotys¹, and Vijay Ganesh¹

¹ University of Waterloo, Waterloo, ON, Canada
{snejati,cgebotys,vganesh}@uwaterloo.ca

² University of Passau, Passau, Germany
Jan.Horacek@uni-passau.de

Abstract. We present an algebraic fault attack (AFA) solver for recovering secret bits from hardware implementations of the SHA family of hash functions. The crucial insight in our method is the use of SHA-based propagation and conflict-analysis methods in the inner-loop of a Boolean conflict-driven clause-learning SAT solver, à la the DPLL(T) paradigm. In our method the fault-injected part of the hash function is translated into a Boolean formula (which is then fed as input to the SAT solver), while the rest is encoded via a programmatic interface as part of the SAT solver's propagation and conflict analysis routines. Such an approach enables the addition of learnt clauses to the SAT solver in an on-demand and lazy fashion. We evaluated our tool under a variety of fault models, and showed that we can recover the secret bits faster and with far fewer number of injected faults compared to previous best work. AFA is a powerful way of empirically verifying the strength of a cryptographic function's implementation.

1 Introduction

Cryptographic hash functions, such as the SHA family, play a critical role in a variety of settings in cryptography (e.g., authenticated encryption, pseudo random number generation, digital signatures, etc.) [27]. While there is some recent progress on practical collision attacks on SHA-1 [33], inversion attacks on the full version of the standard SHA family of functions are still impractical [11]. Given that these functions seem highly resistant to direct inversion attacks, many researchers have turned to *implementation inversion attacks*¹, wherein, the attacker gathers information on implementations of these hash functions

¹ There are two basic approaches to implementation attacks, namely, passive and active implementation attacks. In passive attacks, the attacker measures some aspect of the computations on a target implementation via side-channel such as power consumption or timing, to find patterns that can be exploited. By contrast, in active attacks the target implementation is manipulated as part of the attack. In this paper, we consider only active attacks.

(or any cryptographic primitive) in an attempt to reduce the size of search space. One form of this type of attack, called *fault injection analysis*, involves intentionally introducing faults in the operation of cryptographic devices and analyzing the incorrect outputs to recover the embedded secret key. These faults could be injected via a variety of methods, like heating or varying the voltage of the power supply in a controlled fashion to attack hardware implementing these functions [1, 4, 5]. Fault attacks were first proposed in 1997 as a way to break RSA-CRT cipher (cf. [7]).

There are broadly two classes of fault attacks that researchers have studied, namely, *differential fault attacks* (DFA) and *algebraic fault attacks* (AFA). The DFA method was first used in breaking DES [6], and has been applied to many block ciphers [2, 21, 23], stream ciphers [20], and hash functions [15, 19, 26]. At a high level, the DFA method exploits the differences in the relation between the faulty outputs and the intermediate variables compared to the correct outputs in order to recover an inner state. Propagation of induced faults in forward direction and deduction of fault differences, backward from the output to the fault location, so-called *fault equations*, is examined manually by a cryptanalyst.

Algebraic fault attack methods combine fault injections with algebraic cryptanalysis [10]. In this approach, the cryptographic function and faults are translated into algebraic equations over a finite field, and the secret key or message is recovered by solving these equations using a SAT or SMT solver. Fault equations refer in this case to an algebraic representation of the cryptographic function starting from the injected fault location up to its output. The advantage of AFA over DFA is that the solver takes care of propagation of the fault, and thus significantly reducing the human effort required to launch a successful attack. AFA has been used to automate DFA methods on block ciphers [35, 36], stream ciphers [28] and hash functions [18, 25].

AFA methods are a powerful way of empirically verifying the strength of cryptographic function's implementation through fault analysis. AFA methods have significant advantages over previous approaches since they leverage the continuous scalability improvements in SAT and SMT solvers. Having said that, it is well known that merely using the solver as a blackbox (à la the *eager* approach) is not going to yield the best results. The efficacy of AFA methods broadly rely on three important factors that any SAT/SMT solver user would readily recognize, namely, the type of encoding of the cryptographic primitive in Boolean or suitable SMT logic, the underlying solver, and the effectiveness with which the user is able to tune or modify the underlying solver's heuristics.

In their original paper on AFA [10], the authors describe a *lazy* approach to AFA, wherein, part of the cryptographic primitive (more precisely, the fault-injected part) is translated into a Boolean formula, and the rest of the primitive is used to verify solutions generated by the solver. If the solution is incorrect, their tool blocks it by adding an appropriate clause to the solver, and repeats until the correct solution is found. While their method is clearly sound, complete, and terminating, the authors do not exploit the solver's power in a whitebox fashion nor do they explore encodings that may be best suited for an algebraic fault

attack. While researchers have explored different kinds of encodings subsequent to the paper by Courtois et al. [10], none of them use the underlying SAT/SMT solver in DPLL(T) fashion (cf. [18, 35, 36]).

1.1 High-Level Overview of Our Method to Algebraic Fault Attack

In this paper, we propose a DPLL(T)-style AFA method, wherein, we extend both the *Boolean constraint propagator* (BCP) and the conflict-analysis heuristics in a state-of-the-art SAT solver, MapleSAT [24]. More precisely we propose a programmatic SAT solver-based method [17] for AFA. Our extension of BCP is similar to theory propagation in DPLL(T), and we refer to this extension as the SHA propagator. The conflict-analysis extension is similar to the theory conflict analysis in DPLL(T), and we refer to this extension as the SHA conflict analyzer. *Programmatic SAT solving* is a particular variation of DPLL(T), and differs from the general concept in 3 ways: first, the theory solver in the context of programmatic SAT can be an arbitrary piece of code, in that we place no requirements of completeness on it; second, this code might in turn be particularized to every input to the solver. That is, unlike the T -solver in DPLL(T) which remains invariant for all formulas from the language of T , the code added via the programmatic interface in a programmatic SAT solver can be specific and unique to each input; and finally, the interface of programmatic SAT solvers is much simpler than that of SMT solvers which are the quintessential implementation of DPLL(T).

How our Method Works. At a high level, in our method the fault-injected part of the hash function, along with a target, is translated into a Boolean formula (which is then fed as input to the SAT solver), while the full implementation of SHA is encoded via a programmatic interface as part of the SAT solver's propagation and conflict analysis routines. Such an approach enables the addition of conflict clauses to the appropriate database in an on-demand and lazy fashion. We refer to our addition to the solver's propagation routine as the *SHA propagator*, and the one to the solver's conflict analysis routine as the *SHA conflict analyzer*. We evaluate our tool under a variety of fault models and show that we can recover the secret bits (in our setting, the secret bits correspond to any message that hashes to the given target) with fewer number of injected faults compared to previous best work (reducing the cost of attack). Although fault injections are done on hardware devices, in this work we are simulating the fault injection in software, by picking a random value as embedded secret bits and XORing random values to the intermediate state words as fault injection.

SHA Propagator. While analyzing different encodings of the SHA hash functions in Boolean logic, we noticed that the native BCP in SAT solvers does not propagate all the input bits (once set) all the way to the output bits of the SHA function for certain kinds of encodings of SHA in Boolean logic. More precisely, given a Boolean function f over input variables x and output variables y , there exist encodings ϕ_f (in conjunctive normal form) such that the standard-issue BCP does not propagate the values assigned to x all the way to y . A natural

and cost-effective way to strengthen the native BCP in SAT solvers would then be to add a SHA propagator that propagates inputs to the encoding of SHA to all its output bits, and adds clauses to the clause database appropriately. In our experiments, this method alone gave a massive boost to the performance of MapleSAT over AFA instances.

SHA Conflict Analyzer. As alluded to above, the SHA conflict analyzer is the checker that verifies whether the solution found by the solver is indeed a valid message for the given target. If not, a conflict clause is added to the conflict clause database of the solver. This mechanism prunes the search space dramatically according to our experiments. Otherwise, the validated solution is output by the solver. Unlike the AFA method proposed by Courtois et al. [10], our SHA conflict analyzer is called in the inner loop of the SAT solver thus taking advantage to both its inherent incrementality and conflict analysis capabilities.

1.2 Contributions

1. We present a new AFA method (see Sect. 4), implemented as part of the MapleSAT solver via its programmatic interface. We define and implement a SHA propagator and a SHA conflict analyzer as part of MapleSAT¹. We note that our method can be easily extended to other cryptographic primitives such as symmetric ciphers.
2. Further, we evaluate our tool under several different fault models from single bit to words, and show that we can recover the secret bits (in our setting, the secret bits correspond to any message that hashes to the given target) with far fewer number of injected faults in both SHA-1 and SHA-2 hash functions compared to previous approaches (see experiments in Sect. 5).
3. Finally, we show that our programmatic approach to AFA is up to an order of magnitude faster than using a SAT solver as a black box (see experiments in Sect. 5).

Paper Structure. The paper is structured as follows: In Sect. 2 we recall the SHA-1 and SHA-256 functions algebraically, together with notions such as generalized arc consistency. In Sect. 3 we describe the programmatic interface in the MapleSAT solver [24]. In Sect. 4 we describe our AFA in general and our programmatic SAT solver-based AFA method. In Sect. 5 we present our results on attacking SHA-1 and SHA-2 and demonstrate that our programmatic approach outperforms the previous best AFA both in terms of number of faults and scalability. We present related work in Sect. 6. Finally, we conclude in Sect. 7.

¹ Additional resources can be found at <https://sites.google.com/view/crypto-sat/algebraic-fault-analysis>.

2 Background

2.1 Arc Consistency and SAT

The following definition is motivated by the notion of arc consistency in Constraint Satisfaction Problems (CSP). We spell out a modified version of general arc consistency given in [3].

Definition 1. Let R be an inference rule of propositional logic. Let ϕ be a Boolean formula which encodes a constraint C in CNF. We say that the encoding of C into ϕ **R -maintains Generalized Arc Consistency (GAC)** if for all partial assignments α , i.e., a conjunction of literals, and for all literal ℓ the following holds

$$C \wedge \alpha \vdash \ell \Rightarrow \phi \wedge \alpha \vdash_R \ell \quad (\text{i.e., } \ell \text{ is derived from } \phi \wedge \alpha \text{ by } R).$$

The following example illustrates the fact that some encodings do not maintain GAC under unit propagation (UP), which is the default propagation procedure in SAT solvers.

Example 1. Consider the pseudo-Boolean constraint: $x + y \leq 0$, i.e., $x, y \in \{0, 1\}$ and “+” denotes integer addition. We can encode this constraint into a CNF formula ϕ by using a half-adder with inputs x and y and forcing the outputs to be zero. The half-adder relations for carry and sum outputs c and s are $x \wedge y$, $x \oplus y$. The final encoding of $C = (c \leftrightarrow x \wedge y) \wedge (s \leftrightarrow x \oplus y) \wedge (\neg s \wedge \neg c)$ in CNF is $\phi = (\neg x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$. It is clear that x and y should be set to zero. But these values are not discovered by applying UP on ϕ .

One would naturally expect that the assignment α to the input variables, fully unit propagate to the output bits, but this may not always be the case and depends on the encoding ϕ .

2.2 Description of SHA-1

SHA-1 was designed by NSA and standardized by NIST in 1995 (see the standard in [14]). It was widely used in many applications, but after the recent full collision reported in [33], security practitioners moved away to stronger alternatives such as SHA-2 or SHA-3, although SHA-1 seems to be still resistant against preimage and second preimage attacks.

SHA-1 uses the Merkle-Damgård construction, where each block has 512 bits. Each block is given to a *compression function* that outputs 160 bits, which is used as part of the input to the next block. We recall only a part of the SHA-1 specification. For the full description of SHA-1, we refer to [14]. The internal state of SHA-1 is 160 bits. More precisely, five 32-bit words a_i, \dots, e_i for each round i . There are 80 rounds, and in each round a 32-bit message word W_i will be

mixed in to update the state bits. The *round function* for the round $i = 0, \dots, 79$ is defined as follows

$$(a_{i+1}, b_{i+1}, c_{i+1}, d_{i+1}, e_{i+1}) \leftarrow (F_i(b_i, c_i, d_i) \boxplus e_i \boxplus (a_i \lll 5) \boxplus W_i \boxplus K_i, a_i, b_i \lll 30, c_i, d_i), \tag{I-R}$$

where \lll is left rotation, \boxplus is addition modulo 2^{32} and K_i is the round constant. The function F_i is a Boolean map operating on three 32-bit words and generating a 32-bit word. This function changes every 20 rounds and will be one of these: $\text{Ch}(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$, $\text{XOR}(x, y, z) = x \oplus y \oplus z$ or $\text{Maj}(x, y, z) = (x \wedge y) \oplus (y \wedge z) \oplus (x \wedge z)$.

The *message expansion* relation for expanding the initial message words W_0, \dots, W_{15} from the 512 input bits to 32-bit message words for 80 rounds of SHA-1 is defined by

$$W_i = (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) \lll 1, \text{ for } i \in \{16, \dots, 79\}. \tag{I-M}$$

2.3 Description of SHA-256

SHA-256 is in the standard hash function family of SHA-2 [12]. Its structure is similar to SHA-1, but with a more complex round function and message expansion. The input block size is 512 bits and it has 64 rounds. Using the following *message expansion* relation, the 16 32-bit input words, will be expanded to 64 32-bit words.

$$W_i = \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}, \text{ for } i \in \{16, \dots, 63\}, \tag{II-M}$$

where

$$\begin{aligned} \sigma_0(x) &= (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3), \\ \sigma_1(x) &= (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10). \end{aligned}$$

The internal state is 256 bits consisting of eight 32-bit words labeled as a_i, b_i, \dots, h_i for each round i . The *state update relations* is described with the following equations:

$$(a_{i+1}, b_{i+1}, c_{i+1}, d_{i+1}, e_{i+1}, f_{i+1}, g_{i+1}, h_{i+1}) \leftarrow (T_1 + T_2, a_i, b_i, c_i, d_i + T_1, e_i, f_i, g_i) \tag{II-R}$$

with

$$\begin{aligned} T_1 &= h_i + \Sigma_1(e_i) + \text{Ch}(e_i, f_i, g_i) + K_i + W_i, \\ T_2 &= \Sigma_0(a_i) + \text{Maj}(a_i, b_i, c_i), \\ \Sigma_0(x) &= (x \ggg 2) \oplus (x \ggg 13) \oplus (x \ggg 22), \\ \Sigma_1(x) &= (x \ggg 6) \oplus (x \ggg 11) \oplus (x \ggg 25), \end{aligned}$$

where the functions Ch and Maj are the same as in SHA-1, K_i denotes the SHA-2 round constant, and W_i denotes the processed expanded message block.

3 Programmatic Interface in SAT Solvers

We call a SAT solver *programmatic* [17] if it is augmented with a set of callback functions that allow the user to add functionality to the solver's propagation and conflict analysis routines on the fly. The idea is inspired by the DPLL(T) architecture, in which a theory solver provides support for theory propagation and theory conflict analysis to the base Boolean DPLL solver. *Programmatic SAT solving* [17] is a particular variation of DPLL(T) which is more flexible and differs from the general concept as described in Sect. 1.1. The main advantage of using programmatic SAT is that it allows easy customization of the SAT solver to specific Boolean instances rather than an entire theory. The SAT developer thus has more fine-grained control over the power of SAT. This architecture has also been found useful in solving some problems in combinatorics [8], and much more effective than using a normal CNF encoding [9].

3.1 Programmatic Conflict Analysis

We are only interested in the values of message bits, which are a very small subset of all of the variables needed to encode the algebraic fault equation system into CNF. Whenever we solve the instance and find the message bits, we should check if it is a legitimate solution (hashes to the same correct hash output). Normally one could wait for the solver to finish solving the whole equation set and then check for the correctness, but we can do this verification as soon as the variables corresponding to the message bits are set. The sooner we reject a spurious solution, the faster the search process becomes. The programmatic conflict analysis callback is invoked when the solver's Boolean propagation routine reaches an inconclusive state or all of the variables are assigned, and there is no conflict. First it recovers the original input message bits, if all message bit variables are set, then hashes the input message bits and checks it against the correct hash output. In case of mismatch, a conflict clause that blocks the current spurious message bits will be returned to the solver. The solver has the reason clauses that led to this partial assignment, thus it can further optimize the returned clause using the implication graph, which makes the blocking clause more effective. The solver then goes through the procedure of backjumping, as in the typical conflict analysis.

3.2 Programmatic Propagation

It is known that when encoding a problem into CNF, we might lose some structural information about the original problem. For example, setting a subset of variables in a CSP instance might imply the value of another variable. But if the encoding of that CSP problem into CNF is not UP-maintaining GAC, then by setting the corresponding variables in the Boolean formula, BCP may not be able to derive the value for the target variables. An example of such an encoding is listed in Example 1 in Sect. 2. It is also mentioned in [13, 32] that encoding of a pseudo-Boolean constraint into CNF using adder networks does not maintain

GAC, although these encodings are small and scalable. To overcome this problem, one might use arc-consistent encodings for a particular constraint, or use enhanced propagation routines, e.g., bitvector propagators [34].

In this work, we deal with cryptographic functions having multi-operand additions in each round. There are several encodings for these operations in the literature. Nossum's encoding² [30] gives a very compact CNF, which works very well in practice. Unfortunately, a curious side effect of having this minimal encoding is that after setting all of the input bits, BCP might not be able to set all output bits. There are two options to work around this problem, either empower the encoding, or strengthen the unit propagation. Based on our initial experiments and experiments in [30] with straightforward Tseitin encoding of adders, empowered encoding of adders can become very expensive (reaching time limit on all instances). Better propagation (based on SHA1) would be effective no matter the encoding. We therefore explored the latter option in this paper.

Our programmatic propagation (PP) is called in the main search loop of the solver after BCP is done, and no conflicts are detected. The callback looks at the least significant bits of the operands and output in each of the multi-operand additions. If all bits up to some bit position k are set, it checks if the k least significant bits of the output are set as well. If they are not set, it returns clauses that encode the direct implication between input bits and output bit in the missing output bit positions. For an example of encoding implications, if $x = T$, $y = F$ is an assignment to the inputs of $z = x + y$ relation, and z is not set, we return $x = T \wedge y = F \rightarrow z = T$ or $\neg x \vee y \vee z$. These implications force the solver to set the output bits in the next cycle. Although our implementation finds more implications than unit propagation does, it is not guaranteed that every encoding PP-maintains GAC according to Definition 1.

Definition 2. *Let ϕ be a CNF encoding of a Boolean function f , and let R be an inference rule of propositional logic. We say that ϕ **R -maintains Input/Output GAC**³ if for an assignment α that contains assignments to the input variables of f , the assignment of the output variables of f are derived from $\phi \wedge \alpha$ by R .*

For example, a direct Tseitin encoding of a CIRCUIT-SAT instance to CNF has the property given in Definition 2. Our implementation of programmatic propagation looks at the inputs of the multi-operand addition and generates direct implications between input and output bits. If any subset of the input bits is set, and a subset of the output bits can be determined (through addition), those output bits are set either by unit propagation through formula clauses or through the direct implication clauses. Therefore we can say that any CNF encoding of multi-operand addition PP-maintains Input/Output GAC.

² A brief description about this encoding and our adaptation can be found here: <https://sites.google.com/view/crypto-sat/algebraic-fault-analysis>.

³ GAC refers to Generalized Arc-Consistency defined in Definition 1.

4 Algebraic Fault Attack on SHA-1 and SHA-2

Here we describe how the attack is mounted on the SHA-1 and SHA-256 compression functions, and where programmatic callbacks fit in. For encoding of SHA in CNF, we used an adapted version of Nossum's encoding [30].

4.1 Algebraic Fault Attack

In practice, faults are induced on a hardware implementation using a device that can generate perturbation, e.g., radiation, heat, laser, etc. The attacker chooses a specific register and applies the fault, which changes the input to the subsequent operations. The choice of which register to apply fault is important, and we refer to that register by *fault location*. The change to the targeted register's value is usually unknown. But with more sophisticated (and more expensive) devices it is possible to narrow down the number of bits in the state that the fault injector is affecting. Therefore the number of bits that can be flipped is a parameter that represents how strong is the attacker. The number of flipped bits shows the *hamming weight of the fault vector* applied, and is usually referred to as *fault model*. Another parameter in our AFA model is the *number of faults* that the attacker is capable of injecting. This parameter represents the cost of the attack, and thus the fewer injections the better.

In the algebraic setting, the transformations from the fault location to the output are encoded as constraints (in our case in CNF), and we refer to it as *correct equations*. For each injected fault, the transformations from the fault location to the output are again encoded but the output value is fixed to the corresponding faulty output, and we refer to them as *faulty equations*. The variables corresponding to the secret message bits are shared between all of these equation sets. Depending on the device that is used for fault injection, the attacker can assume an upper bound on the hamming weight of the difference between correct and faulty values of the fault location register. This can also be encoded as a constraint.

4.2 Attack Model

We assume that the attacker picks and knows the location of the fault, but does not have control over the value of the fault. We also assume that the chaining value at the input of the compression function is fixed to the initialization vector. Note that we do not perform actual hardware fault injections, and the process is simulated in software by XORing random values to the inner state variables.

4.3 Attack on SHA-1

In our attack described in Algorithm 1, we target the last 16 rounds of SHA-1. The message expansion is invertible, provided we have 16 consecutive words (see Eq. I-M). This means that recovering the last 16 expanded message words

Algorithm 1. AFA-SHA (An Algebraic Fault Attack on SHA)

Input: f : a SHA compression function, g : a reverse message expansion of SHA, d : the maximal weight of faults, L : a list of fault locations, k : the number of faults (k is divisible by $\#L$), H : the correct SHA hash image.

Output: M' : a message, such that $f(M') = H$.

```

1: function AFA( $f, g, d, L, k, H$ )
2:   Let  $M$  represent the embedded secret SHA message
3:   Let  $n$  be the number of rounds in  $f$ .
4:   Let  $\phi$  be a CNF encoding of  $H = f_{(n-15)..n}(x)$ .
5:    $\Phi := \phi$ 
6:   for  $\ell$  in  $L$  do
7:     for  $i = 1, \dots, k/\#L$  do
8:       Generate a random fault value  $\delta_i$  with  $w_H(\delta_i) \leq d$ .
9:        $H'_i := f_{(\ell+1)..n}(f_{1..\ell}(M) \oplus \delta_i)$  ▷ Calculate the faulty output
10:      Let  $\phi_i$  be a CNF encoding of  $H'_i = f_{(\ell+1)..n}(x \oplus \delta_i)$ .
11:       $\Phi := \Phi \wedge \bigwedge \phi_i$ .
12:   repeat
13:     Find a model  $\alpha$  for  $\Phi$ .
14:     Extract the assignment for  $W_{n-15}, \dots, W_n$  from  $\alpha$ .
15:     for  $j = n - 16, \dots, 1$  do
16:        $W_j := g(W_{j+1}, \dots, W_{j+16})$ 
17:        $M' := W_0 \parallel \dots \parallel W_{15}$ 
18:        $\Phi := \Phi \wedge \neg M'$ 
19:   until  $f(M') = H$ 
20:   return  $M'$ 
21:
22: function g-SHA-1( $W_0, \dots, W_{15}$ ) ▷ SHA-1 Message expansion in reverse
23:   return  $((W_{15} \ggg 1) \oplus W_{12} \oplus W_7 \oplus W_1)$  ▷ see Eq. I-M
24:
25: function g-SHA-2( $W_0, \dots, W_{15}$ ) ▷ SHA-256 Message expansion in reverse
26:   return  $(W_{15} - \sigma_1(W_{13}) - W_8 - \sigma_0(W_0))$  ▷ see Eq. II-M

```

enables us to recover all message bits. Therefore we inject faults to the input of the last 16 rounds, and more particularly in b_{64} . This fault location is more desirable because of the way the fault propagates in the next rounds. For more details we refer to [19].

Let f be the compression function of SHA-1. Let $f_{1..64}$ (resp. $f_{65..82}$) be the Boolean map representing the first 64 rounds of f (resp. the last 16 rounds of f). Thus we have the following composition $f = f_{65..80} \circ f_{1..64}$. Let M be a SHA-1 message. Consider the correct hash value $H = f(M)$. We can encode fault outputs as $H'_i = f_{65..80}(f_{1..64}(M) \oplus \delta_i)$, where δ_i is a random fault value. These are the steps that we follow:

- We obtain the correct hash output H and several faulty outputs H'_i for the given M .

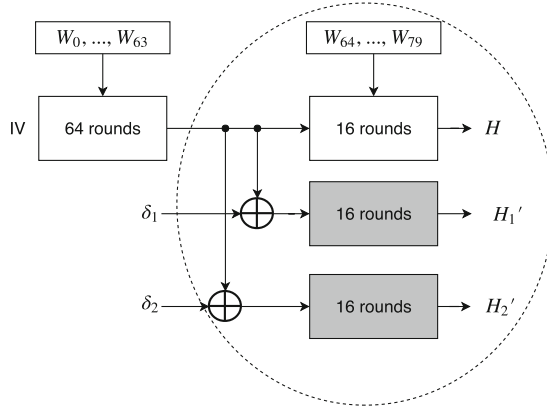


Fig. 1. A high-level diagram of the SHA-1 attack. The values δ_1 and δ_2 represent the injected faults. H denotes the correct hash output and H_1' and H_2' are the faulty outputs. The dashed circle is the part that is being encoded into CNF. The shaded boxes are copies of the white 16 rounds, and W_{64}, \dots, W_{79} variables are shared between all of them.

- Then we encode the set of correct and faulty equations for the last 16 rounds in CNF. Figure 1 shows the parts of the compression function that are being encoded into CNF.
- The composed formula Φ is then given to the SAT solver to find a solution for the last 16 message words.
- The verification loop is implemented in the SHA conflict analyzer. As soon as the corresponding variables to W_{65}, \dots, W_{80} are set, the analyzer will derive the first 16 message words M' by applying the Eq. I-M in reverse (see **g-SHA-1** in Algorithm 1). This value is given to the compression function to see if it hashes to H . If there is a match, the found M' is the final solution, otherwise the last 16 message words will be returned as conflict clauses to the SAT solver, and the search loop continues.

The attack is run by calling the **AFA** function from Algorithm 1 with the following arguments: $\text{AFA}(f_{SHA1}, d, \text{g-SHA-1}, L:\{64\}, k, H)$, where f_{SHA1} is the SHA-1 compression function.

4.4 Attack on SHA-256

Our attack on SHA-256 shares the same framework as in the SHA-1 attack. Our approach is outlined in Algorithm 1. Just like in the SHA-1 attack, we target the last 16 rounds of SHA-256 and the deepest fault location is c_{48} . For details on the impact of choosing this location, we refer to [18]. Since the state update operations in SHA-256 are more complex, the size of encoding is much bigger and the instances are harder to solve. We set a higher time limit and use a multi-stage fault injection approach to limit the effect of fault propagation.

Hao et al. [18] presented an AFA on SHA-256. They first target the last four rounds, inject faults and solve the equations to recover W_{61}, \dots, W_{64} . Then they fix the message words to the found solution and repeat the same procedure for the next four message words. Which means that with another set of fault injections, they recover W_{57}, \dots, W_{60} , and so on, to find the last 16 message words. We follow the same approach to keep the size of instances small. An immediate challenge in this approach is to check the consistency of the solutions for each set of four message words with the hash values. In our approach when we encode all of the relations from the fault injection location to the output, instead of solving the instances in each step, and fixing the solution in other instances, we conjunct all of the encoded instances together and let the solver handle the consistency of solutions. Following this method, we target the last four message words by injecting faults in round 60, and encoding the fault equations. Next we inject faults in round 56 to target the last 8 rounds. Similarly we target the last 12 rounds and last 16 rounds. All of the encoded fault equations together with the correct hash function relations for the last 16 rounds make our SAT instance. We inject the same number of faults in each of those fault locations. Similar to SHA-1, the verification process is implemented in the SHA conflict analysis callback, with the difference of using Eq. II-M (see g-SHA-2 in Algorithm 1), for deriving the first 16 words of M' and SHA-256 compression function is applied to check with the correct output H . Using the AFA function from Algorithm 1, the attack is launched with this call:

$\text{AFA}(f_{SHA2}, d, \text{g-SHA-2}, L:\{60, 56, 52, 48\}, k, H)$, where f_{SHA2} is the SHA-256 compression function.

5 Experimental Results

5.1 Experimental Setup

All experiments were conducted on Intel Xeon CPUs at 3.2 GHz and 16 GB of RAM. We used MapleSAT [24] to implement the programmatic callbacks. There are other SAT solvers like CryptoMiniSAT and lingeling that implement XOR reasoning which could be beneficial in solving ARX⁴ cryptographic functions like SHA-1 and SHA-2. Also SMT solvers that handle bitvectors, like STP, are a good candidate in solving these kinds of instances. But in practice, according to the study in [29], MapleSAT outperforms them on SHA-1 preimage instances. Because of the similarity of SHA-1 preimage instances to our fault instances we picked the best solver and implemented our programmatic interface in it. We have also decided to use the multi-armed bandit restart (MABR) policy [29] in MapleSAT, which adds an additional performance gain on cryptographic instances. We experimented with various assumptions on the number of the injected faults and on the maximal weight of the faults. For each experiment we generated 100 random message-target pairs, and the timeout was set at 4 h for SHA-1 instances and 12 h for SHA-256 instances. For the sake of completeness

⁴ Addition-Rotation-XOR.

and fair comparison, we have added an external loop around MapleSAT that does the verification (repeat-until loop in Algorithm 1) and adds blocking clauses to the solver if an inconsistent solution is found. In this section, whenever we mention the base version of MapleSAT, we mean MapleSAT with the verification loop.

5.2 Attack on SHA-1 and SHA-256

Table 1 shows the results of applying AFA on SHA-1 and SHA-2. Its rows correspond to the maximal weight of the injected faults. Its columns correspond to the number of injected faults during the attack. Starting from a single bit, going to a nibble, a single byte, single word and the most relaxed one is the 32-bit random fault model. Each element in Table 1 represents the number of instances out of 100 randomly generated AFA instances that our solver was able to solve within the time limit. From Table 1b we can see that we are able to recover the message bits with as few as 8 faults in the single byte fault model. In previous attacks on SHA-1, Hemme et al. [19], apply a DFA that uses 1002 faults. In the same fault model (32-bit fault model), we use only 11 faults.

Table 1. The number of solved AFA instances out of 100 for different number of faults and maximal weight of the faults

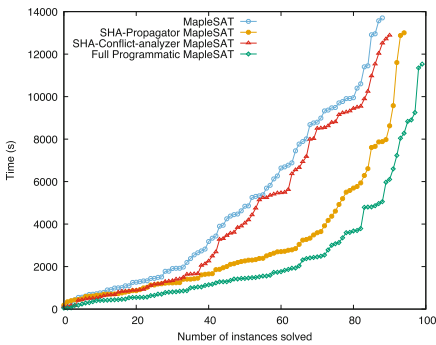
		(a) SHA-1						(b) SHA-256				
		Number of faults						Number of faults				
		8	11	12	16	20		32	40	48	56	
Fault weight	1	65	69	70	64	43	8	28	20	8	0	
	2	85	82	82	73	61	12	32	21	8	2	
	4	95	95	94	87	72	16	69	60	28	9	
	8	100	100	100	91	86	20	90	75	31	10	
	16	90	100	100	90	80	24	100	95	72	20	
	32	75	100	100	89	75	28	95	71	70	34	
								32	71	82	100	48

As described in Sect. 4.4, we inject faults in four different rounds and collect information about the correct and faulty hashes. We experimented with equal number of faults in each of those four rounds. As listed in Table 1b, we were able to recover the target bits using 32 faults in the 24-bit fault model. While Hao et al. [18] use 65 faults in a 32-bit random fault model, our method is able to finish the search with 48 faults in the same fault model. These two data points are highlighted in Table 1b.

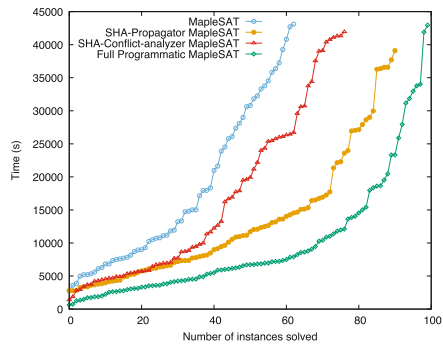
5.3 Performance of the Solver

Here we discuss the performance of our programmatic AFA solver on solving SHA fault instances. In Fig. 2 you can see the cactus plot of MapleSAT solver and the extended versions of MapleSAT with programmatic interface. We have turned

each of the programmatic callbacks on and off to see which of them contributes more to the performance of solver. There are four solvers compared in the plot. The base version of MapleSAT, MapleSAT with the SHA propagator, MapleSAT with SHA conflict analyzer and MapleSAT with both of these callbacks. We also experimented with Opturion CPX [31], which is a constraint solver that combines CP and SAT solving techniques, and won several medals in Minizinc challenge 2015. But unfortunately it performed very poorly on our benchmark, and could solve only a few of instances of 32-bit fault model. The timings in the plot belong to the 32-bit fault model with 11 faults injected in SHA-1, and 48 faults in SHA-2. The plot shows that by embedding the external verification loop inside the SHA conflict analyzer and early detection of inconsistent solutions (rather than waiting for the instance to be completely solved), we can solve two more instances in SHA-1 and 14 more instances in SHA-2. But the main performance boost belongs to the propagation enhancement, in which the solver solves 6 more instances within the time limit in SHA-1, and 28 more instance in SHA-2. From the point of view of number of faults, the lowest number of faults that base version of MapleSAT can recover the secret bits for all of the random messages, is 14, where with the programmatic MapleSAT, it is 11. For the case of SHA-2 the gap is larger and base version needs at least 64 faults, versus 48 faults needed by programmatic MapleSAT. Comparing the total timings for solving all of the instances in a fault model, between MapleSAT and programmatic MapleSAT, if we set the runtime of timed out instances to the time limit, we can see a 2.48x speedup in SHA-1 and 7.73x speedup in SHA-256. If we use the PAR-2 method (penalizing the timed out instances by setting their runtime to double the time limit), which was used in SAT competition 2017, we see a 3.16x speedup in SHA-1 and 14.3x speedup in SHA-2.



(a) 32-bit fault model AFA on SHA-1



(b) 32-bit fault model AFA on SHA-2

Fig. 2. The cactus plot comparing MapleSAT with the MapleSAT after adding each of the programmatic callbacks. Each data point (X, Y) on this plot means X fault instances are solved under Y seconds.

5.4 Discussion

Our results show the versatility of programmatic SAT solver architecture. The key insight is that by taking a state-of-the-art general purpose SAT solver and tailoring it to our cryptographic problem, we achieved considerable performance improvement. Looking at Table 1, one can observe that the data in certain rows suggests that when more faults are injected, fewer instances are solved. At first it might seem counterintuitive because adding more faults helps restrict the search space and hence should improve solver performance. However, note also that with every added fault equation, the number of clauses in the input to the solver grows rapidly (especially in the case of SHA-256), which can crucially slow down propagation. Thus there is a tradeoff between search space reduction and formula size that the cryptanalyst has to contend with.

6 Related Work

The research on fault attacks on SHA-like cryptographic structures was started by Li et al. [23], where they applied a DFA on SHACAL-1, a block cipher based on the structure of SHA-1. Hemme et al. [19] extended their attack to SHA-1. The challenge of applying DFA on SHA-1 is the following: after applying the compression function of SHA-1 on the initialization vector (IV) and the message words, the value of IV is added to the output to make the chaining value for the next block. Hemme et al. handled this addition layer with separate fault injections, and then launched an attack similar to [23]. This is a key reason as to why their attack needs more than thousand faults to be applicable. Our results show that an AFA can succeed with far fewer number of injected faults. In the same fault model of 32 bits, we can find the secret bits with 11 faults.

Jeong et al. [22] proposed a fault attack on the HMAC setting of SHA-2 and showed that key values of size n can be recovered with approximately $n/3$ faults. Hao et al. [18] presented an AFA on SHA-2. They first perform a round of fault injections to recover the last internal state before the final addition. Then they inject some more faults, encode and solve 4 rounds of SHA-256 at a time, fixing the found values at each step for the next solving step. This approach keeps the size of each fault instance small, but the problem is that if the found solution is inconsistent with the chaining input and correct hash value in the final solution, there is no comeback and no fixing mechanism is used. They use 65 faults in total for recovering the last 16 message words and hence full state recovery of SHA-256. They use STP [16] for solving the algebraic equations. For the same fault model, we can recover the secret bits with far fewer faults than their work. In the 32-bit fault model, we achieve the same results with 48 faults.

7 Conclusion and Future Work

We present a new approach to algebraic fault-injection attack on SHA-1 and SHA-2, based on a programmatic version of the MapleSAT solver with SHA-enhanced conflict clause analysis and propagation tailored to the AFA setting.

We evaluated our attack under different assumptions, i.e., the number of injected faults and the maximal weight of faults, and presented results showing that we can recover the secret bits with far fewer injected faults compared to the previous best fault attack methods aimed at SHA-1 and SHA-2. Our programmatic solver can achieve a speedup of up to 14x compared to the baseline solver. Furthermore, our method scales much better than previous AFA approaches. In the future we plan to extend our work to SHA-3, HMAC, and block ciphers.

Acknowledgments. The authors would like to thank Jia Hui Liang for his support with MapleSAT. The second author was financially supported by the DFG project “Algebraische Fehlerangriffe” [KR 1907/6-2].

References

1. Agoyan, M., Dutertre, J.-M., Naccache, D., Robisson, B., Tria, A.: When clocks fail: on critical paths and clock faults. In: Gollmann, D., Lanet, J.-L., Iguchi-Cartigny, J. (eds.) CARDIS 2010. LNCS, vol. 6035, pp. 182–193. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12510-2_13
2. Ali, S.S., Mukhopadhyay, D., Tunstall, M.: Differential fault analysis of AES: towards reaching its limits. *J. Crypt. Eng.* **3**(2), 73–97 (2013)
3. Bailleux, O., Boufkhad, Y., Roussel, O.: New encodings of pseudo-boolean constraints into CNF. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 181–194. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_19
4. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer’s apprentice guide to fault attacks. *Proc. IEEE* **94**(2), 370–382 (2006)
5. Barengi, A., Breveglieri, L., Koren, I., Naccache, D.: Fault injection attacks on cryptographic devices: theory, practice, and countermeasures. *Proc. IEEE* **100**(11), 3056–3076 (2012)
6. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Kaliski, B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 513–525. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0052259>
7. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-69053-0_4
8. Bright, C., Ganesh, V., Heinle, A., Kotsireas, I., Nejati, S., Czarnecki, K.: MATH-CHECK2: A SAT+CAS verifier for combinatorial conjectures. In: Gerdt, V.P., Koepf, W., Seiler, W.M., Vorozhtsov, E.V. (eds.) CASC 2016. LNCS, vol. 9890, pp. 117–133. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45641-6_9
9. Bright, C., Kotsireas, I., Ganesh, V.: A SAT+CAS method for enumerating Williamson matrices of even order. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, 2–7 February 2018, pp. 6573–6580 (2018)
10. Courtois, N.T., Jackson, K., Ware, D.: Fault-algebraic attacks on inner rounds of DES. In: e-Smart 2010 Proceedings: The Future of Digital Security Technologies. Strategies Telecom and Multimedia (2010)

11. Dobraunig, C., Eichlseder, M., Mendel, F.: Analysis of SHA-512/224 and SHA-512/256. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015. LNCS, vol. 9453, pp. 612–630. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48800-3_25
12. Eastlake 3rd, D., Hansen, T.: US secure hash algorithms (SHA and SHA-based HMAC and HKDF). Technical report (2011)
13. Eén, N., Sorensson, N.: Translating pseudo-boolean constraints into SAT. *J. Satisf. Boolean Model. Comput.* **2**, 1–26 (2006)
14. FIPS Publication: 180–4. Federal Information Processing Standards Publication, Secure Hash (2011)
15. Fischer, W., Reuter, C.A.: Differential fault analysis on Grøstl. In: 2012 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 44–54. IEEE (2012)
16. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_52
17. Ganesh, V., O’Donnell, C.W., Soos, M., Devadas, S., Rinard, M.C., Solar-Lezama, A.: Lynx: a programmatic SAT solver for the RNA-folding problem. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 143–156. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_12
18. Hao, R., Li, B., Ma, B., Song, L.: Algebraic fault attack on the SHA-256 compression function. *Int. J. Res. Comput. Sci.* **4**(2), 1 (2014)
19. Hemme, L., Hoffmann, L.: Differential fault analysis on the SHA-1 compression function. In: 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 54–62. IEEE (2011)
20. Hojsík, M., Rudolf, B.: Differential fault analysis of trivium. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 158–172. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71039-4_10
21. Jeong, K., Lee, C.: Differential fault analysis on block cipher LED-64. In: (Jong Hyuk) Park, J.J., Leung, V., Wang, C.L., Shon, T. (eds.) Future Information Technology, Application, and Service. LNEE, vol. 164, pp. 747–755. Springer, Dordrecht (2012). https://doi.org/10.1007/978-94-007-4516-2_79
22. Jeong, K., Lee, Y., Sung, J., Hong, S.: Security analysis of HMAC/NMAC by using fault injection. *J. Appl. Math.* **2013**, 6 (2013)
23. Li, R., Li, C., Gong, C.: Differential fault analysis on SHACAL-1. In: 2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 120–126. IEEE (2009)
24. Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 123–140. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_9
25. Luo, P., Athanasiou, K., Fei, Y., Wahl, T.: Algebraic fault analysis of SHA-3. In: 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 151–156. IEEE (2017)
26. Luo, P., Fei, Y., Zhang, L., Ding, A.A.: Differential fault analysis of SHA3-224 and SHA3-256. In: 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 4–15. IEEE (2016)
27. Menezes, A.J., Van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
28. Mohamed, M.S.E., Bulygin, S., Buchmann, J.: Improved differential fault analysis of Trivium. In: COSADE 2011, pp. 147–158 (2011)

29. Nejati, S., Liang, J.H., Gebotys, C., Czarnecki, K., Ganesh, V.: Adaptive restart and CEGAR-based solver for inverting cryptographic hash functions. In: Paskevich, A., Wies, T. (eds.) VSTTE 2017. LNCS, vol. 10712, pp. 120–131. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72308-2_8
30. Nossum, V.: SAT-based Preimage Attacks on SHA-1 (2012)
31. Opturion: Opturion CPX 1.0.2. <http://cpx.opturion.com/cpx.html>. Accessed 30 Mar 2018
32. Philipp, T., Steinke, P.: PBLib – a library for encoding pseudo-boolean constraints into CNF. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 9–16. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_2
33. Stevens, M., Bursztein, E., Karpman, P., Albertini, A., Markov, Y.: The first collision for full SHA-1. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10401, pp. 570–596. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63688-7_19
34. Wang, W., Søndergaard, H., Stuckey, P.J.: A bit-vector solver with word-level propagation. In: Quimper, C.-G. (ed.) CPAIOR 2016. LNCS, vol. 9676, pp. 374–391. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33954-2_27
35. Zhang, F., Zhao, X., Guo, S., Wang, T., Shi, Z.: Improved algebraic fault analysis: a case study on piccolo and applications to other lightweight block ciphers. In: Prouff, E. (ed.) COSADE 2013. LNCS, vol. 7864, pp. 62–79. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40026-1_5
36. Zhao, X., Guo, S., Zhang, F., Shi, Z., Ma, C., Wang, T.: Improving and evaluating differential fault analysis on LED with algebraic techniques. In: 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 41–51. IEEE (2013)



Correction to: PW-CT: Extending Compact-Table to Enforce Pairwise Consistency on Table Constraints

Anthony Schneider and Berthe Y. Choueiry

Correction to:
Chapter “PW-AC: Extending Compact-Table to Enforce Pairwise Consistency on Table Constraints”
in: J. Hooker (Ed.): *Principles and Practice of Constraint Programming*, LNCS 11008,
https://doi.org/10.1007/978-3-319-98334-9_23

The original version of the chapter was revised. The title has been corrected.

The updated online version of this chapter can be found at
https://doi.org/10.1007/978-3-319-98334-9_23

© Springer Nature Switzerland AG 2018
J. Hooker (Ed.): CP 2018, LNCS 11008, p. E1, 2018.
https://doi.org/10.1007/978-3-319-98334-9_48



Correction to: MLIC: A MaxSAT-Based Framework for Learning Interpretable Classification Rules

Dmitry Malioutov and Kuldeep S. Meel

Correction to:
Chapter “MLIC: A MaxSAT-Based Framework for Learning Interpretable Classification Rules” in: J. Hooker (Ed.):
Principles and Practice of Constraint Programming,
LNCS 11008, https://doi.org/10.1007/978-3-319-98334-9_21

In the original version of this paper there was a typing error in the family name of the first author. “Dmitry Maliotov” should have been “Dmitry Malioutov”. This has now been corrected.

The updated version of this chapter can be found at
https://doi.org/10.1007/978-3-319-98334-9_21

© Springer Nature Switzerland AG 2019
J. Hooker (Ed.): CP 2018, LNCS 11008, p. C1, 2018.
https://doi.org/10.1007/978-3-319-98334-9_49

Abstracts

Encoding Cardinality Constraints Using Multiway Merge Selection Networks

Michał Karpiński^(✉) and Marek Piotrów

Institute of Computer Science, University of Wrocław,
Joliot-Curie 15, 50-383 Wrocław, Poland
`{karp,mpi}@cs.uni.wroc.pl`

Abstract. Boolean cardinality constraints (CCs) state that at most (at least, or exactly) k out of n propositional literals can be true. We propose a new, arc-consistent, easy to implement and efficient encoding of CCs based on a new class of selection networks. Several comparator networks have been recently proposed for encoding CCs and experiments have proved their efficiency. In our construction we use the idea of the multiway merge sorting networks by Lee and Batchier that generalizes the technique of odd-even sorting ones by merging simultaneously more than two subsequences. The new selection network merges 4 subsequences in that way. Based on this construction, we can encode more efficiently comparators in the combine phase of the network: instead of encoding each comparator separately by 3 clauses and 2 additional variables, we propose an encoding scheme that requires 5 clauses and 2 variables on average for each pair of comparators. We also extend the model of comparator networks so that the basic components are not only comparators (2-sorters) but more general m -sorters, for $m \in \{2, 3, 4\}$, that can also be encoded efficiently. We show that with small overhead (regarding implementation complexity) we can achieve a significant improvement in SAT-solver runtime for many test cases. We prove that the new encoding is competitive to the other state-of-the-art encodings.

We present a detailed description of 4-way merge selection networks, where a set of n inputs is split into 4 subsets, from which the k largest elements are recursively selected and the results are merged to obtain the k largest items of the set. We show how to translate such a network into an efficient CNF encoding of a less-than- k cardinality constraint over n literals and estimate the numbers of variables/clauses used in it. We calculate that the CNF encoding based on our network is smaller than the CNF encoding based on the previously-known 2-way merge version. Finally, the new encoding is compared with the selected state-of-the-art encodings based on comparator networks, adders and binary decision diagrams as well as with two popular general constraints solvers – PBLIB and NPSOLVER. The experimental evaluation shows that the new encoding yields better speed-up and overall runtime in the SAT-solver performance.

Full paper to be published in *Constraints*.

The construction is parametrized by any values of k and n , so they can be further optimized by mixing them with other constructions. For example, in our experiments we mixed them with the direct encoding of subproblems with small values of the parameters.

Not All FPRASs Are Equal: Demystifying FPRASs for DNF-Counting (Extended Abstract)

Kuldeep S. Meel¹, Aditya A. Shrotri²(✉), and Moshe Y. Vardi²

¹ National University of Singapore, Singapore, Singapore
meel@comp.nus.edu.sg

² Rice University, Houston, USA
{as128, vardi}@rice.edu

Constrained counting is a fundamental problem in artificial intelligence with a wide variety of applications ranging from network reliability, probabilistic inference, quantified information flow, and the like. We focus on the variant of constrained counting known as DNF-Counting or #DNF, in which the constraints are expressed in Disjunctive Normal Form (DNF). #DNF is important in practice, as problems like query evaluation in probabilistic databases and failure-probability estimation of networks reduce to it. Due to the intractability of the exact version, however, efforts have focused on the design of approximate techniques for #DNF. Consequently, several Fully Polynomial Randomized Approximation Schemes (FPRASs) based on Monte Carlo techniques have been proposed [1]. Recently, it was discovered that hashing-based techniques too lend themselves to FPRASs for #DNF. Despite significant improvements, the complexity of the hashing-based FPRAS is still worse than that of the best Monte Carlo FPRAS by polylog factors [2]. Two questions were left unanswered in previous works: Can the complexity of the hashing-based techniques be improved? How do the various approaches stack up against each other empirically?

In this paper, we first propose a new search procedure for the hashing-based FPRAS that removes the polylog factors from its time complexity. We then present the first empirical study of runtime behavior of different FPRASs for #DNF. We tested the FPRASs across parameters such as formula-size, tolerance and confidence. We measured running time, accuracy, scalability and the number of instances solved. The result of our study produces a nuanced picture. Firstly, we observe that there is no single best algorithm that outperforms all others for all classes of formulas and parameters. Second, we observe that the worst-case complexity analysis is a poor guide to the actual performance of the algorithm.

References

1. Karp, R.M., Luby, M.: Monte-carlo algorithms for enumeration and reliability problems. In: Proceedings of FOCS (1983)
2. Meel, K.S., Shrotri, A.A., Vardi, M.Y.: On hashing-based approaches to approximate DNF-counting. In: Proceedings of FSTTCS (2017)

Full version to appear in Constraints.

Constraint Games for Stable and Optimal Allocation of Demands in SDN

Anthony Palmieri^{1,2}, Arnaud Lallouet^{1,2(✉)}, and Luc Pons¹

¹ Huawei Technologies Ltd., French Research Center, Boulogne-Billancourt, France
{anthony.palmieri, arnaud.lallouet, luc.pons}@huawei.com

² GREYC - Université de Caen, Normandie, France

Software Defined Networking (or SDN) allows to apply a centralized control over a network of commuturs in order to provide better global performances [1]. One of the problem to solve in this context is the multi-commodity path routing (MCPRP) where a set of demands have to be routed at minimum cost in presence of congestion. We study centralized routing with Constraint Programming and selfish routing with Constraint Games. Selfish routing is important for the perceived quality of the solution since no user is able to improve his cost by changing only his own path. A common measure of the loss induced by a decentralized equilibrium with respect to a centralized optimum is the Price of Anarchy.

For each demand a single route from the source to the destination node is to be computed such that the sum of bandwidth routed by a link does not exceed its capacity. Solving a MCPRP to optimality means finding a solution minimizing the global cost of the demands. For the problem of finding a Nash equilibrium, each demand is considered as a player and its individual solution path should also be optimal.

We have used a constraint path model (var array with alldiff + subpath) and several heuristics based on Dijkstra's shortest path to ensure that the search space will be explored in a meaningful way. In addition to the heuristics, a specific lower bound for the remaining sub-problem is computed and used in the branch and bound procedure to drastically cut the search space. Selfish routing is modeled using Constraint Games and the solver Conga [2].

Results on synthetic and real instances show that the performance of CP with the given heuristics and branch and bound is able to solve the classical industrial instances. Results on games show that the Price of Anarchy is close to 1 for many instances, meaning that decentralized algorithms are likely to perform well for this type of problems. Moreover, Constraint games allow to solve up to optimality meaningful games of unprecedented size up to more than a thousands of players with a large space of strategies.

This article will be published in Constraints.

References

1. Mendiola, A., Astorga, J., Jacob, E., Higuero, M.: A survey on the contributions of software-defined networking to traffic engineering. *IEEE Commun. Surv. Tutor.* **19**(2), 918–953 (2017). <https://doi.org/10.1109/COMST.2016.2633579>
2. Palmieri, A., Lallouet, A.: Constraint games revisited. In: Sierra, C. (ed.) *IJCAI 2017*, Melbourne, Australia. pp. 729–735 (2017). ijcai.org

Author Index

- Aïmeur, Esma 229
Akgün, Özgür 3, 362, 727
Amadini, Roberto 13
Ansotegui, Carlos 524
- Belov, Gleb 473
Bessiere, Christian 537
Birnbaum, Philippe 507
Bit-Monnot, Arthur 30
Bono, Massimo 47
Boughaci, Dalila 229
Briot, Nicolas 389
- Cappart, Quentin 490
Choueiry, Berthe Y. 345
Chu, Geoffrey 99
Cohen, David A. 64
Cohen, Liel 669
Cooper, Martin C. 64
Cruz, Waldemar 259, 613
Czarnecki, Krzysztof 436, 453
Czauderna, Tobias 473
- de la Banda, Maria Garcia 81, 403, 473
Dekker, Jip J. 81
Demirović, Emir 99
Dlala, Imen Ouled 554, 570
- Egly, Uwe 276
- Fichte, Johannes K. 109
Fioretto, Ferdinando 688
Fossé, Rohan 128
- Ganesh, Vijay 436, 453, 737
Gange, Graeme 13, 144, 649
Gebotys, Catherine 737
Gent, Ian P. 3, 727
Gerevini, Alfonso Emilio 47
Glorian, Gael 160
Gomes, Carla 601
- He, Shan 649
Hebrard, Emmanuel 179
- Hecher, Markus 109
Hoang, Khoi D. 688
Hoos, Holger H. 195
Horáček, Jan 737
- Ingmar, Linnea 210
- Jabbour, Said 554, 570
Jefferson, Christopher 3, 727
Jguirim, Wafa 64
Joshi, Saurabh 219
Justeau-Allaire, Dimitri 507
- Karpiński, Michał 757
Katsirelos, George 179
Khelifa, Meriem 229
Klapperstueck, Matthias 473
Koenig, Sven 588
Kumar, Prateek 219
Kumar, T. K. Satish 588
- Lagniez, Jean-Marie 160
Lallouet, Arnaud 760
Lazaar, Nadjib 537
Lee, Jasper C. H. 242
Lee, Jimmy H. M. 242
Leo, Kevin 403
Levit, Vadim 707
Liang, Jia Hui 436, 453
Liebman, Ariel 649
Liu, Fanghui 259, 613
Lodha, Neha 109
Lonsing, Florian 276
Lorca, Xavier 507
- Maamar, Mehdi 537
Madelaine, Florent 295
Malioutov, Dmitry 312
Mana, Fatima Ezzahra 554
Martins, Ruben 219, 436, 453
Meel, Kuldeep S. 312, 759
Meisels, Amnon 707
Michel, Laurent 259, 613
Miguel, Ian 3, 362, 727

- Miné, Antoine 420
 Montmirail, Valentin 160
- Nejati, Saeed 737
 Nightingale, Peter 3, 727
- Palmieri, Anthony 328, 760
 Peitl, Tomáš 195
 Pelleau, Marie 420
 Perez, Guillaume 328, 601
 Piotrów, Marek 757
 Pons, Luc 760
 Pontelli, Enrico 688
- Raddaoui, Badran 554, 570
 Rao, Sukrut 219
 Rappazzo, Brendan 601
 Robere, Robert 453
 Rousseau, Louis-Martin 490
- Sais, Lakhdar 554, 570
 Salamon, András Z. 3
 Sarigiannidis, Panagiotis G. 373
 Schaus, Pierre 490
 Schneider, Anthony 345
 Schulte, Christian 210
 Schutt, Andreas 81
 Secouard, Stéphane 295
 Sellmann, Meinolf 524
 Senthoooran, Ilankaikone 473
 Shrotri, Aditya A. 759
 Simon, Laurent 128
- Sioutis, Michael 160
 Slivovsky, Friedrich 195
 Smith, Mitch 473
 Spracklen, Patrick 362
 Stergiou, Kostas 373
 Stuckey, Peter J. 13, 81, 99, 144
 Szeider, Stefan 109, 195
- Tack, Guido 81, 403
 Tesch, Alexander 629
 Thomas, Charles 490
 Tierney, Kevin 524
 Truchet, Charlotte 420
 Tsouros, Dimosthenis C. 373
- Vardi, Moshe Y. 759
 Vismara, Philippe 389
- Wallace, Mark 473, 649
 Wilson, Campbell 649
 Wintersteiger, Christoph M. 436, 453
 Wybrow, Michael 473
- Xu, Hong 588
- Yeoh, William 688
- Zeighami, Kiana 403
 Zhong, Allen Z. 242
 Ziat, Ghiles 420
 Zivan, Roie 669, 688
 Zulkoski, Edward 436, 453